Université Paris-Dauphine

Habilitation à Diriger des Recherches

Vérification automatique de protocoles cryptographiques : modèle formel et modèle calculatoire

Bruno Blanchet
CNRS, École Normale Supérieure, INRIA
Bruno.Blanchet@ens.fr

soutenue le 26 novembre 2008

Président du jury :	Jacques Stern
Rapporteurs :	Andrew Gordon Jean Goubault-Larrecq Serge Vaudenay
Examinateurs :	Ralf Küsters Mark Ryan
Coordinateur :	Vangelis Paschos
Directeur de recherches :	Patrick Cousot

Remerciements

Je tiens tout d'abord à remercier les membres de mon jury d'habilitation. Jacques Stern m'a fait l'honneur de présider mon jury. Je l'en remercie tout particulièrement. Patrick Cousot a été le directeur de mes travaux depuis ma thèse, après avoir été mon directeur de thèse. Je le remercie particulièrement pour la grande liberté scientifique qu'il m'a accordée, sans laquelle je n'aurais jamais pu réaliser les travaux présentés ici. Andrew Gordon, Jean Goubault-Larrecq et Serge Vaudenay ont été les rapporteurs de ce mémoire. Je tiens à les remercier pour le travail considérable qu'ils ont effectué. Ralf Küsters, Vangelis Paschos et Mark Ryan ont bien voulu faire partie de jury d'habilitation ; je les en remercie chaleureusement. Je remercie également Andrew Gordon, Ralf Küsters, Mark Ryan et Serge Vaudenay d'avoir fait le voyage depuis l'étranger pour ma soutenance.

Une grande partie de mon travail depuis ma thèse a été réalisée en collaboration avec des co-auteurs. Je les remercie pour leurs contributions qui ont considérablement enrichi mon travail : Martín Abadi, Xavier Allamigeon, Benjamin Aziz, Avik Chaudhuri, Patrick Cousot, Radhia Cousot, Jérôme Feret, Cédric Fournet, Aaron D. Jaggard, Laurent Mauborgne, Antoine Miné, David Monniaux, Andreas Podelski, David Pointcheval, Xavier Rival, Andre Scedrov et Joe-Kai Tsay.

Je tiens à remercier en particulier Martín Abadi : c'est essentiellement grâce à lui que j'ai réalisé le travail présenté ici. Il est un des pionniers de la vérification des protocoles cryptographiques, et j'ai eu la chance d'effectuer sous sa direction un stage de deux mois à Bell Labs Research, Palo Alto. Ce stage a considérablement influencé la suite de ma recherche : il a été le point de départ de mon travail sur la vérification des protocoles cryptographiques présenté dans ce mémoire, et aussi d'une collaboration fructueuse avec Martín Abadi qui s'est poursuivie depuis.

Je remercie particulièrement Jacques Stern pour avoir initié mon travail sur la vérification des protocoles cryptographiques dans le modèle calculatoire, et David Pointcheval pour m'avoir patiemment expliqué les preuves calculatoires des protocoles. Le travail présenté dans le chapitre 3 n'aurait pas existé sans eux.

Je remercie aussi tous les utilisateurs de mes logiciels ProVerif et CryptoVerif qui, par leurs remarques pertinentes, ont contribué à améliorer ces logiciels et m'ont encouragé à poursuivre leur développement.

Mes remerciements vont également à tous les membres des laboratoires dans lesquels j'ai effectué ces travaux : Bell Labs Research, Palo Alto (août-octobre 2000), le projet Moscova de l'INRIA Rocquencourt (jusqu'en septembre 2001), le département d'informatique de l'École normale supérieure (à partir d'octobre 2001), et le Max-Planck-Institut für Informatik (novembre 2001-août 2004). Ils m'ont accueilli dans une ambiance très agréable et m'ont fourni d'excellentes conditions pour réaliser mon travail. Je remercie en particulier Martín Abadi à Bell Labs Research ; Jean-Jacques Lévy, Alain Deutsch et Sylvie Loubressac à l'INRIA ; tous les membres passés et présents de l'équipe Interprétation Abstraite ainsi que Joëlle Isnard, Michèle Angely, Lise-Marie Bivard, Sylvia Imbert, Valérie Mongiat et le Service de Prestations Informatiques à l'ENS ; Harald Ganzinger, Andreas Podelski et Ellen Fries au MPI.

Ce travail a été partiellement soutenu par le projet ANR (Agence Nationale de la Recherche) ARA SSIA FormaCrypt.

Avant-propos

Ce mémoire d'habilitation présente une synthèse des travaux que j'ai effectués depuis ma thèse de doctorat. Ces travaux ont principalement concerné la vérification automatique de protocoles cryptographiques.

Le premier chapitre présente une brève introduction aux protocoles cryptographiques, et situe mon travail dans l'abondante littérature sur la vérification des protocoles cryptographiques. Le deuxième chapitre traite du vérificateur automatique de protocoles ProVerif, fondé sur le modèle formel des protocoles. Le troisième chapitre traite quant à lui du vérificateur CryptoVerif, qui est fondé sur le modèle calculatoire des protocoles. La conclusion présente quelques perspectives de recherche dans ce domaine. Enfin, le chapitre 5 résume mes activités d'enseignement et d'encadrement.

En annexe, vous trouverez un curriculum vitae détaillé avec ma liste de publications, ainsi que quatre de mes publications parmi les plus importantes.

En dehors de mon travail sur les protocoles cryptographiques, j'ai également participé, de novembre 2001 à novembre 2003, au projet Astrée sur la vérification de programmes C temps réel embarqués critiques, avec Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival. Ce projet a donné lieu à la réalisation de l'analyseur Astrée (`www.astree.ens.fr`) qui est capable de prouver automatiquement l'absence d'erreurs à l'exécution dans des programmes de plusieurs centaines de milliers de lignes [BCC+02, BCC+03]. Ce travail n'est pas détaillé dans ce mémoire.

Table des matières

1 Introduction **1**
1.1 Protocoles cryptographiques . . . 1
1.1.1 Primitives cryptographiques . . . 1
1.1.2 Un exemple de protocole . . . 3
1.1.3 Des protocoles pour des applications variées . . . 4
1.1.4 Intérêt de la vérification formelle . . . 5
1.2 Modèles des protocoles . . . 6
1.3 Propriétés de sécurité . . . 6
1.3.1 Propriétés de trace et propriétés d'équivalence . . . 6
1.3.2 Secret . . . 7
1.3.3 Authentification . . . 7
1.4 Vérification des protocoles dans le modèle formel . . . 8
1.4.1 Propriétés de trace (secret, authentification, ...) . . . 8
1.4.2 Propriétés d'équivalence . . . 12
1.5 Lien entre modèles formel et calculatoire . . . 13
1.6 Preuve des protocoles dans le modèle calculatoire . . . 14
1.7 Conclusion . . . 16

2 Vérification des protocoles dans le modèle formel **17**
2.1 Représentation formelle des protocoles cryptographiques . . . 17
2.1.1 Historique . . . 17
2.1.2 Un langage de représentation des protocoles . . . 18
2.1.3 Un exemple de protocole dans ce langage . . . 20
2.1.4 Sémantique . . . 21
2.1.5 Extension aux théories équationnelles . . . 21
2.2 Les clauses de Horn . . . 22
2.2.1 Définition du secret . . . 23
2.2.2 Du pi calcul vers les clauses de Horn . . . 23
2.2.3 Résolution sur les clauses . . . 28
2.2.4 Vérification des propriétés de correspondances . . . 31
2.2.5 Scénarios à plusieurs phases . . . 33
2.2.6 Preuves d'équivalences . . . 34
2.3 Résultats . . . 37
2.4 Conclusion . . . 38

3 Vérification des protocoles dans le modèle calculatoire **39**
3.1 Langage de représentation des jeux . . . 40
3.2 Équivalence observationnelle . . . 44
3.3 Transformations de jeux . . . 44
3.3.1 Transformations syntaxiques . . . 44
3.3.2 Utiliser les hypothèses de sécurité sur les primitives . . . 45

3.4 Propriétés de sécurité . . . 49
3.4.1 Secret . . . 49
3.4.2 Correspondances . . . 50
3.5 Stratégie de preuve . . . 52
3.6 Résultats . . . 53
3.7 Conclusion . . . 54

4 Conclusion et perspectives **55**

5 Activités d'enseignement et d'encadrement **57**
5.1 Enseignement . . . 57
5.1.1 Travaux dirigés à l'École polytechnique . . . 57
5.1.2 Travaux dirigés à l'ENSTA . . . 57
5.1.3 Travaux dirigés à l'Université de Versailles . . . 57
5.1.4 Cours en DEA et Master . . . 58
5.2 Encadrement . . . 58
5.2.1 Reconstruction d'attaques contre des protocoles cryptographiques . . . 58
5.2.2 Analyse de protocoles présentés comme une liste de messages . . . 59
5.2.3 Analyse d'implantations de protocoles cryptographiques en Java . . . 59

Bibliographie **61**

A Curriculum vitae **81**

B Articles joints **89**
B.1 *Analyzing Security Protocols with Secrecy Types and Logic Programs*
Martín Abadi et Bruno Blanchet . . . 91
B.2 *Automatic Verification of Correspondences for Security Protocols*
Bruno Blanchet . . . 135
B.3 *Automated Verification of Selected Equivalences for Security Protocols*
Bruno Blanchet, Martín Abadi et Cédric Fournet . . . 213
B.4 *A Computationally Sound Mechanized Prover for Security Protocols*
Bruno Blanchet . . . 273

Chapitre 1

Introduction

Sommaire

1.1 Protocoles cryptographiques **1**
1.1.1 Primitives cryptographiques 1
1.1.2 Un exemple de protocole 3
1.1.3 Des protocoles pour des applications variées 4
1.1.4 Intérêt de la vérification formelle 5
1.2 Modèles des protocoles **6**
1.3 Propriétés de sécurité **6**
1.3.1 Propriétés de trace et propriétés d'équivalence 6
1.3.2 Secret 7
1.3.3 Authentification 7
1.4 Vérification des protocoles dans le modèle formel **8**
1.4.1 Propriétés de trace (secret, authentification, ...) 8
1.4.2 Propriétés d'équivalence 12
1.5 Lien entre modèles formel et calculatoire **13**
1.6 Preuve des protocoles dans le modèle calculatoire **14**
1.7 Conclusion **16**

Depuis ma thèse, l'essentiel de mon travail a concerné la vérification automatique de protocoles cryptographiques. Ce chapitre présente une introduction à ce domaine de recherche très actif et y situe mes contributions. Les chapitres suivants se concentreront davantage sur mon propre travail.

1.1 Protocoles cryptographiques

Un protocole est une convention qui détermine les messages échangés entre plusieurs ordinateurs sur un réseau. Un protocole cryptographique utilise des primitives cryptographiques (chiffrement, signature, ..., expliquées un peu plus en détail ci-dessous) afin de garantir que les messages sont échangés de façon sûre, même si le réseau lui-même n'est pas sûr. C'est le cas en particulier d'Internet, sur lequel des ordinateurs inconnus, potentiellement hostiles, peuvent se connecter.

1.1.1 Primitives cryptographiques

Les protocoles cryptographiques utilisent comme briques de base des primitives cryptographiques. Quelques primitives parmi les plus courantes sont les suivantes.

Chiffrement à clé partagée Le chiffrement à clé partagée permet de coder un message à l'aide d'une clé de telle sorte qu'il ne peut être déchiffré en un temps raisonnable que par quelqu'un qui connaît cette clé. On note traditionnellement $\{M\}_k$ le message M chiffré sous la clé k ; la clé k est utilisée pour obtenir M à partir du chiffré $\{M\}_k$.

Un exemple de schéma de chiffrement à clé partagée est DES, qui a été abandonné au profit d'AES, plus sûr. Le chiffrement permet de garantir la confidentialité du message transmis.

Chiffrement à clé publique Le chiffrement à clé publique, ou asymétrique, concept introduit par Diffie et Hellman [DH76] en 1976, se distingue du chiffrement à clé partagée en ce que la clé de déchiffrement n'est pas la même que la clé de chiffrement. La clé de chiffrement est publique, donc n'importe qui peut chiffrer un message. Par contre, la clé de déchiffrement est secrète. Seul le possesseur de cette clé, destinataire du message, peut déchiffrer. On note $\{M\}_{pk}$ le chiffré de M sous la clé publique pk ; la clé secrète sk permet d'obtenir M à partir de $\{M\}_{pk}$. Le schéma de chiffrement à clé publique le plus connu est RSA, de Rivest, Shamir et Adleman [RSA78].

L'avantage principal du chiffrement à clé publique est que les participants qui s'échangent des messages n'ont pas besoin de partager un secret a priori. Par contre, il est beaucoup plus coûteux que le chiffrement à clé partagée. Pour cette raison, on l'utilise en général pour communiquer une clé partagée qui sera utilisée ensuite pour chiffrer les données elles-mêmes.

Signatures Les signatures reposent également sur la cryptographie asymétrique, mais cette fois la situation est inverse : la clé de signature est secrète. Seul le possesseur de cette clé peut signer. La clé de vérification des signatures est publique, de sorte que n'importe qui peut vérifier qu'une signature est correcte. On note $\{M\}_{sk}$ la signature du message M avec la clé secrète sk.

Le cryptosystème RSA est également à la base d'un schéma de signature. Les signatures garantissent l'authenticité du message signé. (Seul le possesseur de la clé secrète peut signer le message.)

Mise en accord de clés de Diffie-Hellman La mise en accord de clés de Diffie-Hellman [DH76] est fondée sur la propriété suivante de l'exponentiation modulaire : $(g^a)^b = (g^b)^a = g^{ab}$ dans le groupe $\mathbb{Z}_p^*$, où p est un grand nombre premier et g est un générateur de $\mathbb{Z}_p^*$, et sur l'hypothèse qu'il est difficile de calculer g^{ab} à partir de g^a et g^b, sans connaître les nombres aléatoires a et b (hypothèse de Diffie-Hellman calculatoire), ou sur l'hypothèse plus forte qu'il est difficile de distinguer g^a, g^b, g^{ab} de g^a, g^b, g^c sans connaître les nombres aléatoires a, b et c (hypothèse de Diffie-Hellman décisionnelle).

Ces propriétés sont exploitées pour établir une clé partagée entre deux participants A et B d'un protocole : A choisit aléatoirement a et envoie à B g^a ; symétriquement, B choisit aléatoirement b et envoie à A g^b. A peut alors calculer $(g^b)^a$, puisqu'il a a et reçoit g^b, tandis que B calcule $(g^a)^b$. Ces deux valeurs étant égales, elles peuvent être utilisées pour calculer la clé partagée. L'attaquant, par contre, dispose de g^a et g^b, mais pas de a et b donc, par l'hypothèse de Diffie-Hellman calculatoire, il ne peut pas calculer la clé.

Ou exclusif Le ou exclusif fournit un schéma de chiffrement idéal : pour chiffrer un message M, on choisit un nombre aléatoire a, et on calcule le ou exclusif de a et M bit à bit, $a \oplus M$. On retrouve M par $a \oplus (a \oplus M)$. Cependant, ce schéma n'est sûr que si la clé a est utilisée une seule fois, ce qui pose un problème pratique important : il faut transmettre au récepteur du message une clé aussi longue que l'ensemble des messages à chiffrer. On utilise donc rarement le ou exclusif seul comme schéma de chiffrement, mais on l'utilise en combinaison avec d'autres primitives dans des protocoles.

Fonctions de hachage Une fonction de hachage, notée ici h, calcule un nombre, le haché, dont la longueur est du même ordre que celle d'une clé cryptographique (quelques centaines de

bits) à partir de données de longueur quelconque. Le haché sert à vérifier l'intégrité de la donnée hachée : si le haché est le même, la donnée hachée est considérée comme inchangée, donc une fonction de hachage doit satisfaire des propriétés comme la résistance à la pré-image (à partir de h, il est difficile de trouver m tel que $h = \mathrm{h}(m)$) et la résistance aux collisions (il est difficile de trouver m_1 et m_2 distincts tels que $\mathrm{h}(m_1) = \mathrm{h}(m_2)$). Les fonctions de hachage les plus connues sont SHA-1 et MD5. (Des attaques ont été trouvées contre ces deux fonctions [WY05, WYY05].)

Le lecteur qui désire plus de détail sur les primitives cryptographiques pourra consulter les ouvrages d'introduction à la cryptologie [Sch96, Sti05].

1.1.2 Un exemple de protocole

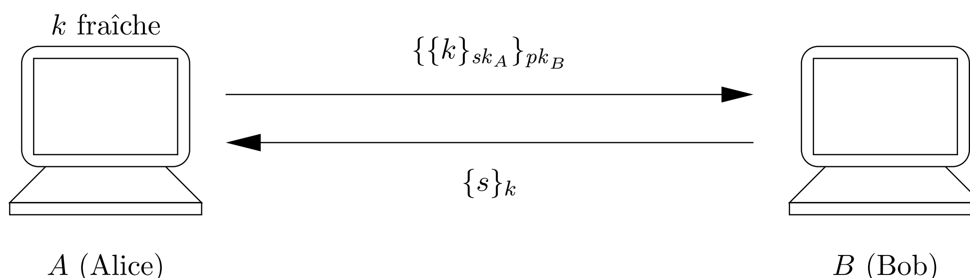

Fig. 1.1 – Un exemple simple de protocole

Nous illustrons la notion de protocole cryptographique sur l'exemple suivant, version simplifiée du protocole de distribution de clés de Denning-Sacco à clé publique [DS81].

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B: & \{\{k\}_{sk_A}\}_{pk_B} \quad k \text{ fraîche} \\ \text{Message 2.} & B \rightarrow A: & \{s\}_k \end{array}$$

Ce protocole est illustré dans la figure 1.1. Dans le protocole, le participant A choisit une clé fraîche k à chaque exécution du protocole. Il signe cette clé avec sa clé secrète sk_A, et chiffre le message obtenu avec la clé publique de son interlocuteur B, et lui envoie le message. Quand il le reçoit, B déchiffre (en utilisant sa clé secrète sk_B), vérifie la signature de A et obtient la clé k. Ayant vérifié cette signature, B est convaincu que la clé a été choisie par A, et le chiffrement sous pk_B garantit que seul B a pu déchiffrer le message, donc k doit être partagée entre A et B. B chiffre alors un secret s sous la clé partagée k. Seul A devrait être capable de déchiffrer le message et d'obtenir le secret s.

En général, dans la littérature, comme dans l'exemple ci-dessus, les protocoles sont décrits informellement en donnant la liste des messages qui doivent être échangés entre les participants. Cependant, il faut faire attention que ces descriptions sont seulement informelles : elles indiquent ce qui se passe en absence d'attaquant. Mais un attaquant peut capturer les messages ou envoyer ces propres messages, donc la source ou la destination d'un message peut ne pas être celle attendue. De plus, ces descriptions laissent implicites les vérifications qui sont effectuées par les participants quand ils reçoivent les messages. Comme l'attaquant peut envoyer des messages différents de ceux attendus, et exploiter la réponse obtenue, ces vérifications sont très importantes : elles déterminent quels messages seront refusés ou acceptés, et peuvent donc protéger ou non contre des attaques. Les modèles formels des protocoles précisent tout cela. Un tel modèle sera présenté à la section 2.1.

Le protocole ci-dessus est sujet à une attaque présentée dans la figure 1.2. Dans cette attaque, A exécute le protocole avec un participant malhonnête C. Ce participant récupère le premier message du protocole $\{\{k\}_{sk_A}\}_{pk_C}$, le déchiffre et le rechiffre avec la clé publique de B. Le message obtenu $\{\{k\}_{sk_A}\}_{pk_B}$ correspond exactement au premier message d'une session entre

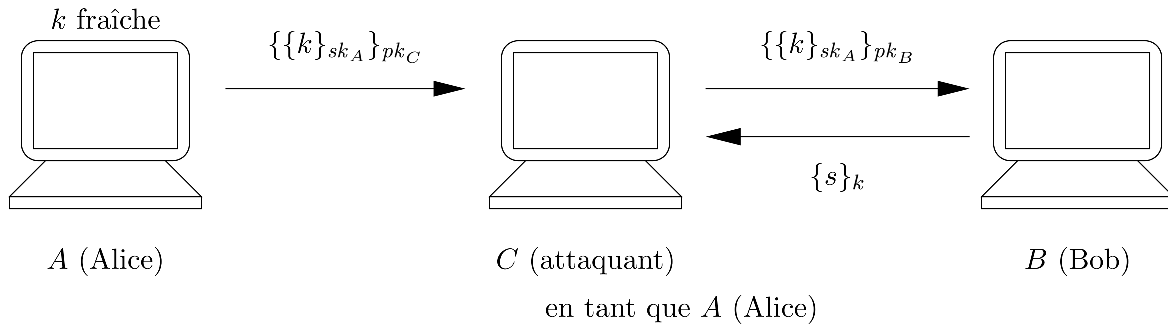

FIG. 1.2 – Une attaque contre ce protocole

A et B. C envoie alors ce message à B en se faisant passer pour A. B répond en envoyant le secret s, destiné à A, chiffré sous k. C, ayant obtenu la clé k par le premier message peut alors déchiffrer ce message et obtenir le secret s.

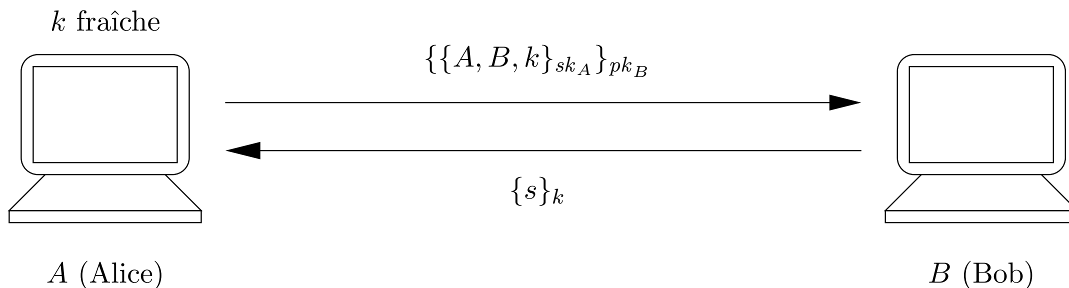

FIG. 1.3 – Le protocole corrigé

Le protocole peut facilement être corrigé. On ajoute les identités de A et B au message signé, ce qui donne le protocole suivant, également présenté dans la figure 1.3 :

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B : & \{\{A, B, k\}_{sk_A}\}_{pk_B} \quad k \text{ fraîche} \\ \text{Message 2.} & B \rightarrow A : & \{s\}_k \end{array}$$

Quand il reçoit le premier message, B vérifie que les identités de A et B sont correctes (en particulier, que sa propre identité apparaît en deuxième position). Après cette modification, dans une session entre A et C, l'attaquant C reçoit $\{\{A, C, k\}_{sk_A}\}_{pk_C}$. Il ne peut alors plus transformer ce message en $\{\{A, B, k\}_{sk_A}\}_{pk_B}$, parce qu'il ne peut pas transformer la signature qui contient C en une signature qui contient B à la place. L'attaque précédente est donc impossible. Cependant, cela ne prouve pas que le protocole est correct : il peut y avoir d'autres attaques. Nous poursuivrons l'étude de ce petit exemple dans le chapitre 2 et verrons comment il peut être vérifié automatiquement.

1.1.3 Des protocoles pour des applications variées

Longtemps, l'usage de la cryptographie a été essentiellement militaire, pour pouvoir communiquer des informations secrètes sans que l'ennemi puisse les obtenir. De nos jours, la cryptographie est beaucoup utilisée dans le domaine civil, en particulier sur Internet. Un type de protocole très fréquent est l'échange de clés : deux participants utilisent un protocole pour convenir d'une clé partagée, puis utilisent cette clé pour transmettre des données de façon sûre. Les protocoles suivants fonctionnent de cette façon :

– SSH (*Secure SHell*) [Ylö06] est utilisé pour des connexions sûres vers des machines distantes et pour des transferts de fichiers.

- TLS (*Transport Layer Security*) [DR06], qui a succédé à SSL (*Secure Socket Layer*), fournit une couche au-dessus de TCP qui peut fournir des communications sécurisées à n'importe quelle application. C'est en particulier le protocole utilisé pour les URL `https://`. SSL version 1.0 n'a jamais été publié, SSL 2.0 contenait des failles de sécurité, que SSL 3.0 a cherché à corriger [WS96]. TLS 1.0 est très proche de SSL 3.0, et la version actuelle TLS 1.1 contient encore quelques améliorations mineures.
- IKEv2 (*Internet Key Exchange, version 2*) [Kau05] est le protocole d'échange de clés d'IPsec, qui permet de connecter des machines distantes comme si elles faisaient partie d'un réseau privé, formant ainsi un réseau privé virtuel (VPN, *Virtual Private Network*). (La version 2 corrige des faiblesses de la première version, en particulier concernant la résistance aux attaques par déni de service. Une telle attaque consiste à initier de nombreuses connexions sur un serveur de façon à ce qu'il n'ait plus les ressources nécessaires pour répondre aux demandes légitimes de connexion. Une telle attaque est facile si l'attaquant peut initier une connexion avec peu de ressources alors que le serveur doit effectuer des opérations coûteuses.)

Certains protocoles ont des buts plus spécifiques. On peut citer par exemple :

- Les protocoles de vote électronique, tels que [BFP+01, CRS05], cherchent à garantir que chaque votant peut vérifier que son vote est correctement pris en compte, que le secret des votes est préservé, qu'un votant ne peut pas prouver à quelqu'un d'autre comment il a voté (pour éviter l'achat de votes).
- Les protocoles de signature de contrat, par exemple [BWW00] ou [GM99] (qui est erroné [CKS04] et la correction proposée dans [CKS04] est elle-même erronée [MR06]), cherchent en particulier à garantir que chacun des signataires ne peut pas obtenir un contrat signé sans que les autres l'obtiennent aussi.
- Les protocoles de courrier électronique certifié [AGHP02, LMBG05] fournissent l'équivalent électronique de la lettre recommandée.
- Des protocoles sont utilisés pour sécuriser les communications sur les réseaux sans fil WiFi [IEE99] (WEP, *Wired Equivalent Privacy*, qui est sujet à des attaques [BHL06] et a été remplacé par WPA, *WiFi Protected Access* [WFA] et WPA2 [IEE04]).
- Des protocoles cryptographiques sont également utilisés dans le cadre de la téléphonie mobile ou des paiements par carte bancaire.

Des bibliothèques de protocoles, comme [CJ97] et comme le site Internet SPORE (*Security Protocols Open Repository*) à l'adresse `http://www.lsv.ens-cachan.fr/spore/`, fournissent de nombreux exemples de protocoles cryptographiques de la littérature.

1.1.4 Intérêt de la vérification formelle

La conception de protocoles cryptographiques est particulièrement délicate, comme le montre les nombreuses erreurs trouvées dans des protocoles après leur publication. Un exemple extrême est le protocole de Needham-Schroeder à clé publique [NS78], publié en 1978, et contre lequel Lowe a découvert une attaque en 1996 en utilisant le vérificateur de modèles FDR [Low96]. Certains autres exemples de protocoles erronés ont été cités ci-dessus. De plus, les failles de sécurité ne peuvent pas être détectées par le test des protocoles, car elles n'apparaissent qu'en présence d'un attaquant. Des erreurs dans des protocoles peuvent avoir des conséquences graves, comme des pertes financières dans le cas du commerce électronique. Pour toutes ces raisons, il est particulièrement important d'avoir des preuves formelles que les protocoles sont sûrs. C'est pourquoi ce sujet a fait l'objet de recherches très actives.

1.2 Modèles des protocoles

Pour modéliser un protocole cryptographique, on suppose que le réseau est totalement contrôlé par l'attaquant, qui peut écouter les messages transmis, calculer sur ces messages, et envoyer aux participants du protocole n'importe quel message qu'il a réussi à calculer. Un tel attaquant est appelé attaquant actif, par opposition à un attaquant passif qui écoute seulement les messages échangés, sans envoyer ses propres messages.

Deux modèles des protocoles cryptographiques ont été considérés :

- Le modèle formel, dû à Needham et Schroeder [NS78] et à Dolev et Yao [DY83], et souvent appelé modèle de Dolev-Yao, dans lequel les fonctions cryptographiques sont considérées comme des boîtes noires, les messages sont des termes sur ces fonctions cryptographiques et l'attaquant est restreint à calculer à l'aide de ces fonctions. Ce modèle suppose une cryptographie parfaite. Ainsi, pour le chiffrement, on suppose qu'on ne peut déchiffrer que si on a la clé. Plus généralement, on peut ajouter des équations pour modéliser les propriétés des primitives cryptographiques, mais on fait toujours l'hypothèse que les seules égalités vraies sont celles explicitement données par ces équations.
- Le modèle calculatoire (traduction de l'anglais *computational*), développé au début des années 1980 par Goldwasser, Micali, Rivest, Yao, entre autres (voir par exemple [GM84, GMR88, Yao82]) dans lequel les messages sont des suites de bits (0 ou 1) et l'attaquant peut exécuter n'importe quel algorithme modélisé par une machine de Turing probabiliste. La longueur des clés est déterminée par une valeur appelée paramètre de sécurité, et le temps d'exécution de l'attaquant doit être polynomial dans le paramètre de sécurité. Une propriété de sécurité est considérée comme vraie quand la probabilité qu'elle ne soit pas satisfaite est négligeable dans le paramètre de sécurité. (On dit qu'une fonction est négligeable quand elle est inférieure à tout inverse d'un polynôme.) On peut borner explicitement cette probabilité en fonction du temps de calcul de l'attaquant et de la probabilité de casser chaque primitive cryptographique, c'est ce qu'on appelle la sécurité exacte.

Le modèle calculatoire est beaucoup plus réaliste, mais prouver des protocoles dans ce modèle est délicat et, jusqu'à très récemment, ces preuves étaient manuelles. Le modèle formel, au contraire, se prête bien à des preuves automatiques, essentiellement par énumération de tous les messages que peut calculer l'attaquant. Dans les années 1990 et jusqu'à maintenant, la preuve de protocoles dans le modèle formel a été un champ d'application important pour les méthodes formelles de vérification.

1.3 Propriétés de sécurité

Les protocoles peuvent chercher à satisfaire des propriétés de sécurité très variées. Les propriétés les plus courantes peuvent être classées en deux catégories, propriétés de traces et propriétés d'équivalence. Nous définissons ces deux catégories, et mentionnons deux exemples particulièrement importants : secret et authentification.

1.3.1 Propriétés de trace et propriétés d'équivalence

Les propriétés de trace sont des propriétés qui peuvent être définies sur chaque trace d'exécution du protocole. Le protocole satisfait une telle propriété quand elle est vraie pour toute trace dans le modèle formel, sauf pour un ensemble de traces de probabilité négligeable dans le modèle calculatoire. Par exemple, le fait que certains états ne soient pas accessibles est une propriété de trace.

Les propriétés d'équivalence ou d'indistinguabilité signifient que l'attaquant ne peut pas distinguer deux protocoles. Dans le modèle formel, on parle en général d'équivalence de processus

ou d'équivalence observationnelle [AG99, AG98, AF01], alors que dans le modèle calculatoire, on parle plutôt d'indistinguabilité. Ces équivalences permettent de modéliser des propriétés de sécurité subtiles. En particulier, elles permettent de modéliser qu'un protocole satisfait une spécification en requérant que le protocole et sa spécification soient équivalents. Elles fournissent des preuves compositionnelles : si un protocole P est équivalent à P', on peut remplacer P par P' dans un protocole plus complexe. Dans le modèle calculatoire, cette approche est à la base de l'idée de composabilité universelle [Can01]. Par contre, dans le modèle formel, leur preuve est plus difficile à automatiser que les preuves de propriétés de traces : elles ne peuvent pas s'exprimer sur une trace, mais nécessitent des relations entre traces (ou entre processus).

1.3.2 Secret

Le secret, ou confidentialité, signifie que l'attaquant ne peut pas obtenir certaines informations sur les données. Dans le modèle formel, le secret peut être modélisé de deux façons :

- Le plus souvent, le secret signifie que l'attaquant n'est pas capable d'obtenir exactement une certaine donnée. En cas d'ambiguïté, cette notion sera appelée secret syntaxique.
- On utilise parfois une notion plus forte, qu'on appellera secret fort et qui signifie que l'attaquant ne peut pas détecter un changement de la valeur secrète [Aba99, Bla04a]. Autrement dit, l'attaquant n'a aucune information sur la valeur du secret.

La distinction entre secret syntaxique et secret fort peut être illustrée par un exemple simple : considérons une donnée dont l'attaquant connaît la moitié des bits mais pas l'autre moitié. Cette donnée est secrète au sens syntaxique, car l'attaquant ne peut pas la reconstruire, mais pas au sens du secret fort, puisqu'il peut voir si un des bits qu'il connaît change. La notion de secret syntaxique ne peut pas être utilisée pour exprimer le secret de données à choisir parmi des constantes connues. Par exemple, parler du secret syntaxique d'un bit 0 ou 1 ne fait pas de sens, car l'attaquant connaît dès le départ les valeurs 0 et 1. On doit dans ce cas utiliser le secret fort : l'attaquant ne doit pas être capable de distinguer un protocole qui utilise la valeur 0 du même protocole qui utilise la valeur 1. Cortier, Rusinowitch et Zălinescu [CRZ07] ont montré que ces deux notions de secret sont souvent équivalentes, pour des données atomiques (qui ne sont jamais séparées en plusieurs morceaux, comme les nonces, qui sont des nombres aléatoires choisis indépendamment à chaque exécution du protocole, donc utilisés une seule fois[1]) et des primitives cryptographiques probabilistes.

La notion de secret fort est intuitivement plus proche de la notion de secret utilisée dans le modèle calculatoire, qui signifie qu'un attaquant polynomial probabiliste ne peut pas distinguer avec une probabilité non-négligeable le secret d'un nombre aléatoire [AFP06].

Le secret syntaxique est une propriété de trace, tandis que le secret fort et le secret calculatoire sont des propriétés d'équivalence.

1.3.3 Authentification

L'authentification signifie que, si un participant A exécute le protocole apparemment avec un participant B, alors B exécute le protocole apparemment avec A, et réciproquement. En général, on requiert également que A et B partagent les mêmes valeurs des paramètres du protocole.

Dans le modèle formel, ceci est généralement modélisé par des propriétés de correspondance [WL93, Low97], de la forme : si A exécute un certain événement e_1 (par exemple, A termine le protocole avec B), alors B a exécuté un certain événement e_2 (par exemple, B a commencé une session du protocole avec A). Il existe plusieurs variantes de ces propriétés. Par exemple, on peut requérir que chaque exécution de e_1 correspond à une exécution distincte de e_2 (correspondance injective) ou, au contraire, que si e_1 a été exécuté alors e_2 a été exécuté

[1]En anglais, *nonce word* désigne un mot créé pour l'occasion, qui n'est pas destiné à être réutilisé.

au moins une fois (correspondance non-injective). Les événements e_1 et e_2 peuvent également inclure plus ou moins de paramètres suivant la propriété souhaitée. Ces propriétés sont des propriétés de trace.

La modélisation est assez similaire dans le modèle calculatoire, avec la notion de *matching conversations* [BR93b] et les formalisations plus récentes qui utilisent des identifiants de session, comme [BPR00, AFP06], qui requièrent essentiellement que les messages échangés vus par A et par B sont les mêmes, à probabilité négligeable près.

1.4 Vérification des protocoles dans le modèle formel

Nous traitons d'abord de la vérification des propriétés de trace, plus facile, puis des propriétés d'équivalence.

1.4.1 Propriétés de trace (secret, authentification, ...)

La vérification automatique des protocoles dans le modèle formel est certes plus facile que dans le modèle calculatoire, mais elle présente quand même des difficultés importantes. Essentiellement, l'espace d'états à explorer est infini, pour deux raisons : la taille des messages n'est pas bornée en présence d'un attaquant actif ; le nombre de sessions (exécutions) du protocole n'est pas borné. Par contre, on peut facilement borner le nombre de participants au protocole sans oublier d'attaques [CLC04] : pour les protocoles qui ne font pas de tests de différence, un participant honnête est suffisant pour le secret si on autorise un participant à jouer tous les rôles du protocole, deux participants sont suffisants pour l'authentification.

Une solution simple à ce problème consiste à n'explorer qu'une partie finie de l'espace d'états, en limitant arbitrairement à la fois l'ensemble des messages et le nombre de sessions du protocole. On peut alors appliquer des techniques standard de vérification de modèles (*model-checking*), en utilisant des systèmes comme FDR [Low96], Murϕ [MMS97], Maude [DMT98], Brutus [CJM00], Elan [Cir01] ou SATMC (*SAT-based Model-Checker*) [ACG03]. Ceci permet de trouver des attaques contre les protocoles, mais pas de prouver l'absence d'attaques, puisque des attaques peuvent apparaître dans une partie inexplorée de l'espace d'états. (On peut d'ailleurs construire une famille de protocoles telle que le n-ième protocole présente une attaque avec n sessions parallèles [Mil99].)

Si on limite seulement le nombre de sessions, la vérification des protocoles reste décidable : l'insécurité (existence d'une attaque) est NP-complète, moyennant des hypothèses raisonnables sur les primitives utilisées [RT03]. Dans le cas où les primitives cryptographiques ont des relations algébriques, la situation est plus compliquée. Par exemple, le ou exclusif est traité dans le cas d'un nombre borné de sessions dans [CLS03, CKRT03b, CKRT05] et la mise en accord de clés de Diffie-Hellman dans [CKRT03a], toujours avec une complexité NP. Des algorithmes pratiques ont été réalisés pour vérifier les protocoles avec un nombre borné de sessions, par résolution de contraintes, comme [MS01] et CL-AtSe (*Constraint-Logic-based Attack Searcher*) [CV01], ou par des extensions de la vérification de modèles comme OFMC (*On-the-Fly Model-Checker*) [BMV03]. Par contre, pour un nombre non-borné de sessions, le problème est indécidable [DLMS04] pour un modèle raisonnable des protocoles.

De nombreuses méthodes ont été utilisées pour vérifier des protocoles avec un nombre non-borné de sessions, malgré cette indécidabilité, soit en se restreignant à des sous-classes, soit en faisant intervenir l'utilisateur, soit en tolérant la non-terminaison de la vérification, soit avec des systèmes incomplets (qui peuvent répondre "je ne sais pas").

– Des logiques ont été construites pour raisonner sur les protocoles. Les logiques de croyances, comme les logiques BAN, de Burrows, Abadi et Needham [BAN89], GNY, de Gong, Needham et Yahalom [GNY90], et SVO, de Syverson et van Oorschot [SvO94], raisonnent sur ce que croient les participants du protocole. Des procédures de décision pour les

logiques BAN et GNY ont été conçues par Monniaux [Mon99], et la logique BAN a été utilisée dans un outil automatique [KW96].
La logique BAN est un des premiers formalismes introduits pour raisonner sur les protocoles. Cependant, l'inconvénient majeur de ces logiques est qu'elles ne reposent pas directement sur la sémantique opérationnelle du protocole.
Une autre logique, appelée PCL (*Protocol Composition Logic*) [DMP03, DDMP05] permet de prouver qu'une formule est vraie après qu'un participant a exécuté certaines actions, en se fondant sur la sémantique du protocole. Elle permet des raisonnements rigoureux et systématiques sur les protocoles, mais n'a pas été automatisée jusqu'à maintenant. Cremers [Cre08] pointe quelques faiblesses dans cette logique. Cervesato, Meadows et Pavlovic [CMP05] ont conçu une logique qui permet de prouver l'authentification à partir d'hypothèses de secret, à prouver dans un autre formalisme. Ceci permet de séparer les preuves de secret des preuves d'authentification.
- Paulson [Pau98] a utilisé l'assistant de preuves Isabelle pour prouver la sécurité de protocoles cryptographiques. Les preuves dans un assistant de preuves requièrent typiquement beaucoup d'intervention humaine, mais permettent de prouver n'importe quel résultat mathématiquement correct. Une exception à ce point est le système automatique de Cortier, Millen et Rueß dans PVS [CMR01], qui ne traite que le secret. Le prouveur TAPS [Coh03], fondé sur la logique du premier ordre, réussit souvent sans ou avec peu d'intervention humaine.
- Le typage a également été utilisé pour la preuve de protocoles : Abadi [Aba99] prouve le secret fort pour des protocoles qui utilisent le chiffrement à clé partagée. (Nous mentionnons ce travail ici bien qu'il s'agisse d'une propriété d'équivalence, à cause de sa proximité avec les travaux sur la vérification du secret syntaxique par typage.) En collaboration avec Abadi, nous avons conçu un système de type pour prouver le secret pour des protocoles qui utilisent le chiffrement à clé publique [AB03], puis nous l'avons étendu à des primitives variées [AB05a].
Gordon et Jeffrey [GJ03, GJ04, GJ02] ont conçu le système Cryptic, pour vérifier l'authentification par typage. Ils traitent la cryptographie à clé partagée et à clé publique.
Bugliesi et al. [BFM07] vérifient également l'authentification par typage. L'avantage principal de leur système est qu'il est compositionnel : il permet de prouver indépendamment la correction du code de chaque rôle du protocole. Cependant, la forme des messages est restreinte à certains termes étiquetés. Cette approche est comparée avec Cryptic dans [BFM05].
Dans tous ces systèmes de type, les types expriment des informations sur le niveau de sécurité des données, comme par exemple "secret" pour des données secrètes, "public" pour des données publiques. Le typage est mieux adapté à un usage au moins partiellement manuel qu'à une vérification complètement automatique : l'inférence de types est souvent difficile, donc des annotations de types sont nécessaires. La vérification de types peut par contre souvent être automatisée, comme dans le cas de Cryptic. Les types fournissent des contraintes qui peuvent aider les concepteurs de protocoles à garantir les propriétés de sécurité souhaitées, mais les protocoles existants peuvent ne pas satisfaire ces contraintes même s'ils sont corrects.
- Le vérificateur de Heather et Schneider [HS05], fondé sur les fonctions de rang, permet de vérifier les protocoles qui utilisent des clés symétriques ou asymétriques atomiques. (Essentiellement, ce système construit une fonction, la fonction de rang, qui associe une certaine valeur aux termes qui peuvent être connus par l'attaquant et une autre aux termes qui ne le peuvent pas.)
- Les "*strand spaces*" [FHG99] sont un formalisme qui permet de raisonner sur les protocoles. Ce formalisme a été utilisé à la fois pour des preuves manuelles et dans l'outil automatique Athena [SBP01] qui combine vérification de modèles et preuve de théorèmes,

et utilise les *strand spaces* pour réduire l'espace d'états. Scyther [Cre06] utilise une extension de la méthode d'Athena avec des motifs de traces (*trace patterns*) pour analyser simultanément un groupe de traces. Ces outils limitent parfois le nombre de sessions pour garantir la terminaison.

- Broadfoot, Lowe et Roscoe [BLR00, RB99, BR04] ont étendu l'approche par vérification de modèles à un nombre non-borné de sessions. Ils recyclent les nonces, pour en utiliser un nombre fini dans un nombre infini d'exécutions. La technique a d'abord été utilisée pour des exécutions séquentielles, puis généralisée à des exécutions parallèles dans [BR04], mais avec la restriction supplémentaire que les participants doivent être "factorisables". (Essentiellement, une exécution du participant doit pouvoir être séparée en plusieurs exécutions telles que chaque exécution contient une seule valeur fraîche.)
- Une des toutes premières approches pour la vérification de protocoles est l'*Interrogator*, de Millen, Clark et Freedman [MCF87, Mil95]. Dans ce système, écrit en Prolog, l'accessibilité d'un état après une suite de messages est représentée par un prédicat, et le programme effectue une recherche en arrière pour déterminer si un état est accessible ou non. Le principal problème de cette approche est la non-terminaison, et il est partiellement résolu en rendant le programme interactif, pour que l'utilisateur guide la recherche.
  L'analyseur de protocoles NRL (*Naval Research Labs*) [Mea96, EMM06] a amélioré cette technique en utilisant le rétrécissement (*narrowing*) dans les systèmes de réécriture. Il n'effectue pas d'abstractions. Il est donc correct et complet, mais peut ne pas terminer.
- On peut obtenir des résultats de décidabilité pour un nombre non-borné de sessions, pour des sous-classes des protocoles. Par exemple, Ramanujan et Suresh [RS03] montrent que le secret est décidable pour une classe de protocoles étiquetés, c'est-à-dire où chaque message est distingué des autres par une constante distincte (étiquette). Le schéma d'étiquetage utilisé interdit les copies aveugles, c'est-à-dire dans lesquelles un message est copié par un participant du protocole qui ne peut pas décomposer ce message. Arapinis et Duflot ont étendu ce résultat [AD07], en interdisant toujours les copies aveugles. Comon et Cortier [CC05] montrent que le secret est décidable pour les protocoles qui ne créent qu'un nombre borné de nouvelles données et qui satisfont certaines restrictions sur les copies de termes, proches de l'absence de copies aveugles. Ces résultats de décidabilité sont en général très restrictifs en pratique.
- Plusieurs méthodes sont fondées sur des abstractions [CC79] : elles surestiment les possibilités d'attaques, la plupart du temps en calculant un sur-ensemble de la connaissance de l'attaquant. Elles permettent d'obtenir des systèmes totalement automatiques mais incomplets.
  - Bolignano [Bol97] a été un précurseur des méthodes d'abstraction pour les protocoles cryptographiques. Il confond des clés, nonces, ... de façon à ce qu'il n'en reste qu'un ensemble fini, puis applique une procédure de décision.
  - Monniaux [Mon03] a introduit une méthode de vérification de protocoles fondée sur une représentation abstraite de l'ensemble des termes que peut connaître l'attaquant par des automates d'arbres. Cette méthode a été étendue par Goubault-Larrecq [GL00]. Genet et Klay [GK00] combinent l'utilisation d'automates d'arbres avec de la réécriture. Cette méthode a conduit à la réalisation du vérificateur TA4SP (*Tree-Automata-based Automatic Approximations for the Analysis of Security Protocols*) [BKV06].
    Le principal inconvénient de cette approche est que les automates d'arbres ne permettent pas de représenter une information relationnelle sur les termes : quand une variable apparaît plusieurs fois dans un message, on oublie qu'elle a la même valeur à toutes ses occurrences dans le message, ce qui limite la précision de l'analyse.
  - L'analyse de flot de contrôle [Bod00, BDNN98] calcule l'ensemble des messages possibles à chaque point du protocole. Elle est également non-relationnelle, et elle confond les nonces créés au même point du protocole dans différentes sessions. Ces approximations

permettent d'obtenir une complexité au pire cubique dans la taille du protocole. Elle a d'abord été définie pour le secret de protocoles à clé partagée, puis étendue à l'authenticité des messages et aux protocoles à clé publique [BBD+05], avec une complexité polynomiale.
– Dans sa thèse, Feret [Fer05] présente une analyse relationnelle par interprétation abstraite sur un métalangage qui permet d'encoder beaucoup de langages de modélisation : pi calcul, ambients, bio-ambients, mais aussi spi calcul, pour analyser des protocoles cryptographiques. Cette analyse est capable de distinguer les différentes sessions d'un protocole.
– La plupart des vérificateurs automatiques de protocoles cherchent à calculer la connaissance de l'attaquant. Au contraire, le vérificateur Hermès [BLP06] détermine des formes de messages, par exemple chiffrement sous certaines clés, qui garantissent la préservation du secret. L'article traite du chiffrement à clé partagée et à clé publique, mais la méthode peut aussi s'appliquer aux signatures et fonctions de hachage.
– Backes et al. [BCM07] prouvent le secret et l'authentification par une analyse fondée sur l'interprétation abstraite. Cette analyse construit un graphe causal (*causal graph*) qui capture la causalité entre les événements du protocole. Les propriétés de sécurité sont prouvées en parcourant ce graphe. Cette analyse termine toujours, mais est incomplète. Elle suppose que les messages sont typés, de sorte que les noms (qui représentent des nombres aléatoires) peuvent être distingués des autres termes.
– Enfin, Weidenbach [Wei99] a introduit une méthode automatique de preuves de protocoles fondée sur la résolution sur des clauses de Horn. C'est la méthode sur laquelle j'ai le plus travaillé : elle est à la base du vérificateur ProVerif et fera l'objet du chapitre 2. Elle est incomplète car elle ignore le nombre de répétitions de chaque action du protocole. La terminaison n'est pas garantie en général, mais elle est garantie sur certaines sous-classes de protocoles, et elle peut être obtenue dans tous les cas grâce à une approximation supplémentaire, qui fait perdre de l'information relationnelle sur les messages, en transformant les clauses de Horn en clauses de la sous-classe décidable $\mathcal{H}_1$ [GL05]. Goubault-Larrecq a montré comment reconstruire un témoin de preuve en Coq à partir d'une preuve du protocole obtenue dans $\mathcal{H}_1$ [GL08], ce qui permet de vérifier que l'outil a correctement prouvé le protocole.
Sans cette approximation supplémentaire, même si elle ne termine pas toujours et est incomplète, cette méthode a l'intérêt de fournir un bon équilibre en pratique : elle termine dans l'immense majorité des cas et est très précise et rapide. Elle permet de traiter des primitives cryptographiques variées, définies par des règles de réécriture ou par certaines équations.
Cette méthode peut être vue comme une généralisation de la méthode de vérification par automates d'arbres. (Les automates d'arbres peuvent être codés dans les clauses de Horn.) En collaboration avec Martín Abadi [AB05a], nous avons montré que cette méthode est équivalente à l'instance la plus précise d'un système de type générique pour les protocoles cryptographiques.

Des plateformes qui regroupent plusieurs techniques de vérification ont également été réalisées :
– CAPSL (*Common Authentication Protocol Specification Language*) [DM00] fournit un langage de description de protocoles, qui est traduit dans un langage intermédiaire, CIL (*CAPSL Intermediate Language*), à base de réécriture de multi-ensembles (ou de façon équivalente de clauses de Horn avec existentiels en logique linéaire) [CDL+99]. Ce langage intermédiaire peut être traduit dans les langages d'entrée de Maude, NRL, Athena et du vérificateur par résolution de contraintes [MS01]. Ce modèle à base de multi-ensembles ou de logique linéaire a été comparé avec les *strand spaces* dans [CDKS00, CDL+05].
– AVISPA (*Automated Validation of Internet Security Protocols and Applications*) [ABB+05] fournit comme CAPSL un langage de description de protocoles de haut-niveau HLPSL

(*High-Level Protocol Specification Language*), qui est traduit dans un langage intermédiaire [AVI03] à base de réécriture de multi-ensembles. Quatre vérificateurs prennent en entrée ce langage intermédiaire : SATMC pour un espace d'états borné, CL-AtSe et OFMC pour un nombre borné de sessions, TA4SP pour un nombre non-borné de sessions. Il existe aussi un traducteur de HLPSL vers ProVerif [GMP05].

1.4.2 Propriétés d'équivalence

La notion d'équivalence est un outil souvent utilisé dans les calculs de processus, même en l'absence de cryptographie [MPW92]. C'est l'outil de preuve introduit avec le spi calcul par Abadi et Gordon [AG99, AG98]. Le spi calcul est limité à quelques primitives cryptographiques, d'abord le chiffrement à clé partagée, puis chiffrement à clé publique, signatures, et fonctions de hachage. Il a été considérablement étendu par le pi calcul appliqué d'Abadi et Fournet [AF01], qui permet de spécifier des primitives cryptographiques très variées, définies par une théorie équationnelle. D'autres calculs de processus et des variantes des notions d'équivalence ont également été utilisés [FGM00, BR05]. Cette approche a été comparée avec le spi calcul dans [GM03] ; un calcul de processus a été comparé avec le modèle de réécriture de multi-ensembles dans [BCLM05].

Les techniques de preuves d'équivalences peuvent se classer comme suit :

- Des techniques manuelles, qui peuvent traiter le cas général [AG99, AG98, AF01, BG02, BDP02, BN05, Cor03]. Une idée fondamentale à la base de la plupart de ces techniques est la suivante. L'équivalence observationnelle (ou congruence barbue) est définie en considérant un attaquant quelconque exécuté en parallèle avec les processus que l'on veut prouver équivalents ; la définition de l'équivalence observationnelle comprend donc une quantification universelle sur tous les processus qui représentent l'attaquant, ce qui rend sa preuve difficile. Pour résoudre ce problème, on introduit une notion de bisimilarité étiquetée, dans laquelle les interactions avec l'attaquant sont remplacées par des transitions étiquetées par l'action effectuée lors de cette interaction (émission ou réception, avec le message émis ou reçu). On montre que la bisimilarité étiquetée est équivalente à l'équivalence observationnelle. Ceci permet de supprimer la quantification universelle sur l'attaquant, mais il reste encore à prouver manuellement la bisimilarité étiquetée ; cette technique peut bien sûr aussi être utilisée comme un premier pas vers l'automatisation de la preuve d'équivalences.
- Si on se réduit à un espace d'états fini, on peut automatiser les preuves d'équivalences, comme dans [DFG00], qui automatise l'approche de [FGM00].
- Dans le cas d'un nombre borné de sessions, on dispose également de procédures automatiques : [DSV03] présente une technique à base de vérification de modèles pour prouver l'équivalence de tests (*may-testing equivalence*) dans le spi calcul, qui est une variante de l'équivalence plus faible que la congruence barbue, car elle ne tient pas compte des points de choix dans les processus.

  Hüttel [Hüt02] fournit une procédure de décision pour la bisimilarité étiquetée (ici, *framed bisimilarity*) dans le spi calcul, pour un nombre borné de sessions, mais la complexité de l'algorithme le rend difficilement applicable en pratique. (Le problème est au moins PSPACE-difficile.)

  Borgström, Briais et Nestmann [BBN04] fournissent une sémantique symbolique du spi calcul (dans laquelle les messages qui viennent de l'attaquant sont représentés par des variables, ce qui évite de devoir considérer chaque message possible indépendamment). Une telle sémantique est un premier pas vers l'automatisation de la preuve d'équivalences. Delaune, Kremer et Ryan [DKR07] définissent une bisimulation symbolique pour le pi calcul appliqué, qui est également un pas vers l'automatisation des preuves d'équivalences.
- Dans le cas d'un nombre non-borné de sessions, l'automatisation du cas général est nettement plus difficile. En collaboration avec Martín Abadi et Cédric Fournet, nous avons

traité le cas particulier suivant en étendant ProVerif : nous montrons l'équivalence observationnelle entre deux processus P et Q qui ne diffèrent que par les termes qu'ils contiennent, en se ramenant à une propriété de trace sur un processus qui représente à la fois P et Q. Ce travail sera expliqué plus en détail dans la section 2.2.6.
- Abadi et Cortier [AC06] fournissent une procédure de décision pour l'équivalence statique, c'est-à-dire essentiellement l'équivalence observationnelle en présence d'un attaquant passif, pour des primitives cryptographiques définies par une grande classe de théories équationnelles.

Bien qu'il soit déjà long, cet inventaire des techniques de vérification de protocoles dans le modèle formel n'est certainement pas exhaustif. Il montre cependant la grande variété des techniques utilisées pour la vérification de protocoles dans le modèle formel, et l'intérêt suscité par ce problème dans la communauté des chercheurs en informatique, et en particulier en vérification formelle.

1.5 Lien entre modèles formel et calculatoire

Récemment, suite au travail fondateur d'Abadi et Rogaway [AR02], on a cherché à faire le lien entre le modèle formel et le modèle calculatoire, et à montrer qu'une preuve obtenue dans le modèle formel était aussi valide dans le modèle calculatoire, ce qui permet d'obtenir des preuves automatiques de protocoles dans le modèle calculatoire.

- Abadi et Rogaway [AR02] ont montré que si deux messages sont indistinguables au sens formel alors ils sont aussi indistinguables au sens calculatoire, si la seule primitive est le chiffrement à clé partagée, moyennant quelques restrictions techniques supplémentaires.
  Abadi et Jürjens [AJ01] ont étendu ce résultat au cas de protocoles en présence d'un attaquant passif, Baudet, Cortier et Kremer [BCK05] à des primitives cryptographiques définies par une grande classe de théories équationnelles (incluant le ou exclusif et le chiffrement symétrique déterministe), Abadi, Baudet et Warinschi [ABW06] à une théorie équationnelle qui comprend chiffrement à clé publique probabiliste et chiffrement à clé partagée probabiliste et déterministe, Bresson, Lakhnech, Mazaré et Warinschi [BLMW07] à l'exponentiation modulaire (utilisée pour la mise en accord de clés de Diffie-Hellman).
- Micciancio et Warinschi [MW04b] montrent que les états et les traces dans le modèle calculatoire correspondent (à probabilité négligeable près) aux états et traces dans le modèle formel, pour le chiffrement à clé publique en présence d'un attaquant actif. Alors l'authentification dans le modèle formel implique l'authentification dans le modèle calculatoire.
  Cortier et Warinschi [CW05] et Janvier, Lakhnech et Mazaré [JLM05] étendent ce travail aux signatures. Cortier et Warinschi [CW05] montrent également que le secret syntaxique dans le modèle formel implique le secret dans le modèle calculatoire pour les nonces.
  Janvier, Lakhnech et Mazaré [JLM06] étendent ces résultats aux fonctions de hachage dans le modèle de l'oracle aléatoire, pour les propriétés de trace et pour le secret des nonces pourvu qu'ils ne soient pas sous des applications de fonctions de hachage. (Le modèle de l'oracle aléatoire [BR93a, CGH04] est un modèle idéalisé où on suppose l'existence de fonctions aléatoires, qui donnent un résultat aléatoire pour toute nouvelle valeur de leur argument, mais redonnent le même résultat pour la même valeur de l'argument.) Cortier, Kremer, Küsters et Warinschi [CKKW06] suppriment la restriction sur le hachage des nonces grâce à une notion modifiée de secret formel.
  Un outil [CHW06] a été développé en se fondant sur [CW05] pour obtenir des preuves calculatoires en utilisant le vérificateur formel AVISPA, pour des protocoles qui utilisent le chiffrement à clé publique et les signatures.
- Micciancio et Warinschi [MW04a] étudient également la propriété réciproque dans le cas passif : ils montrent que, si un attaquant ne peut pas distinguer deux systèmes dans le modèle calculatoire, alors il ne peut pas les distinguer dans le modèle formel, quand la seule

primitive est le chiffrement à clé partagée authentifié en présence d'un attaquant passif. (Dans sa définition de base, le chiffrement à clé partagée ne garantit pas l'authenticité du message : si l'attaquant choisit deux messages M_1 et M_2 et récupère un chiffré $\{M_b\}_k$, où b vaut 0 ou 1, il ne doit pas être capable de déterminer b, mais cela n'interdit pas qu'un attaquant puisse créer un chiffré, par exemple par modification d'un chiffré qu'il a reçu. Quand l'attaquant a une probabilité négligeable de créer un chiffré correct sans avoir la clé, on dit que le chiffrement est authentifié. On peut créer un schéma de chiffrement authentifié en adjoignant au chiffré un code d'authentification de message [BN00].)

- Herzog [Her03] montre que la sécurité dans le modèle formel implique la sécurité dans le modèle calculatoire, pour le chiffrement à clé publique, pourvu que le chiffrement soit "*plaintext-aware*", c'est-à-dire qu'être capable de créer un chiffré implique de connaître le message clair correspondant. Cette propriété est réalisable dans le modèle de l'oracle aléatoire.

  Herzog, Liskov et Micali [HLM03] donnent une définition modifiée de "*plaintext-aware*", qui est suffisante pour le résultat ci-dessus et implémentable en utilisant un tiers de confiance.
- Backes, Pfitzmann et Waidner [BPW03a, BPW03b, BP04] ont développé une bibliothèque cryptographique abstraite qui inclut chiffrement à clé partagée authentifié et à clé publique, code d'authentification de messages, signatures et nonces, et ils ont montré sa correction par rapport aux primitives calculatoires, sous des attaques actives arbitraires. Backes et Pfitzmann [BP05a] lient les notions de secret calculatoire et formel dans le cadre de cette bibliothèque. Ce travail lie le modèle calculatoire à une version non-standard du modèle de Dolev-Yao, dans laquelle la longueur des messages est présente. Il a été utilisé pour une preuve du protocole de Needham-Schroeder corrigé par Lowe [Low96] vérifiée dans un assistant de preuve [SBB+06]. Ce travail est fondé sur la notion de simulation entre machines, qui a été comparée avec l'approche par correspondances entre traces de Micciancio, Warinschi et al. dans [BDK07].
- Canetti et Herzog [CH06] montrent comment une analyse symbolique du style de celle faite dans le modèle de Dolev-Yao peut être utilisée pour prouver des propriétés de sécurité des protocoles dans le cadre de la composabilité universelle [Can01] pour une classe restreinte de protocoles qui utilisent seulement le chiffrement à clé publique. Ils utilisent alors mon vérificateur automatique de protocoles dans le modèle de Dolev-Yao, ProVerif [Bla04a], pour vérifier les protocoles dans ce cadre.

Cette approche a eu des succès importants, mais a aussi des limitations : des hypothèses supplémentaires sont nécessaires, car les deux modèles ne correspondent pas exactement. Les primitives cryptographiques doivent satisfaire des propriétés de sécurité fortes pour qu'elles correspondent à des primitives formelles. De plus, les protocoles doivent satisfaire certaines restrictions. Ainsi, pour le chiffrement à clé partagée, il ne doit pas exister de cycles de clés (dans lesquels une clé est chiffrée directement ou indirectement par elle-même, comme dans $\{k\}_k$ ou $\{k\}_{k'}, \{k'\}_k$) ou bien une définition spécifique de la sécurité du chiffrement est nécessaire [ABHS05, BPS07]. (L'existence de cycles de clés pour un nombre borné de sessions est un problème co-NP-complet [CZ06].) Ces limitations ont conduit à l'idée d'automatiser directement les preuves dans le modèle calculatoire.

1.6 Preuve des protocoles dans le modèle calculatoire

Dans le modèle calculatoire, les preuves de protocoles sont essentiellement manuelles. Le principe de base de ces preuves consiste à faire des preuves par réduction : on montre que s'il existe une attaque contre le protocole, alors on peut construire une attaque contre une hypothèse de sécurité sur une des primitives cryptographiques utilisées. Les propriétés de sécurité des primitives sont elles-mêmes prouvées par réduction : une attaque contre une primitive implique

de résoudre un problème mathématique connu et considéré comme difficile (factorisation de grands entiers, logarithme discret, ...).

Cependant, les preuves par réduction deviennent vite complexes quand plusieurs hypothèses de sécurité sur les primitives sont nécessaires pour prouver le protocole. Shoup [Sho01, Sho02, Sho04] et Bellare et Rogaway [BR06] ont proposé d'organiser ces preuves en suites de jeux :

- Le premier jeu correspond au protocole à prouver, en interaction avec un attaquant. Le but de la preuve est de montrer que la probabilité de casser une certaine propriété de sécurité de ce protocole est négligeable.
- Les jeux suivants sont obtenus les uns après les autres, par transformations successives, de sorte que la différence de probabilité entre deux jeux consécutifs est négligeable. La preuve que cette probabilité est négligeable repose soit sur une unique hypothèse de sécurité sur une primitive, soit sur des transformations de jeux qui sont sûres inconditionnellement. (Par exemple, deux nombres aléatoires choisis uniformément dans un ensemble de chaînes de bits de longueur le paramètre de sécurité ont une probabilité négligeable d'être égaux.)
- Le dernier jeu est tel que la propriété de sécurité souhaitée est évidemment vraie, de par la forme même du jeu. (Par exemple, si on souhaite montrer qu'un certain événement du protocole a une probabilité négligeable d'être exécuté, le dernier jeu ne contiendra simplement pas cet événement.)

On peut alors en déduire que la probabilité de casser la propriété de sécurité dans le jeu initial est négligeable. On peut évaluer cette probabilité en faisant la somme des différences de probabilité entre deux jeux consécutifs et de la probabilité de casser la propriété de sécurité dans le jeu final (qui est souvent nulle).

Des cadres de travail ont été développés afin de systématiser ces preuves, mais sans pour autant les automatiser :

- Lincoln et al. [LMMS98, LMMS99, MMS03, RMST04, MRST06] ont développé un calcul de processus polynomial probabiliste pour l'analyse des protocoles cryptographiques. Ils définissent une notion d'équivalence observationnelle pour ce calcul, qui correspond à l'indistinguabilité entre les jeux. Ils dérivent également des propriétés de compositionalité et un système de preuve équationnel pour ce calcul.
- Datta et al. [DDM+05, DDMW06] ont conçu une logique correcte vis-à-vis du modèle calculatoire. Cette logique est une adaptation au modèle calculatoire de la *Protocol Composition Logic* conçue pour le modèle formel [DMP03, DDMP05].
- Corin et den Hartog [CdH06] utilisent une logique dans le style de la logique de Hoare pour formaliser les preuves par jeux.
- Canetti et al. [CCK+06] utilisent le cadre des *time-bounded task-PIOAs* (*Probabilistic Input/Output Automata*) pour prouver des protocoles cryptographiques dans le modèle calculatoire. Ce cadre leur permet de combiner des comportements probabilistes et non-déterministes.

Barthe, Cerderquist et Tarento [BCT04, Tar05] ont formalisé le modèle générique et le modèle de l'oracle aléatoire dans l'assistant de preuve Coq, et ont prouvé des schémas de signature dans ce cadre. Par rapport aux approches précédentes, cette technique a l'avantage de fournir des preuves vérifiées mécaniquement, mais elle nécessite beaucoup d'intervention humaine.

Les travaux suivants cherchent eux à automatiser les preuves dans le modèle calculatoire :

- Halevi [Hal05] explique que réaliser un prouveur automatique fondé sur les suites de jeux serait utile et suggère des idées dans cette direction, mais il n'a pas encore réalisé un tel prouveur.
- Laud [Lau05] a conçu un système de type pour prouver les protocoles cryptographiques dans le modèle calculatoire. Ce système de type traite le chiffrement à clé partagée et à clé publique, avec un nombre non-borné de sessions. Il repose sur la bibliothèque cryptographique de Backes-Pfitzmann-Waidner [BPW03a]. Un algorithme d'inférence de type est donné dans [BL06].

- Laud [Lau03] a conçu une analyse automatique pour prouver le secret pour des programmes dans un petit langage avec des boucles *while*, qui utilisent le chiffrement à clé partagée probabiliste mais pas de déchiffrement, avec des attaquants passifs. Avec Vene [LV05], il a conçu un système de types pour le même objectif. Smith et Alpízar [SA06] traitent les programmes avec déchiffrement. Courant, Ene et Lakhnech [CEL07] ont conçu un système de type pour les programmes qui utilisent le chiffrement à clé partagée déterministe, ce qui introduit de nouvelles difficultés.
- Laud [Lau04] a réalisé un système qui prouve le secret pour des protocoles utilisant le chiffrement à clé partagée probabiliste en présence d'attaquants actifs, mais pour seulement une session du protocole. Bien que ce système soit assez limité, il produit des preuves par jeux.
  En collaboration avec David Pointcheval, nous avons considérablement étendu cette approche, en traitant des primitives cryptographiques variées et un nombre polynomial de sessions, en présence d'un attaquant actif. Ce travail a conduit à la réalisation du vérificateur automatique CryptoVerif, et fera l'objet du chapitre 3.
  Récemment, Tšahhirov et Laud [TL07] ont développé un outil de vérification de protocoles par suites de jeux. Cet outil utilise une représentation des jeux par des graphes de dépendances, et il est pour l'instant moins développé que CryptoVerif : il ne traite que le chiffrement à clé publique et prouve des propriétés de secret ; il ne fournit pas de borne explicite sur la probabilité de succès d'une attaque.

1.7 Conclusion

Ce chapitre a présenté une introduction au domaine de protocoles cryptographiques et de leur vérification. Il a montré combien ce domaine de recherche a été et reste encore très actif. Les chapitres suivants présentent mes propres recherches, qui ont conduit à la réalisation de deux vérificateurs automatiques de protocoles, ProVerif et CryptoVerif.

Chapitre 2

Vérification des protocoles dans le modèle formel

Sommaire

2.1 Représentation formelle des protocoles cryptographiques 17
2.1.1 Historique 17
2.1.2 Un langage de représentation des protocoles 18
2.1.3 Un exemple de protocole dans ce langage 20
2.1.4 Sémantique 21
2.1.5 Extension aux théories équationnelles 21
2.2 Les clauses de Horn 22
2.2.1 Définition du secret 23
2.2.2 Du pi calcul vers les clauses de Horn 23
2.2.3 Résolution sur les clauses 28
2.2.4 Vérification des propriétés de correspondances 31
2.2.5 Scénarios à plusieurs phases 33
2.2.6 Preuves d'équivalences 34
2.3 Résultats 37
2.4 Conclusion 38

Dans ce chapitre, nous résumons les résultats de recherche qui ont permis la réalisation du vérificateur automatique de protocoles ProVerif. Ce vérificateur est fondé sur le modèle formel, de Dolev-Yao. Il peut traiter un nombre non-borné de sessions et des primitives cryptographiques variées, définies par des règles de réécriture ou par des équations. Nous présentons tout d'abord la représentation formelle des protocoles utilisée en entrée par ProVerif, puis nous expliquons la technique de vérification utilisée, d'abord pour le secret, puis pour les propriétés de correspondances et les équivalences observationnelles. Au passage, nous résumons des résultats de terminaison de cette méthode et des résultats de comparaison avec d'autres travaux (typage, modèle de réécriture de multi-ensembles). Enfin, nous donnons quelques exemples d'études de protocoles effectuées avec l'aide de ProVerif.

2.1 Représentation formelle des protocoles cryptographiques

2.1.1 Historique

Afin de vérifier formellement des protocoles, il est tout d'abord indispensable de disposer d'un modèle formel de ces protocoles, avec une sémantique opérationnelle claire. De nombreux modèles ont été proposés dans littérature, comme des calculs de processus [AG99, FGM00, AF01, BR05], les *strand spaces* [FHG99], et la réécriture de multi-ensembles [CDL+99].

$M, N ::=$	termes
x, y, z	variable
a, b, c, k, s	nom
$f(M_1, \ldots, M_n)$	application de constructeur
$P, Q ::=$	processus
$\overline{M}\langle N \rangle.P$	émission
$M(x).P$	réception
0	processus nul
$P \mid Q$	composition parallèle
$!P$	réplication
$(\nu a)P$	restriction
let $x = g(M_1, \ldots, M_n)$ in P else Q	application de destructeur
let $x = M$ in P	définition locale
if $M = N$ then P else Q	conditionnelle

Fig. 2.1 – Syntaxe du calcul de processus

Ces modèles se distinguent les uns des autres par leur expressivité :

- La plupart des modèles supposent que tous les messages sont envoyés sur un réseau public, où l'attaquant peut les manipuler. Ceci est suffisant pour traiter la plupart des protocoles de base. Les calculs de processus qui étendent le pi calcul, comme le spi calcul [AG99] et le pi calcul appliqué [AF01] sont plus expressifs, car ils considèrent également des canaux privés, qui peuvent servir à coder des communications internes ou des cellules mémoire, par exemple.
- Les primitives cryptographiques supportées peuvent être plus ou moins générales. Beaucoup de modèles, dont le spi calcul [AG99] traitent quelques primitives fixées a priori. Le pi calcul appliqué [AF01] est beaucoup plus général : il permet de modéliser les propriétés des primitives cryptographiques par des théories équationnelles quelconques.

Le modèle que nous présentons en détail ci-dessous est intermédiaire entre le spi calcul et le pi calcul appliqué : il permet de définir des primitives cryptographiques par des règles de réécriture, ce qui est plus restrictif que des théories équationnelles générales. Nous résumerons brièvement son extension à une large classe de théories équationnelles dans la section 2.1.5.

2.1.2 Un langage de représentation des protocoles

La figure 2.1 donne la syntaxe de notre calcul de processus. Ce calcul distingue les termes, qui représentent les messages du protocole, des processus, qui représentent les programmes qui manipulent ces termes. Les noms a, b, c, k, s représentent des données atomiques (nonces, clés, ...), alors que les variables x, y, z peuvent être substituées par n'importe quel message. Le calcul distingue deux catégories de symboles de fonction pour représenter les primitives cryptographiques : les constructeurs, souvent notés f, et les destructeurs, souvent notés g.

Les constructeurs construisent de nouveaux termes. Donc les termes sont les variables, les noms et les applications de constructeurs $f(M_1, \ldots, M_n)$. Au contraire, les destructeurs n'apparaissent pas dans les termes, mais manipulent les termes dans les processus. Les destructeurs sont des fonctions partielles que les processus peuvent appliquer. Le processus let $x = g(M_1, \ldots, M_n)$ in P else Q essaie d'évaluer $g(M_1, \ldots, M_n)$; si cela réussit, x est lié au résultat obtenu et P est exécuté ; sinon, Q est exécuté. Plus précisément, la sémantique d'un destructeur g d'arité n est définie par un ensemble fini $\mathrm{def}(g)$ de règles de réécriture de la forme $g(M_1, \ldots, M_n) \to M$ où $M_1, \ldots, M_n, M$ sont des termes sans noms, et les variables de M apparaissent dans $M_1, \ldots, M_n$. Nous étendons naturellement ces règles par $g(M'_1, \ldots, M'_n) \to M'$

N-uplets :
Constructeur : n-uplet $n\text{tuple}(x_1, \ldots, x_n)$
Destructeurs : projections $i\text{th}_n(n\text{tuple}(x_1, \ldots, x_n)) \rightarrow x_i$
Chiffrement à clé partagée :
Constructeur : chiffrement de x sous la clé y, $\text{sencrypt}(x, y)$
Destructeur : déchiffrement $\text{sdecrypt}(\text{sencrypt}(x, y), y) \rightarrow x$
Chiffrement à clé partagée probabiliste :
Constructeur : chiffrement de x sous la clé y avec l'aléa r, $\text{sencrypt}_\text{p}(x, y, r)$
Destructeur : déchiffrement $\text{sdecrypt}_\text{p}(\text{sencrypt}_\text{p}(x, y, r), y) \rightarrow x$
Chiffrement à clé publique probabiliste :
Constructeurs : chiffrement de x sous la clé y avec l'aléa r, $\text{pencrypt}_\text{p}(x, y, r)$
génération de la clé publique à partir de la clé secrète y, $\text{pk}(y)$
Destructeur : déchiffrement $\text{pdecrypt}_\text{p}(\text{pencrypt}_\text{p}(x, \text{pk}(y), r), y) \rightarrow x$
Signatures :
Constructeurs : signature du x avec la clé secrète y, $\text{sign}(x, y)$
génération de la clé publique à partir de la clé secrète y, $\text{pk}(y)$
Destructeurs : vérification de signature $\text{checksignature}(\text{sign}(x, y), \text{pk}(y)) \rightarrow x$
message sans signature $\text{getmessage}(\text{sign}(x, y)) \rightarrow x$
Signatures qui ne révèlent pas le message :
Constructeurs : signature de x avec la clé secrète y, $\text{nmrsign}(x, y)$
génération de la clé publique à partir de la clé secrète x, $\text{pk}(y)$
constante true
Destructeur : vérification $\text{nmrchecksign}(\text{nmrsign}(x, y), \text{pk}(y), x) \rightarrow \text{true}$
Fonctions de hachage :
Constructeur : fonction de hachage $\text{h}(x)$

FIG. 2.2 – Constructeurs et destructeurs

si et seulement s'il existe une substitution σ et une règle de réécriture $g(M_1, \ldots, M_n) \rightarrow M$ dans $\text{def}(g)$ telles que $M'_i = \sigma M_i$ pour tout $i \in \{1, \ldots, n\}$ et $M' = \sigma M$. En utilisant les constructeurs et les destructeurs, nous pouvons représenter les structures de données et les primitives cryptographiques comme résumé dans la figure 2.2. Par exemple, la règle de réécriture $\text{sdecrypt}(\text{sencrypt}(x, y), y) \rightarrow x$ signifie que, quand on déchiffre un chiffré $\text{sencrypt}(M, N)$ avec la bonne clé N, on obtient le clair M. L'application du destructeur $\text{sdecrypt}(M, N)$ échoue dans le message M à déchiffrer n'est pas un chiffré ou est un chiffré avec une clé différente de N. (Pour le chiffrement à clé publique, nous présentons seulement un chiffrement probabiliste car, dans le modèle calculatoire, un chiffrement à clé publique sûr est forcément probabiliste. Nous avons choisi de présenter les signatures déterministes ; nous pourrions facilement modéliser des signatures probabilistes en ajoutant un troisième argument r contenant l'aléa, comme pour le chiffrement. L'aléa r doit être choisi par une restriction (νr) qui crée un nom frais r, représentant un nombre aléatoire frais.)

Les autres constructions de la syntaxe de la figure 2.1 sont standard ; la plupart viennent du pi calcul. Le processus $M(x).P$ reçoit un message sur le canal M, le stocke dans la variable x, et exécute P. Le processus $\overline{M}\langle N \rangle.P$ envoie le message N sur le canal M, puis exécute P. (Nous autorisons les communications sur des canaux qui peuvent être n'importe quel terme.) Le processus nul 0 ne fait rien. Le processus $P \mid Q$ est la composition parallèle de P et Q, utilisée par exemple quand P et Q représentent des programmes exécutés par différents participants du protocole. La réplication $!P$ représente un nombre non-borné de copies de P en parallèle ; elle permet de représenter un nombre non-borné de sessions du protocole. La restriction $(\nu a)P$ crée un nouveau nom a puis exécute P. La conditionnelle $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ exécute P si M et N se réduisent vers le même terme à l'exécution ; sinon elle exécute Q. Cette conditionnelle peut

être définie comme du sucre syntaxique pour $\mathsf{let}\ x = \mathrm{equal}(M, N)\ \mathsf{in}\ P\ \mathsf{else}\ Q$, où le destructeur equal est défini par $\mathrm{equal}(y, y) \rightarrow y$ et x n'apparaît pas dans P. Nous définissons $\mathsf{let}\ x = M\ \mathsf{in}\ P$ comme du sucre syntaxique pour $P\{M/x\}$, où $\{M/x\}$ est la substitution qui à x associe M. Une branche else peut être omise quand elle contient seulement 0.

Le nom a est lié dans le processus $(\nu a)P$. La variable x est liée dans P dans les processus $M(x).P$ et $\mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q$. Nous notons $\mathrm{fn}(P)$ et $\mathrm{fv}(P)$ les ensembles de noms et de variables libres dans P, respectivement. Un processus est clos quand il n'a pas de variable libre.

2.1.3 Un exemple de protocole dans ce langage

Nous illustrons ce langage en codant l'exemple de protocole de la section 1.1.2 par le processus P_0 suivant :

$$\begin{aligned}
&P_0 = (\nu sk_A)(\nu sk_B)\mathsf{let}\ pk_A = \mathrm{pk}(sk_A)\ \mathsf{in}\ \mathsf{let}\ pk_B = \mathrm{pk}(sk_B)\ \mathsf{in}\ \overline{c}\langle pk_A\rangle.\overline{c}\langle pk_B\rangle.\\
&\qquad (P_A(pk_A, sk_A) \mid P_B(pk_B, sk_B, pk_A))\\
&P_A(pk_A, sk_A) = !\ c(x_pk_B).(\nu k)(\nu r)\overline{c}\langle \mathrm{pencrypt_p}(\mathrm{sign}(k, sk_A), x_pk_B, r)\rangle.\\
&\qquad c(x).\mathsf{let}\ z = \mathrm{sdecrypt}(x, k)\ \mathsf{in}\ 0\\
&P_B(pk_B, sk_B, pk_A) = !\ c(y).\mathsf{let}\ y' = \mathrm{pdecrypt_p}(y, sk_B)\ \mathsf{in}\\
&\qquad \mathsf{let}\ x_k = \mathrm{checksignature}(y', pk_A)\ \mathsf{in}\ \overline{c}\langle \mathrm{sencrypt}(\mathrm{s}, x_k)\rangle
\end{aligned}$$

Un tel processus peut être donné en entrée à l'outil ProVerif (dans une syntaxe ASCII). Ce processus crée tout d'abord les clés secrètes sk_A et sk_B, calcule les clés publiques correspondantes pk_A et pk_B, et envoie ces clés sur le canal public c, de sorte que l'attaquant a ces clés publiques. Ensuite, il exécute les processus P_A et P_B en parallèle. Ces processus correspondent respectivement aux rôles de A et B dans le protocole. Ils commencent tous les deux par une réplication, ce qui permet de modéliser un nombre non-borné de sessions du protocole.

Le processus P_A reçoit tout d'abord sur le canal public c la clé x_pk_B, qui est la clé publique de l'interlocuteur de A dans le protocole. Ce message ne fait pas à proprement parler partie du protocole ; il permet à l'attaquant de choisir avec qui A va exécuter une session. Lors d'une session normale du protocole cette clé est pk_B, mais l'attaquant peut aussi choisir une autre clé, par exemple une de ses propres clés. Ensuite P_A exécute le rôle de A : il crée une nouvelle clé k, la signe avec sa clé secrète sk_A, puis chiffre le tout sous x_pk_B avec l'aléa r, et envoie le message obtenu sur le canal c. P_A attend alors le deuxième message du protocole sur le canal c, il le stocke dans x et le déchiffre. Si le déchiffrement réussit, le résultat (normalement le secret s) est stocké dans z.

Le processus P_B reçoit le premier message du protocole sur le canal c, le stocke dans y, le déchiffre avec sk_B et vérifie la signature avec pk_A. (La signature est vérifiée avec la clé pk_A de A et non avec une clé arbitraire choisie par l'attaquant car B n'envoie le deuxième message $\{\mathrm{s}\}_k$ que si son interlocuteur est le participant honnête A.) Si ces vérifications réussissent, B pense que x_k est une clé partagée entre A et B, et il envoie le secret s chiffré sous x_k. Si le protocole est correct, s doit rester secret.[1] En utilisant la technique décrite ci-dessous, ProVerif montre que ce n'est pas le cas pour ce protocole, mais que c'est le cas pour sa version corrigée.

Dans le modèle ci-dessus, nous avons supposé pour plus de simplicité que A et B jouent chacun un seul rôle dans le protocole. On pourrait facilement écrire un modèle plus général où ils jouent les deux rôles, ou même fournir une interface à l'attaquant qui lui permet de créer dynamiquement de nouveaux participants du protocole.

[1] Le secret s est un nom libre de P_0 ; la définition du secret (définition 2.3 ci-dessous) ne permet pas de considérer le secret de noms liés. Bien sûr, le nom s ne fait pas partie de la connaissance initiale de l'attaquant, qui contient uniquement le nom libre c.

$$E, \mathcal{P} \cup \{\, 0 \,\} \rightarrow E, \mathcal{P} \qquad \text{(Red Nil)}$$
$$E, \mathcal{P} \cup \{\, !P \,\} \rightarrow E, \mathcal{P} \cup \{\, P, !P \,\} \qquad \text{(Red Repl)}$$
$$E, \mathcal{P} \cup \{\, P \mid Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P, Q \,\} \qquad \text{(Red Par)}$$
$$E, \mathcal{P} \cup \{\, (\nu a)P \,\} \rightarrow E \cup \{a'\}, \mathcal{P} \cup \{\, P\{a'/a\} \,\} \qquad \text{(Red Res)}$$
où $a' \notin E$.
$$E, \mathcal{P} \cup \{\, \overline{N}\langle M\rangle.Q, N(x).P \,\} \rightarrow E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\} \qquad \text{(Red I/O)}$$
$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P\{M'/x\} \,\} \qquad \text{(Red Destr 1)}$$
si $g(M_1, \ldots, M_n) \rightarrow M'$
$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} \qquad \text{(Red Destr 2)}$$
s'il n'existe aucun M' tel que $g(M_1, \ldots, M_n) \rightarrow M'$

Fig. 2.3 – Sémantique formelle

2.1.4 Sémantique

La sémantique de ce langage est définie formellement dans la figure 2.3. Le plus souvent, la sémantique de tels calculs de processus est définie en combinant une équivalence structurelle et une relation de réduction, l'équivalence structurelle permettant de transformer les processus pour application les réductions, comme dans [AG99, AF01]. La sémantique de notre calcul peut aussi être définie de cette façon [AB05a, BAF08]. Ici, nous avons préféré la définir avec seulement une relation de réduction, sur des configurations de la forme $E, \mathcal{P}$, où l'environnement E est un ensemble fini de noms et $\mathcal{P}$ est un multi-ensemble fini de processus. L'environnement E doit contenir tous les noms libres de processus de $\mathcal{P}$. La configuration $\{a_1, \ldots, a_n\}, \{P_1, \ldots, P_n\}$ correspond intuitivement au processus $(\nu a_1) \ldots (\nu a_n)(P_1 \mid \ldots \mid P_n)$. Les règles de réduction de la figure 2.3 exécutent les processus comme suit : (Red Nil) supprime un processus nul ; (Red Repl) crée une nouvelle copie de P ; (Red Par) décompose une composition parallèle ; (Red Res) crée un nom frais a' (qui n'apparaît pas dans E), l'ajoute à E et substitue a par a' ; (Red I/O) est la règle de communication : elle réduit une émission et une réception sur le même canal N en envoyant le message M ; (Red Destr 1) et (Red Destr 2) exécutent les applications de destructeurs : (Red Destr 1) quand l'application réussit et se réduit en M', (Red Destr 2) quand elle échoue. Dans ces règles, le symbole $\cup$ désigne l'union multi-ensemble sur les multi-ensembles de processus. Les constructions $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ et $\mathsf{let}\ x = M\ \mathsf{in}\ P$ étant définies comme du sucre syntaxique, leur sémantique se déduit de celle des autres instructions.

L'avantage d'une telle sémantique par rapport aux sémantiques plus traditionnelles est qu'elle dirige davantage l'exécution des processus. Ainsi, le renommage n'est effectué que par (Red Res), au lieu de pouvoir être effectué à chaque réduction, et les transformations des réplications et compositions parallèles sont aussi plus dirigées. Ceci simplifie certaines preuves, par exemple pour les propriétés de correspondances (section 2.2.4) ou la reconstruction d'attaques (section 2.2.2).

2.1.5 Extension aux théories équationnelles

En collaboration avec Martín Abadi et Cédric Fournet [BAF05, BAF08], nous avons étendu notre travail pour traiter des primitives cryptographiques définies par des théories équationnelles. L'algèbre de termes formée par les constructeurs est munie d'une théorie équationnelle, définie par un nombre fini d'équations. Par exemple, on peut modéliser un schéma de chiffrement symétrique dans lequel le déchiffrement réussit toujours (mais peut retourner un message qui

n'a pas de sens) par les équations

$$\begin{aligned} \text{sdecrypt}(\text{sencrypt}(x,y),y) &= x \\ \text{sencrypt}(\text{sdecrypt}(x,y),y) &= x \end{aligned} \tag{2.1}$$

où sencrypt et sdecrypt sont des constructeurs. La première équation est standard ; la deuxième permet d'éviter que le test d'égalité $\text{sencrypt}(\text{sdecrypt}(M,N),N) = M$ révèle que M est un chiffré sous N. Ces équations sont vérifiées par les schémas de chiffrement par blocs, qui sont bijectifs. On peut également modéliser la mise en accord de clés de Diffie-Hellman par l'équation [AF01, ABF07]

$$(\text{b}\hat{\ }x)\hat{\ }y = (\text{b}\hat{\ }y)\hat{\ }x \tag{2.2}$$

où b est une constante et ˆ est un constructeur binaire.

L'idée essentielle de notre extension aux équations est de traduire ces équations en un ensemble de règles de réécriture associées aux constructeurs. Par exemple, les équations (2.1) sont traduites dans les règles de réécriture

$$\begin{aligned} &\text{sencrypt}(x,y) \rightarrow \text{sencrypt}(x,y) && \text{sdecrypt}(x,y) \rightarrow \text{sdecrypt}(x,y) \\ &\text{sencrypt}(\text{sdecrypt}(x,y),y) \rightarrow x && \text{sdecrypt}(\text{sencrypt}(x,y),y) \rightarrow x \end{aligned} \tag{2.3}$$

tandis que l'équation (2.2) est traduite en

$$x\hat{\ }y \rightarrow x\hat{\ }y \qquad (\text{b}\hat{\ }x)\hat{\ }y \rightarrow (\text{b}\hat{\ }y)\hat{\ }x \tag{2.4}$$

Intuitivement, ces règles de réécriture permettent, en les appliquant *exactement une fois* pour chaque constructeur, d'obtenir les différentes formes des termes modulo la théorie équationnelle considérée.[2] Les constructeurs sont alors simplement évalués comme les destructeurs dans le calcul ci-dessus. Nous avons défini formellement le fait qu'un ensemble de règles de réécriture modélise une théorie équationnelle ; nous avons conçu des algorithmes qui calculent à partir des équations des règles de réécriture qui modélisent la théorie équationnelle en question [BAF08, section 5]. Nous avons montré que chaque trace dans le calcul à théorie équationnelle correspond à une trace dans le calcul à règles de réécriture, et réciproquement [BAF08, Lemme 1].[3] On est alors ramené au cas plus simple où il n'y a pas d'équations. L'avantage principal de cette méthode est que la résolution, utilisée ci-dessous sur les clauses de Horn, peut continuer à utiliser l'unification syntaxique ordinaire (au lieu de devoir utiliser l'unification modulo la théorie équationnelle), et reste donc efficace.

Cette extension aux équations a cependant des limitations : elle ne permet pas de modéliser les opérations associatives, comme le ou exclusif, car cela nécessiterait une infinité de règles de réécriture. Il serait peut-être possible de traiter ces symboles en utilisant l'unification modulo la théorie équationnelle en question au lieu de l'unification syntaxique, au prix d'une plus grande complexité. Dans le cas d'un nombre borné de sessions, le ou exclusif est traité dans [CLS03, CKRT03b, CKRT05] et une théorie équationnelle plus complète pour l'exponentiation modulaire (utilisée pour la mise en accord de clés de Diffie-Hellman) est traitée dans [CKRT03a]. Un algorithme d'unification pour l'exponentiation modulaire est présenté dans [MN02].

2.2 Les clauses de Horn

Dans cette section, nous décrivons notre méthode de vérification de protocoles, fondée sur les clauses de Horn. L'idée d'utiliser des clauses de Horn pour vérifier les protocoles a été introduite par Weidenbach [Wei99]. Nous avons étendu ses travaux en définissant une traduction

[2]Les règles de réécriture du style $\text{sdecrypt}(x,y) \rightarrow \text{sdecrypt}(x,y)$ sont nécessaires pour que sdecrypt réussisse toujours. Grâce à cette règle, l'évaluation de $\text{sdecrypt}(M,N)$ réussit et laisse ce terme inchangé quand M n'est pas de la forme $\text{sencrypt}(M',N)$.

[3]Plus précisément, les tests d'inégalité dans (Red Destr 2) doivent toujours être faits modulo la théorie équationnelle, même dans le calcul à règles de réécriture.

systématique d'un modèle formel des protocoles en clauses (alors qu'il construisait les clauses manuellement) et en prouvant d'autres propriétés que le secret. Nous commençons par la propriété la plus simple, le secret, puis présentons nos extensions à des propriétés plus complexes (correspondances, équivalences).

2.2.1 Définition du secret

Nous supposons que le protocole est exécuté en présence d'un attaquant qui peut écouter tous les messages, calculer et envoyer les messages qu'il a, suivant le modèle de Needham-Schroeder [NS78] et Dolev-Yao [DY83]. Un tel attaquant est représenté par n'importe quel processus qui a un certain ensemble de noms *Init* dans sa connaissance initiale.

Définition 2.1 Soit *Init* un ensemble fini de noms. Le processus clos Q est un *Init-attaquant* si et seulement si $\mathrm{fn}(Q) \subseteq \mathit{Init}$.

Intuitivement, on dit qu'une trace publie un message M quand ce message est envoyé sur un canal public (dans *Init*). Si l'attaquant obtient le message M, il peut toujours l'envoyer sur un canal de *Init*.

Définition 2.2 Soit M un terme clos. On dit qu'une trace $E_0, \mathcal{P}_0 \to^* E', \mathcal{P}'$ *publie* M si et seulement s'il existe E, $\mathcal{P}$, x, P, Q, et $c \in \mathit{Init}$ tels que cette trace contienne la réduction $E, \mathcal{P} \cup \{\, \overline{c}\langle M\rangle.Q, c(x).P \,\} \to E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\}$.

On dit alors que P_0 préserve le secret de M si M n'est jamais publié, en présence d'un attaquant quelconque.

Définition 2.3 Soit M un terme tel que $\mathrm{fn}(M) \subseteq \mathrm{fn}(P_0)$. Le processus P_0 *préserve le secret de toutes les instances de M à partir de Init* si et seulement si, pour tout *Init*-attaquant Q_0, pour toute substitution σ, il n'existe aucune trace $\mathrm{fn}(P_0) \cup \mathit{Init}, \{P_0, Q_0\} \to^* E', \mathcal{P}'$ qui publie σM.

2.2.2 Du pi calcul vers les clauses de Horn

Le vérificateur de protocoles ProVerif prend en entrée un processus P_0, qui représente le protocole à vérifier, et un ensemble fini de noms *Init*, correspondant à la connaissance initiale de l'attaquant. Nous supposons que les noms liés de P_0 sont deux à deux distincts et distincts des noms libres et des noms de *Init*. ProVerif calcule alors un ensemble de clauses de Horn représentant le protocole et l'attaquant.

Dans ces clauses, les messages sont représentés par des motifs (ou “termes”, mais nous utilisons le mot “motif” pour les distinguer des termes qui apparaissent dans les protocoles). Ces motifs sont notés p, comme *pattern* en anglais, et sont définis par la grammaire suivante :

$p ::=$	motifs
x, y, z, i	variable
$a[p_1, \ldots, p_n]$	nom
$f(p_1, \ldots, p_n)$	application de constructeur

Les motifs diffèrent des termes par la représentation des noms. Nous associons à chaque réplication une variable fraîche i nommée identifiant de session, qui prend une valeur distincte pour chaque copie de processus créée par la réplication. Dans les motifs, les noms créés par des restrictions (νa) sont représentés comme des fonctions $a[p_1, \ldots, p_n]$ des messages reçus avant de créer le nom a, des résultats d'applications de destructeurs calculées avant de créer a et des identifiants de session des réplications au-dessus de (νa) dans l'arbre syntaxique de P_0. Plus précisément, on a une construction $a[p_1, \ldots, p_n]$ pour chaque nom de *Init* et chaque nom libre et chaque restriction dans P_0. Nous traitons a comme un symbole de fonction, et écrivons

$a[p_1, \ldots, p_n]$ au lieu de $a(p_1, \ldots, p_n)$ uniquement pour le distinguer d'un constructeur. Si a est dans *Init* ou libre dans P_0, l'arité de cette fonction est 0, et a est simplement représenté par $a[\,]$. Si a est lié par une restriction $(\nu a)P$ dans P_0, l'arité de cette fonction est le nombre de réceptions de messages, d'applications de destructions et de réplications au-dessus de $(\nu a)P$. Par exemple, dans le processus $!c(x).(\nu k)$, on utilise le motif $c[\,]$ pour le nom c et $k[i, x]$ pour tout nom créé par la restriction (νk), où i est l'identifiant de session associé à la réplication et x le message reçu sur le canal c. L'identifiant de session permet de distinguer tous les noms créés par des restrictions. (Sans lui, tous les noms créés par (νk) après avoir reçu le même message x seraient confondus.) Ceci est particulièrement important pour la preuve de propriétés de correspondances (voir section 2.2.4).

Les clauses utilisent deux prédicats attacker et message. Le fait attacker(p) signifie que l'attaquant peut avoir le message p, et le fait message(p, p') signifie que le message p' peut être envoyé sur le canal p.

$F ::=$	faits
attacker(p)	connaissance de l'attaquant
message(p, p')	message sur un canal

Les clauses comprennent à la fois des clauses qui représentent l'attaquant et des clauses qui représentent le protocole.

Clauses pour l'attaquant

Les actions de l'attaquant sont représentées par les clauses suivantes :

Pour tout $a \in$ *Init*, attacker$(a[\,])$ (Init)

attacker$(b[x])$ où b n'apparaît pas dans P_0 ni dans *Init* (Rn)

Pour tout constructeur f d'arité n,
attacker$(x_1) \wedge \ldots \wedge$ attacker$(x_n) \Rightarrow$ attacker$(f(x_1, \ldots, x_n))$ (Rf)

Pour tout destructeur g,
pour toute règle de réécriture $g(M_1, \ldots, M_n) \rightarrow M$ dans def(g),
attacker$(M_1) \wedge \ldots \wedge$ attacker$(M_n) \Rightarrow$ attacker(M) (Rg)

message$(x, y) \wedge$ attacker$(x) \Rightarrow$ attacker(y) (Rl)

attacker$(x) \wedge$ attacker$(y) \Rightarrow$ message(x, y) (Rs)

Les clauses (Init) expriment que l'attaquant connaît initialement les noms de *Init*. La clause (Rn) signifie que l'attaquant peut créer de nouveaux noms. Ces noms sont représentés par des motifs de la forme $b[x]$. Les clauses (Rf) et (Rg) expriment que l'attaquant peut appliquer tous les symboles de fonction, (Rf) pour les constructeurs, (Rg) pour les destructeurs. La clause (Rl) signifie que l'attaquant peut écouter sur les canaux qu'il a, donc s'il a le canal x et que y est envoyé sur x, alors il obtient y. La clause (Rs) signifie que l'attaquant peut envoyer n'importe quel message y qu'il a sur n'importe quel canal x qu'il a.

Clauses pour le protocole

Les clauses pour le protocole sont définies par induction sur la syntaxe du processus P_0. L'environnement ρ associe à chaque nom et variable un motif. Si f est un constructeur, on étend l'environnement ρ aux termes comme une substitution, par $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$.

La traduction $[\![P]\!]\rho s H$ d'un processus P est un ensemble de clauses, où ρ est un environnement, s est une suite de motifs qui contient les arguments à utiliser dans le codage des noms frais,

et H une suite de faits de la forme message(p, p'). La suite vide est notée $\emptyset$; la concaténation d'un motif p à la suite s est notée s, p ; la concaténation d'un fait F à la suite H est notée $H \wedge F$. La traduction $[\![P]\!]\rho s H$ est définie comme suit :

$[\![0]\!]\rho s H = \emptyset$
$[\![P \mid Q]\!]\rho s H = [\![P]\!]\rho s H \cup [\![Q]\!]\rho s H$
$[\![!P]\!]\rho s H = [\![P]\!]\rho(s, i)H$ où i est une variable fraîche
$[\![(\nu a)P]\!]\rho s H = [\![P]\!](\rho[a \mapsto a[s]])sH$
$[\![M(x).P]\!]\rho s H = [\![P]\!](\rho[x \mapsto x'])(s, x')(H \wedge \text{message}(\rho(M), x'))$ où x' est une variable fraîche
$[\![\overline{M}\langle N\rangle.P]\!]\rho s H = [\![P]\!]\rho s H \cup \{H \Rightarrow \text{message}(\rho(M), \rho(N))\}$
$[\![\mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]\rho s H =$
$\bigcup\{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma s, \sigma' p')(\sigma H) \mid g(p'_1, \dots, p'_n) \to p'$ est dans def(g) et (σ, σ') est une paire de substitutions la plus générale telle que $\sigma\rho(M_1) = \sigma' p'_1, \dots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho s H$

La traduction d'un processus est un ensemble de clauses qui permettent de dériver que le processus envoie certains messages. La suite H contient les messages reçus par le processus, qui peuvent déclencher l'envoi d'autres messages.

- Le processus nul 0 ne fait rien ; sa traduction est donc vide.
- La traduction de la composition parallèle $P \mid Q$ est l'union des traductions de P et Q, car $P \mid Q$ peut exécuter toutes les actions de P et de Q (y compris celles qui résultent d'une interaction entre P et Q).
- Dans le cas de la réplication, l'identifiant de session i est ajouté à l'environnement ρ et à la suite s. (La variable i n'est soumise à aucune contrainte spéciale ; elle peut être substituée par n'importe quel terme.) A part cette addition, la réplication est ignorée, car les clauses peuvent être appliquées un nombre quelconque de fois, en logique classique.
- Dans le cas de la restriction (νa), on ajoute à l'environnement ρ l'image de a, qui est le motif $a[s]$, où la suite s contient les identifiants de session des réplications au-dessus de (νa), les messages reçus au-dessus de (νa) et les résultats d'applications de destructeurs calculées au-dessus de (νa).
- Dans la traduction d'une réception, les suites H, ρ et s sont étendues avec le message reçu.
- La traduction d'une émission ajoute une clause, qui représente que la réception des messages de H peut déclencher l'émission du message en question.
- Enfin, la traduction de l'application de destructeur est l'union du cas où le destructeur réussit et de celui où le destructeur échoue. Dans le premier cas, ρ, s et H sont instanciés avec une substitution σ qui enregistre que le destructeur a réussi. Dans le deuxième cas, on considère que la branche else peut toujours être exécutée. Cette approximation ne pose pas de problème dans la plupart des cas, car cette branche se contente en général de ne rien faire ou d'envoyer un message d'erreur. Dans [BAF08], nous avons cependant montré comment représenter exactement dans les clauses l'échec des destructeurs, à l'aide d'un prédicat supplémentaire nounif qui exprime des propriétés de la forme $\forall x_1, \dots, x_n.p \neq p'$. (Ce prédicat est traité par des règles de simplification spécifiques dans l'algorithme de résolution.)

Les clauses qui correspondent au processus P_0 sont calculées par $[\![P_0]\!]\rho_0\emptyset\emptyset$. Ces clauses sont de la forme message$(p_1, p'_1) \wedge \dots \wedge$ message$(p_n, p'_n) \Rightarrow$ message(p, p') quand le processus P_0 envoie le message p' sur le canal p après avoir reçu les message $p'_1, \dots, p'_n$ sur les canaux $p_1, \dots, p_n$ respectivement. Quand le canal c est dans *Init*, l'attaquant a c, donc message$(c[\,], p)$ est équivalent à attacker(p) par (Rl) et (Rs). ProVerif remplace alors message$(c[\,], p)$ par attacker(p) dans les clauses générées.

Résultats et exemple

Soit $\rho_0 = \{a \mapsto a[\,] \mid a \in \mathrm{fn}(P_0)\}$. L'ensemble de clauses correspondant au processus P_0 est

$$\mathcal{R}_{P_0,\mathit{Init}} = [\![P_0]\!]\rho_0\emptyset\emptyset \cup \{\mathrm{attacker}(a[\,]) \mid a \in \mathit{Init}\} \cup \{(\mathrm{Rn}), (\mathrm{Rf}), (\mathrm{Rg}), (\mathrm{Rl}), (\mathrm{Rs})\}$$

Le théorème suivant permet de prouver le secret à partir des clauses de Horn :

Théorème 2.1 *Soit P_0 un processus clos, M un terme tel que $\mathrm{fn}(M) \subseteq \mathrm{fn}(P_0)$ et p le motif obtenu en remplaçant chaque nom a de M par $a[\,]$. Si $\mathrm{attacker}(p)$ n'est pas dérivable à partir de $\mathcal{R}_{P_0,\mathit{Init}}$ alors P_0 préserve le secret de toutes les instances de M à partir de Init.*

Nous avons effectué la preuve de ce théorème en utilisant comme intermédiaire un système de types. Plus précisément, en collaboration avec Martín Abadi [AB05a], nous avons conçu un système de types générique qui permet de prouver le secret pour des protocoles qui utilisent des primitives cryptographiques variées, définies par des constructeurs et des destructeurs. (Ce système de types étend un système précédent pour le chiffrement à clé publique [AB03].) Nous avons montré que la méthode de vérification à clauses de Horn (pour une variante sans identifiants de session) correspond à une instance particulière de ce système de types, dans laquelle les types sont les motifs clos, ce qui prouve sa correction. Nous avons également montré que cette instance est la plus précise possible : si une propriété de secret peut être prouvée par une instance quelconque de notre système de types, alors elle peut-être prouvée par la méthode à clauses de Horn.

Une petite extension de ce travail [Bla08a, annexe B] montre que la variante avec identifiants de session correspond également à un système de types, qui permet de prouver sa correction, et donc le théorème 2.1. Pour pouvoir appliquer ce théorème, il faut déterminer si un fait est dérivable à partir des clauses. ProVerif utilise pour cela un algorithme de résolution décrit dans la section 2.2.3.

Exemple 2.1 Posons $\mathit{Init} = \{c\}$ la connaissance initiale de l'attaquant. Pour le processus P_0 de la section 2.1.3, les clauses $[\![P_0]\!]\rho_0\emptyset\emptyset$ sont, après remplacement de $\mathrm{message}(c[\,], p)$ par $\mathrm{attacker}(p)$:

$$\mathrm{attacker}(\mathrm{pk}(sk_A[\,])) \tag{2.5}$$
$$\mathrm{attacker}(\mathrm{pk}(sk_B[\,])) \tag{2.6}$$
$$\mathrm{attacker}(x_pk_B) \Rightarrow \mathrm{attacker}(\mathrm{pencrypt_p}(\mathrm{sign}(k[i, x_pk_B], sk_A[\,]), x_pk_B, r[i, x_pk_B])) \tag{2.7}$$
$$\mathrm{attacker}(\mathrm{pencrypt_p}(\mathrm{sign}(x_m, sk_A[\,]), \mathrm{pk}(sk_B[\,]), x_r)) \Rightarrow \mathrm{attacker}(\mathrm{sencrypt}(\mathrm{s}[\,], x_m)) \tag{2.8}$$

Les clauses (2.5) et (2.6) correspondent aux deux émissions dans P_0 lui-même, $\overline{c}\langle pk_A\rangle\overline{c}\langle pk_B\rangle$. Elles expriment que l'attaquant a les clés publiques. La clause (2.7) correspond à l'émission dans P_A : si l'attaquant a x_pk_B, il peut l'envoyer à la première réception de P_A, et P_A répond alors avec le message $\mathrm{pencrypt_p}(\mathrm{sign}(k[i, x_pk_B], sk_A[\,]), x_pk_B, r[i, x_pk_B])$, que l'attaquant intercepte. La deuxième réception de P_A et l'application de destructeur qui suit ne génèrent aucune clause, car aucun message n'est émis. Enfin, la clause (2.8) correspond à l'émission dans P_B : si l'attaquant obtient un message de la forme $\mathrm{pencrypt_p}(\mathrm{sign}(x_m, sk_A[\,]), \mathrm{pk}(sk_B[\,]), x_r)$, il peut envoyer ce message à P_B. Le déchiffrement et la vérification de signature réussissent, donc P_B répond en envoyant s chiffré sous x_m, que l'attaquant intercepte.

Le fait $\mathrm{attacker}(\mathrm{s}[\,])$ est dérivable à partir des clauses $\mathcal{R}_{P_0,\mathit{Init}}$. On ne peut donc pas prouver le secret de s par le théorème 2.1 pour ce protocole. La dérivation obtenue par ProVerif est la suivante : l'attaquant crée une clé secrète par (Rn), d'où le fait $\mathrm{attacker}(b[x])$. Par (Rf) pour le constructeur pk, on dérive $\mathrm{attacker}(\mathrm{pk}(b[x]))$. Par (2.7), on dérive $\mathrm{attacker}(\mathrm{pencrypt_p}(\mathrm{sign}(k[i, \mathrm{pk}(b[x])], sk_A[\,]), \mathrm{pk}(b[x]), r[i, \mathrm{pk}(b[x])])))$. Par (Rg) pour le destructeur $\mathrm{pdecrypt_p}$, on dérive $\mathrm{attacker}(\mathrm{sign}(k[i, \mathrm{pk}(b[x])], sk_A[\,])$, car on a $\mathrm{attacker}(b[x])$. Par (Rg) pour le destructeur

getmessage, on dérive attacker($k[i, \mathrm{pk}(b[x])]$). Par (2.6), on a attacker($\mathrm{pk}(sk_B[\,])$), donc par (Rn) et (Rf) pour le constructeur $\mathrm{pencrypt_p}$, on dérive attacker($\mathrm{pencrypt_p}(\mathrm{sign}(k[i, \mathrm{pk}(b[x])], sk_A[\,]), \mathrm{pk}(sk_B[\,]), b[x'])$). Par (2.8), on dérive attacker($\mathrm{sencrypt}(\mathrm{s}[\,], k[i, \mathrm{pk}(b[x])])$). Enfin, sachant attacker($k[i, \mathrm{pk}(b[x])]$), on dérive attacker(s[ ]) par (Rg) pour le destructeur sdecrypt.

Cette dérivation correspond à l'attaque contre ce protocole mentionnée à la section 1.1.2, et que nous rappelons ici :

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow C: & \{\{k\}_{sk_A}\}_{pk_C} \\ \text{Message 1'.} & C(A) \rightarrow B: & \{\{k\}_{sk_A}\}_{pk_B} \\ \text{Message 2.} & B \rightarrow C(A): & \{s\}_k \end{array}$$

L'envoi du message 1 correspond à l'application de la clause (2.7) ; ensuite l'attaquant calcule le message 1' et la clé k par application de destructeurs et constructeurs correspondant aux clauses (Rg) et (Rf) de la dérivation ci-dessus. La réception du message 1' et l'envoi du message 2 correspond à la clause (2.8), et le déchiffrement final du message 2 à la clause (Rg) pour sdecrypt.

On peut modéliser de façon analogue la version corrigée du protocole, et calculer les clauses correspondantes. ProVerif montre alors que attacker(s[ ]) n'est pas dérivable à partir de ces clauses. Par le théorème 2.1, on obtient alors que le protocole corrigé préserve le secret de s à partir de $\{c\}$.

Dans l'exemple ci-dessus, nous avons expliqué informellement que la dérivation obtenue correspond à une attaque. Lors d'un stage sous ma direction, Xavier Allamigeon a étendu ProVerif pour qu'il reconstruise automatiquement une attaque à partir d'une dérivation [AB05c]. L'attaque reconstruite est une trace du processus P_0 qui publie le secret M. La stratégie utilisée pour reconstruire cette trace consiste à exécuter la sémantique du processus P_0, en se guidant à l'aide de la dérivation : une réduction du processus n'est exécutée que si une clause dans la dérivation correspond à cette réduction. Nous avons montré la correction et la terminaison de cet algorithme. Nous avons également donné une définition formelle de cette correspondance entre clauses et réductions, en donnant une construction explicite d'une dérivation à partir d'une trace de P_0. Nous avons alors montré un résultat de complétude partielle de la reconstruction d'attaques : si toutes les émissions dans P_0 sont de la forme $\overline{M}\langle N\rangle.P$ où M est un nom de *Init* non-lié dans P_0 ou bien $P = 0$, et que la dérivation correspond à une trace, alors notre algorithme réussit à reconstruire une trace correspondant à la dérivation. De plus, avec les mêmes hypothèses, notre algorithme reconstruit la trace sans faire marche arrière. Il est donc très efficace dans ce cas, et en pratique il est en général très rapide. Nous avons testé avec succès cet algorithme de reconstruction d'attaques sur de nombreux protocoles de la littérature. Pour citer un exemple extrême, nous avons pu reconstruire une attaque impliquant 200 sessions en parallèle contre le protocole $f^{200}g^{200}$ [Mil99]. (Le protocole $f^n g^n$ a une attaque qui utilise n sessions en parallèle.)

Dans l'exemple ci-dessus, la dérivation obtenue correspondait à une attaque. Ce n'est malheureusement pas toujours le cas, car la construction des clauses de Horn introduit des approximations. Ces approximations sont très utiles pour pouvoir traiter un espace d'états infini, mais à cause de ces approximations, il peut arriver que ProVerif trouve une dérivation bien que le secret soit préservé. Dans ce cas, la reconstruction de trace échoue bien sûr. Le cas où ProVerif trouve une dérivation mais la reconstruction de trace échoue correspond à une réponse "je ne sais pas". Dans les autres cas, ProVerif donne une réponse exacte : soit il ne trouve pas de dérivation et la propriété souhaitée est prouvée, soit la reconstruction de trace réussit et on obtient une attaque contre la propriété en question.

La principale approximation effectuée par ProVerif est que les clauses sont applicables un nombre quelconque de fois, donc les répétitions (ou non) des actions sont ignorées. En conséquence, les protocoles qui doivent d'abord garder un secret puis le révèlent ensuite ne peuvent pas être prouvés par ProVerif. Par exemple, le processus $P_0 = (\nu c)(\overline{c}\langle s\rangle \mid c(x).\overline{d}\langle c\rangle)$ préserve le secret de s à partir de $\{d\}$, mais ProVerif ne peut pas le prouver, car attacker($s[\,]$)

est dérivable à partir des clauses (Rl), message$(c[\,], s[\,])$ et message$(c[\,], x) \Rightarrow$ attacker$(c[\,])$ qui sont dans $\mathcal{R}_{P_0,\{d\}}$. (message$(d[\,], c[\,])$ est équivalent à attacker$(c[\,])$ car d est un canal public.) Les clauses ne prennent pas en compte que l'émission $\overline{c}\langle s\rangle$ doit avoir été exécutée avant que l'attaquant obtienne le canal c. (Dans cet exemple, c est le secret qui est d'abord gardé puis révélé.) Cet exemple peut aussi être compris en remarquant que les clauses générées sont les mêmes que pour le processus $P_0' = (\nu c)(!\overline{c}\langle s\rangle \mid !c(x).\overline{d}\langle c\rangle)$, où les actions sont répétées et qui, lui, ne préserve pas le secret de s à partir de $\{d\}$.

Dans [Bla05], nous avons comparé le modèle en logique linéaire [CDL+99] (ou de façon équivalente le modèle à réécriture de multi-ensembles) avec le modèle abstrait à clauses de Horn (variante sans identifiants de session). Nous avons montré que ce dernier est obtenu à partir du modèle en logique linéaire par une abstraction (au sens formel de l'interprétation abstraite [CC79]) qui ignore le nombre de répétitions de chaque action. Le modèle en logique linéaire représente le protocole par des formules de la forme :

$$!\forall y_1, \dots, y_p.(F_1 \otimes \dots \otimes F_n \multimap \exists x_1, \dots x_m.F_1' \otimes \dots \otimes F_{n'}') \tag{2.9}$$

où les existentiels correspondent à la création de noms frais. Dans la présentation par multi-ensembles, l'état du système est un multi-ensemble de faits, et quand on applique la formule (2.9), on retire de l'état des faits instance de $F_1, \dots, F_n$ et on ajoute les instances correspondantes de $F_1', \dots, F_{n'}'$, après avoir remplacé $x_1, \dots x_m$ par des noms frais. Après abstraction, on obtient des formules en logique classique, de la forme

$$\forall y_1, \dots, y_p.(F_1 \wedge \dots \wedge F_n \Rightarrow \exists x_1, \dots, x_m.F_1' \wedge \dots \wedge F_{n'}') \tag{2.10}$$

qui peuvent être transformées en clauses de Horn après skolemisation de $x_1, \dots, x_m$. C'est cette skolemisation qui transforme les noms frais en fonctions des messages précédemment reçus, sans introduire de nouvelle approximation, car la skolemisation préserve la satisfiabilité en logique classique. (Par contre, la skolemisation introduit une approximation si elle est faite en logique linéaire.) Ainsi, par rapport au modèle en logique linéaire, la seule approximation est celle du nombre de répétition des actions.

Par contre, par rapport à notre calcul de processus, qui est plus riche que le modèle en logique linéaire, on peut noter une approximation supplémentaire : pour l'émission $\overline{M}\langle N\rangle.P$, la représentation à clauses de Horn considère que le processus P peut toujours être exécuté, comme si le processus était $\overline{M}\langle N\rangle \mid P$, alors qu'en fait P ne peut être exécuté qu'après avoir émis N sur le canal M. (Les branches else des destructeurs sont aussi approchées dans la version présentée ici, mais, comme noté ci-dessus, ProVerif les traite précisément.)

2.2.3 Résolution sur les clauses

Afin de déterminer si un fait est dérivable à partir des clauses, ProVerif utilise la résolution avec sélection libre [dN95, Lyn97, BG01] (alors que Weidenbach [Wei99] utilisait la résolution avec sélection ordonnée). Nous rappelons cet algorithme et résumons les principales optimisations implantées dans ProVerif, puis nous discutons sa terminaison.

L'algorithme

L'algorithme de résolution infère de nouvelles clauses comme suit : à partir de deux clauses $R = H \Rightarrow C$ et $R' = F \wedge H' \Rightarrow C'$ (où F est une hypothèse quelconque de R'), il infère $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$ où C et F sont unifiables et σ est l'unificateur le plus général de C et F. La clause $R \circ_F R'$ combine donc R et R', de sorte que R est utilisée pour prouver l'hypothèse F de R'. La résolution est guidée par une fonction de sélection sel : sel(R) retourne un fait (une hypothèse ou la conclusion) de R, et l'étape de résolution ci-dessus n'est effectuée que si sel$(R) = C$ et sel$(R') = F$. L'algorithme de saturation saturate$(\mathcal{R}_0)$ applique ces étapes

de résolution jusqu'à ce qu'un point fixe soit atteint, c'est-à-dire qu'aucune nouvelle clause ne soit créée. Quand le point fixe est atteint, saturate($\mathcal{R}_0$) retourne le sous-ensemble des clauses R dans le point fixe telles que sel(R) est la conclusion de R[4].

L'algorithme de résolution avec sélection libre est correct[5] pour n'importe quelle fonction de sélection, mais le choix de cette fonction influence considérablement ses performances (et sa terminaison). On peut remarquer que le fait attacker(x) où x est une variable s'unifie avec n'importe quel fait attacker(p), donc si attacker(x) est sélectionné, l'algorithme ne terminera pratiquement jamais. On évite donc de sélectionner attacker(x). Une fonction de sélection naturelle est donc :

$$\mathrm{sel}_0(H \Rightarrow C) = \begin{cases} C & \text{si tous les éléments de } H \text{ sont de la forme attacker}(x),\ x \text{ variable} \\ F & \text{où } F \neq \mathrm{attacker}(x) \text{ et } F \in H, \text{ sinon} \end{cases}$$

L'algorithme implanté dans ProVerif contient de nombreuses optimisations. Nous résumons ci-dessous les principales. D'autres optimisations sont présentées dans [Bla08a] ainsi que, pour des prédicats spécifiques comme nounif, dans [Bla04a, BAF08]. Ces optimisations sont appliquées sur les clauses initiales et après chaque étape de résolution. Certaines de ces optimisations sont spécifiques des protocoles, comme les deux premières ci-dessous, alors que d'autres sont standard.

- Décomposition des constructeurs de données : on appelle *constructeur de données* un constructeur f d'arité n accompagné de n destructeurs g_i définis par $g_i(f(x_1, \dots, x_n)) \to x_i$. Des exemples typiques de constructeurs de données sont les n-uplets.
  Si f est un constructeur de données, attacker($f(p_1, \dots, p_n)$) est équivalent à attacker(p_1) $\wedge \dots \wedge$ attacker(p_n) par les clauses (Rf) attacker(x_1) $\wedge \dots \wedge$ attacker(x_n) $\Rightarrow$ attacker($f(x_1, \dots, x_n)$) et (Rg) attacker($f(x_1, \dots, x_n)$) $\Rightarrow$ attacker(x_i). On remplace alors le fait attacker($f(p_1, \dots, p_n)$) par attacker(p_1) $\wedge \dots \wedge$ attacker(p_n) dans les clauses. Si ce remplacement est effectué dans la conclusion d'une clause, n clauses sont créées avec les mêmes hypothèses et les conclusions attacker(p_1), ..., attacker(p_n) respectivement. Ce remplacement est effectué récursivement : si p_i est encore une application d'un constructeur de données, on effectue à nouveau le même remplacement.
  Les clauses (Rf) et (Rg) correspondant au constructeur de données et à ses destructeurs associés sont exclues de cette transformation.
- Élimination des hypothèses attacker(x) : les hypothèses attacker(x), où x n'apparaît nulle part ailleurs dans la clause, sont supprimées. En effet, ces hypothèses peuvent toujours être satisfaites, par exemple par (Rn).
- Élimination des hypothèses dupliquées : on ne conserve qu'une copie des hypothèses présentes en double dans une clause.
- Élimination des tautologies : les tautologies (clauses dont la conclusion est déjà présente dans les hypothèses) sont supprimées.
- Élimination des clauses subsumées : on dit que $H_1 \Rightarrow C_1$ subsume $H_2 \Rightarrow C_2$ si et seulement s'il existe une substitution σ telle que $\sigma H_1 \subseteq H_2$ (inclusion de multi-ensembles) et $\sigma C_1 = C_2$. On élimine toutes les clauses qui sont subsumées par une autre clause de l'ensemble de clauses courant.

La correction de cet algorithme est justifiée par le théorème suivant :

Théorème 2.2 *Soit F un fait clos. Le fait F est dérivable à partir de $\mathcal{R}_0$ si et seulement s'il est dérivable à partir de* saturate($\mathcal{R}_0$).

[4]Pour des raisons historiques, les articles publiés utilisent une notation légèrement différente pour la fonction de sélection sel : sel(R) retourne un sous-ensemble des hypothèses de R, avec sel($H \Rightarrow C$) = $\{F\}$ quand $F \in H$ est sélectionnée et sel($H \Rightarrow C$) = $\emptyset$ quand la conclusion C est sélectionnée.

[5]Dans ce mémoire, la notion de correction est comprise au sens de la sécurité, c'est-à-dire qu'aucune attaque n'est omise. Cette notion de correction correspond pour l'algorithme de résolution au fait qu'aucune dérivation n'est omise, c'est-à-dire à la *complétude* de l'algorithme au sens de la programmation logique.

Ce théorème est un cas particulier de [Bla08a, Lemme 2]. Il montre qu'on peut saturer les clauses par saturate sans changer l'ensemble des faits dérivables. On peut alors déterminer quelles instances de $pred(p_1, \dots, p_n)$ sont dérivables par le calcul suivant : $\text{solve}_{P_0,Init}(pred(p_1, \dots, p_n)) = \{H \Rightarrow pred(p'_1, \dots, p'_n) \mid H \Rightarrow pred'(p'_1, \dots, p'_n) \in \text{saturate}(\mathcal{R}_0)\}$, où $pred'$ est un nouveau prédicat et $\mathcal{R}_0 = \mathcal{R}_{P_0,Init} \cup \{pred(p_1, \dots, p_n) \Rightarrow pred'(p_1, \dots, p_n)\}$. En effet, $\sigma pred(p_1, \dots, p_n)$ est dérivable à partir de $\mathcal{R}_{P_0,Init}$ si et seulement si $\sigma pred'(p_1, \dots, p_n)$ est dérivable à partir de $\mathcal{R}_0$, donc, par le théorème 2.2, si et seulement si $\sigma pred'(p_1, \dots, p_n)$ est dérivable à partir de saturate($\mathcal{R}_0$), donc si et seulement s'il existe une clause $H \Rightarrow pred(p'_1, \dots, p'_n)$ dans $\text{solve}_{P_0,Init}(pred(p_1, \dots, p_n))$ et une substitution σ' telles que $\sigma' pred(p'_1, \dots, p'_n) = \sigma pred(p_1, \dots, p_n)$ et $\sigma' H$ est dérivable à partir de saturate($\mathcal{R}_0$). En particulier, si $\text{solve}_{P_0,Init}(\text{attacker}(p)) = \emptyset$, attacker($p$) n'est pas dérivable à partir de $\mathcal{R}_{P_0,Init}$ (et si $\text{solve}_{P_0,Init}(\text{attacker}(p))$ est non-vide pour la fonction de sélection sel_0, au moins une instance de attacker(p) est dérivable, puisque H contiendra des faits de la forme attacker(x) dont une instance est dérivable par (Rn)).

Terminaison

L'algorithme de saturation saturate ne termine pas toujours. En collaboration avec Andreas Podelski [BP05b], nous avons montré qu'il termine pour une classe importante de protocoles, les protocoles étiquetés (*tagged protocols*). Un protocole étiqueté est un protocole dans lequel chaque application d'une primitive cryptographique est distinguée des autres par une constante (étiquette). Il est en général facile de transformer un protocole en un protocole étiqueté, en ajoutant des étiquettes. Par exemple, on peut transformer le protocole de la section 1.1.2 en un protocole étiqueté :

$$\begin{array}{llll} \text{Message 1.} & A \rightarrow B : & \{c_1, \{c_0, k\}_{sk_A}\}_{pk_B} & k \text{ fraîche} \\ \text{Message 2.} & B \rightarrow A : & \{c_2, s\}_k & \end{array}$$

où les étiquettes sont c_0, c_1, c_2. Le protocole étiqueté conserve le comportement attendu du protocole, c'est-à-dire que les exécutions sans attaques sont les mêmes. En présence d'attaques, il peut être plus sûr. L'ajout d'étiquettes participe donc à la bonne conception des protocoles, comme expliqué par exemple dans [AN96] : le récepteur d'un message utilise l'étiquette pour l'identifier sans ambiguïté. L'étiquetage évite par conséquent les confusions de types qui apparaissent quand un message est pris pour un autre message. (Ceci est prouvé formellement dans [HLS00] pour un schéma d'étiquetage très proche du nôtre.) Ceci signifie aussi que la sécurité du protocole étiqueté n'implique pas la sécurité de la version non-étiquetée ; il faut donc implanter la version étiquetée. L'étiquetage est aussi motivé par des raisons pratiques, car il facilite le décodage des messages reçus. Pour toutes ces raisons, l'étiquetage est déjà présent dans des protocoles comme SSH.

Nous avons montré que notre algorithme de vérification termine pour les protocoles étiquetés qui utilisent les primitives cryptographiques de la figure 2.2, pourvu que les clés publiques soient atomiques. Dans [BP05b], nous donnons une caractérisation de ces protocoles au niveau des clauses de Horn, alors que dans [Bla08a, section 8.1], nous avons étendu ce résultat en donnant une caractérisation au niveau des processus. L'algorithme termine souvent même quand le protocole n'est pas étiqueté. Ceci peut s'expliquer en partie car, dans certains protocoles, la forme des messages garantit qu'ils ne peuvent pas être confondus les uns avec les autres, même sans étiquettes, par exemple parce qu'ils contiennent des n-uplets d'arité différente ou qu'ils utilisent des primitives cryptographiques différentes. C'est d'ailleurs le cas du protocole de la section 1.1.2. On parle alors d'étiquetage implicite.

D'autres auteurs ont prouvé des résultats liés : Ramanujan et Suresh [RS03] ont montré que le secret est décidable pour les protocoles étiquetés. Leur résultat diffère du nôtre pour deux raisons. Leur schéma d'étiquetage est plus restrictif, car il interdit les copies aveugles. Une copie aveugle se produit quand un participant renvoie une partie d'un message qu'il a reçu sans vérifier ce qui est à l'intérieur de cette partie. D'autre part, ils donnent un résultat de

décidabilité, tandis que notre résultat montre la terminaison d'un algorithme correct, efficace en pratique, mais approché. Arapinis et Duflot ont étendu ce résultat [AD07], en interdisant toujours les copies aveugles. Comon-Lundh et Cortier [CLC03] ont montré la terminaison d'un algorithme qui utilise la résolution binaire ordonnée, la factorisation ordonnée et le *splitting* sur les protocoles qui font au plus une copie aveugle dans chaque message. Notre résultat ne fixe aucune limite sur le nombre de copies aveugles, mais requiert l'étiquetage.

Nous avons également conçu des heuristiques pour améliorer le choix de la fonction de sélection, en vue de favoriser la terminaison de l'algorithme même quand le protocole n'est pas étiqueté [Bla08a, section 8.2].

2.2.4 Vérification des propriétés de correspondances

Les propriétés de correspondances sont des propriétés de la forme "si un certain événement a été exécuté, alors d'autres événements ont été exécutés". Afin de modéliser ces propriétés, on introduit donc une construction supplémentaire dans notre calcul de processus $\mathsf{event}(M).P$ qui exécute l'événement M, puis le processus P. La sémantique de cette construction est définie simplement par

$$E, \mathcal{P} \cup \{ \mathsf{event}(M).P \} \rightarrow E, \mathcal{P} \cup \{ P \} \qquad \text{(Red Event)}$$

Les *Init*-attaquants sont restreints aux processus qui ne contiennent pas d'événements (sinon, aucune correspondance ne pourrait être prouvée). On définit alors le fait qu'une trace exécute un événement :

Définition 2.4 Soit M un terme clos. On dit qu'une trace $E_0, \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ *exécute l'événement* M si et seulement s'il existe E, $\mathcal{P}$ et P tels que cette trace contienne la réduction $E, \mathcal{P} \cup \{ \mathsf{event}(M).P \} \rightarrow E, \mathcal{P} \cup \{ P \}$.

La correspondance $\mathrm{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk})$ signifie intuitivement que, si l'événement M a été exécuté, alors il existe j tel que les événements $M_{j1}, \dots, M_{jl_j}$ ont été exécutés. Plus précisément, pour toute valeur des variables de M, si l'événement M a été exécuté, alors il existe j et des valeurs des variables de $M_{j1}, \dots, M_{jl_j}$ qui n'apparaissent pas dans M tels que les événements $M_{j1}, \dots, M_{jl_j}$ ont été exécutés. La définition formelle est la suivante :

Définition 2.5 Le processus clos P_0 *satisfait la correspondance*

$$\mathrm{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk})$$

en présence d'un *Init*-attaquant si et seulement si, pour tout *Init*-attaquant Q, pour tout E_0 contenant $\mathrm{fn}(P_0) \cup \mathit{Init} \cup \mathrm{fn}(M) \cup \bigcup_{j,k} \mathrm{fn}(M_{jk})$, pour toute substitution σ, pour toute trace $\mathcal{T} = E_0, \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, si $\mathcal{T}$ exécute l'événement σM, alors il existe σ' et $j \in \{1, \dots, m\}$ tels que $\sigma' M = \sigma M$ et, pour tout $k \in \{1, \dots, l_j\}$, $\mathcal{T}$ exécute l'événement $\sigma' M_{jk}$.

Exemple 2.2 Par exemple, on peut modifier le processus P_0 de la section 2.1.3 en ajoutant des événements comme suit :

$$\begin{aligned}
&P_0 = (\nu sk_A)(\nu sk_B)\mathsf{let}\ pk_A = \mathrm{pk}(sk_A)\ \mathsf{in}\ \mathsf{let}\ pk_B = \mathrm{pk}(sk_B)\ \mathsf{in}\ \overline{c}\langle pk_A\rangle\overline{c}\langle pk_B\rangle.\\
&\qquad (P_A(pk_A, sk_A) \mid P_B(pk_B, sk_B, pk_A))\\
&P_A(pk_A, sk_A) = !\ c(x_pk_B).(\nu k)\mathsf{event}(e_A(pk_A, x_pk_B, k)).\\
&\qquad (\nu r)\overline{c}\langle \mathrm{pencrypt_p}(\mathrm{sign}(k, sk_A), x_pk_B, r)\rangle.c(x).\mathsf{let}\ z = \mathrm{sdecrypt}(x, k)\ \mathsf{in}\ 0\\
&P_B(pk_B, sk_B, pk_A) = !\ c(y).\mathsf{let}\ y' = \mathrm{pdecrypt_p}(y, sk_B)\ \mathsf{in}\\
&\qquad \mathsf{let}\ x_k = \mathrm{checksignature}(y', pk_A)\ \mathsf{in}\ \mathsf{event}(e_B(pk_A, pk_B, x_k)).\overline{c}\langle \mathrm{sencrypt}(\mathrm{s}, x_k)\rangle
\end{aligned}$$

L'événement $e_A(pk_A, x_pk_B, k)$ signifie intuitivement que A a démarré une session du protocole entre les participants de clés publiques pk_A (c'est-à-dire A) et x_pk_B, avec la clé partagée k. De façon analogue, l'événement $e_B(pk_A, pk_B, x_k)$ signifie que B a accepté la clé partagée k dans une session entre les participants A et B. On peut alors chercher à montrer la correspondance $\text{event}(e_B(x, y, z)) \rightsquigarrow \text{event}(e_A(x, y, z))$, qui signifie que, si $e_B(x, y, z)$ a été exécuté, alors $e_A(x, y, z)$ a aussi été exécuté. Autrement dit, si B pense exécuter une session du protocole avec A et la clé partagée z, alors A pense exécuter une session du protocole avec B et la même clé z. Ceci fournit une forme d'authentification.

Comme mentionné ci-dessus, notre méthode de vérification par clauses de Horn surapproxime les actions qui peuvent être exécutées. Ainsi, si $\text{attacker}(p)$ est dérivable à partir des clauses, l'attaquant *peut* avoir p : si $\text{attacker}(p)$ n'est pas dérivable, alors on est sûr que l'attaquant n'a pas p, mais la réciproque est fausse. Supposons maintenant que l'on souhaite prouver une propriété de correspondance telle que $\text{event}(e_1(x)) \rightsquigarrow \text{event}(e_2(x))$, c'est-à-dire que l'on veut montrer que, si $e_1(x)$ a été exécuté, alors $e_2(x)$ a été exécuté. Pour faire une telle preuve, on peut surapproximer les exécutions de e_1 : si la preuve réussit avec cette surapproximation, la propriété sera a fortiori vraie dans sémantique exacte. On étend donc l'analyse pour le secret avec un prédicat supplémentaire event, tel que $\text{event}(p)$ signifie que l'événement p (plus formellement, l'événement M de motif associé p) peut avoir été exécuté. On crée des clauses $\text{message}(p_1, p'_1) \wedge \ldots \wedge \text{message}(p_n, p'_n) \Rightarrow \text{event}(p)$ quand le processus exécute l'événement p après avoir reçu les messages $p'_1, \ldots, p'_n$ sur les canaux $p_1, \ldots, p_n$ respectivement. Par contre, on ne peut pas surapproximer les exécutions de l'événement e_2 : si on prouve la correspondance après surapproximation de e_2, on n'est pas vraiment sûr que e_2 va être exécuté, et donc la correspondance peut être fausse dans la sémantique exacte. On doit donc utiliser une autre méthode pour traiter e_2.

Nous utilisons l'idée suivante : nous fixons l'ensemble exact $\mathcal{E}$ des événements autorisés $e_2(p)$ et, pour prouver $\text{event}(e_1(x)) \rightsquigarrow \text{event}(e_2(x))$, nous vérifions que seuls les événements $e_1(p)$ pour p tel que $e_2(p) \in \mathcal{E}$ peuvent être exécutés. Donc, si $e_1(M)$ a été exécuté, alors $e_1(p)$ a été exécuté pour p le motif correspondant à M, donc $e_2(p) \in \mathcal{E}$ a été exécuté, donc $e_2(M)$ a été exécuté, car un seul terme M correspond à un motif p donné grâce aux identifiants de session qui permettent de distinguer les noms créés par une même restriction. En prouvant cette propriété pour toute valeur de $\mathcal{E}$, on obtient la correspondance souhaitée. On introduit donc un prédicat m-event (*must event*) tel que $\text{m-event}(p_0)$ est vrai si et seulement si $p_0 \in \mathcal{E}$. On crée les clauses $\text{message}(p_1, p'_1) \wedge \ldots \wedge \text{message}(p_n, p'_n) \wedge \text{m-event}(p_0) \Rightarrow \text{message}(p, p')$ quand le processus émet p' sur le canal p après avoir exécuté l'événement p_0 et reçu $p'_1, \ldots, p'_n$ sur les canaux $p_1, \ldots, p_n$ respectivement. Autrement dit, l'émission de p' sur le canal p peut être exécutée seulement si $\text{m-event}(p_0)$ est vrai, c'est-à-dire $p_0 \in \mathcal{E}$.

Plus généralement, on étend les formules de calcul des clauses au cas des événements comme suit :

$$[\![\mathsf{event}(M).P]\!]\rho s H = [\![P]\!]\rho s(H \wedge \text{m-event}(\rho(M))) \cup \{H \Rightarrow \text{event}(\rho(M))\}$$

On ajoute l'hypothèse $\text{m-event}(\rho(M))$ à H pour exprimer que P ne peut être exécuté que si l'événement M est autorisé (ce qui est utile pour e_2 dans l'exemple ci-dessus), et on ajoute la clause $H \Rightarrow \text{event}(\rho(M))$ pour exprimer que l'événement peut être exécuté si H est vrai (ce qui est utile pour e_1 dans l'exemple ci-dessus).

Pour déterminer si un événement peut être exécuté, on détermine si le fait correspondant est dérivable à partir des clauses, en étendant l'algorithme de résolution précédent. En effet, la résolution doit être effectuée pour une valeur inconnue de $\mathcal{E}$. Donc on garde les faits m-event sans essayer de les évaluer. (Les évaluer nécessite de connaître $\mathcal{E}$.) Pour cela, on modifie la fonction de sélection pour qu'elle ne sélectionne jamais un fait de la forme $\text{m-event}(p)$. En notant $\mathcal{F}_{\text{me}} = \{\text{m-event}(p) \mid p \in \mathcal{E}\}$, le théorème 2.2 devient alors :

Théorème 2.3 *Soit F un fait clos. Le fait F est dérivable à partir de $\mathcal{R}_0 \cup \mathcal{F}_{\mathrm{me}}$ si et seulement s'il est dérivable à partir de* $\mathrm{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$.

Alors, de même que pour le secret, $\sigma pred(p_1, \dots, p_n)$ est dérivable à partir de $\mathcal{R}_{P_0,Init} \cup \mathcal{F}_{\mathrm{me}}$ si et seulement s'il existe une clause $H \Rightarrow pred(p'_1, \dots, p'_n)$ dans $\mathrm{solve}_{P_0,Init}(pred(p_1, \dots, p_n))$ et une substitution σ' telles que $\sigma' pred(p'_1, \dots, p'_n) = \sigma pred(p_1, \dots, p_n)$ et $\sigma' H$ est dérivable à partir de $\mathrm{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$, où $\mathcal{R}_0 = \mathcal{R}_{P_0,Init} \cup \{pred(p_1, \dots, p_n) \Rightarrow pred'(p_1, \dots, p_n)\}$. Comme les faits m-event ne sont conclus par aucune clause hors de $\mathcal{F}_{\mathrm{me}}$, les faits m-event de $\sigma' H$ sont nécessairement dans $\mathcal{F}_{\mathrm{me}}$, ce qui permet de garantir que les événements correspondants ont été exécutés. On peut alors montrer le théorème suivant :

Théorème 2.4 *Soit P_0 un processus clos. Soient M, M_{jk} ($j \in \{1, \dots, m\}$, $k \in \{1, \dots, l_j\}$) des termes. Soient p, p_{jk} les motifs obtenus en remplaçant les noms a par les motifs $a[\,]$ dans les termes M, M_{jk} respectivement. Supposons que, pour toutes les clauses $R \in \mathrm{solve}_{P_0,Init}(\mathrm{event}(p))$, il existe $j \in \{1, \dots, m\}$, σ' et H tels que $R = H \wedge \text{m-event}(\sigma' p_{j1}) \wedge \dots \wedge \text{m-event}(\sigma' p_{jl_j}) \Rightarrow \mathrm{event}(\sigma' p)$. Alors P_0 satisfait la correspondance $\mathrm{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk})$ en présence d'un Init-attaquant.*

Ce résultat est un cas particulier de [Bla08a, Théorème 4]. Intuitivement, si toutes les clauses sont de la forme $H \wedge \text{m-event}(\sigma' p_{j1}) \wedge \dots \wedge \text{m-event}(\sigma' p_{jl_j}) \Rightarrow \mathrm{event}(\sigma' p)$, alors, pour pouvoir dériver $\mathrm{event}(\sigma' p)$, il faut que $\text{m-event}(\sigma' p_{j1})$, …, $\text{m-event}(\sigma' p_{jl_j})$ soient vrais, donc pour pouvoir exécuter l'événement $\sigma' p$, il faut avoir exécuté les événements $\sigma' p_{j1}$, …, $\sigma' p_{jl_j}$, ce qui montre la correspondance souhaitée.

Exemple 2.3 Dans le processus de l'exemple 2.2, $\mathrm{solve}_{P_0,Init}(\mathrm{event}(e_B(x, y, z)))$ contient la clause $\text{m-event}(e_A(\mathrm{pk}(sk_A[\,]), \mathrm{pk}(y'), k[i, \mathrm{pk}(y')])) \wedge \mathrm{attacker}(y') \Rightarrow \mathrm{event}(e_B(\mathrm{pk}(sk_A[\,]), \mathrm{pk}(sk_B[\,]), k[i, \mathrm{pk}(y')])))$. Cette clause empêche d'appliquer le théorème 2.4 pour prouver la correspondance $\mathrm{event}(e_B(x, y, z)) \rightsquigarrow \mathrm{event}(e_A(x, y, z))$, car le fait $\text{m-event}(e_A(\mathrm{pk}(sk_A[\,]), \mathrm{pk}(y'), k[i, \mathrm{pk}(y')]))$ contient $\mathrm{pk}(y')$ au lieu de $\mathrm{pk}(sk_B[\,])$. Cela correspond à nouveau à l'attaque connue contre ce protocole : A exécute une session avec C de clé secrète y' et clé publique $\mathrm{pk}(y')$, alors que B pense exécuter une session avec A. Par contre, pour la version corrigée du protocole, $\mathrm{solve}_{P_0,Init}(\mathrm{event}(e_B(x, y, z))) = \{\text{m-event}(e_A(\mathrm{pk}(sk_A[\,]), \mathrm{pk}(sk_B[\,]), k[i, \mathrm{pk}(sk_B[\,])])) \Rightarrow \mathrm{event}(e_B(\mathrm{pk}(sk_A[\,]), \mathrm{pk}(sk_B[\,]), k[i, \mathrm{pk}(sk_B[\,])]))\}$, donc la correspondance souhaitée est prouvée.

Nous avons étendu ces résultats aux correspondances injectives, c'est-à-dire dans lesquelles on requiert de plus que chaque exécution de l'événement M correspond à une exécution *distincte* des événements M_{jk}. La preuve de l'injectivité exploite les identifiants de sessions pour distinguer les différentes exécutions du même événement. Nous avons également étendu ce travail aux correspondances imbriquées, qui permettent d'exprimer des contraintes sur l'ordre dans lequel les événements sont exécutés [Bla08a, section 7.2].

La reconstruction d'attaques a été étendue aux correspondances non-injectives, et elle reconstruit l'attaque dans l'exemple 2.3. Nous prévoyons de l'étendre aux correspondances injectives. (La difficulté est que la dérivation correspond à une exécution de l'événement M alors que, pour contredire l'injectivité, il faut exécuter l'événement M deux fois quand un événement M_{jk} est exécuté au plus une fois.)

2.2.5 Scénarios à plusieurs phases

Dans certaines études de protocoles, on considère des scénarios dans lesquels un certain processus est exécuté, puis, dans une deuxième phase, ce processus s'arrête et l'exécution d'un autre processus commence. Par exemple, quand on modélise le compromis de clés à long terme, on considère que, dans une première phase, le protocole est exécuté normalement, puis, dans une deuxième phase, certaines clés sont publiées. On cherche alors à savoir quels secrets des

sessions du protocole exécutées dans la première phase sont préservés malgré le compromis des clés (c'est la notion de *forward secrecy*).

De tels scénarios peuvent être représentés dans ProVerif grâce à une extension de la syntaxe : le processus $t : P$ représente un processus P qui s'exécute dans la phase numéro t. Le système exécute tout d'abord les processus en phase 0. Puis, à un certain moment dans l'exécution, on passe à la phase 1. À ce moment, seuls sont conservés les processus $t : P$ pour $t \geq 1$ prêts à être exécutés (c'est-à-dire que les processus en phase 0 sont arrêtés) et les processus $1 : P$ sont exécutés. Ensuite, on passe à la phase 2, et ainsi de suite.

Cette extension est traduite en clauses de Horn de la façon suivante. On considère des prédicats $\mathrm{attacker}_t$ et $\mathrm{message}_t$ pour chaque phase t, au lieu des prédicats attacker et message. Les clauses pour le protocole utilisent le prédicat $\mathrm{message}_t$ pour traduire le processus P dans $t : P$; les clauses pour l'attaquant sont répétées pour chaque $\mathrm{attacker}_t$. De plus, les clauses

$$\mathrm{attacker}_t(x) \Rightarrow \mathrm{attacker}_{t+1}(x) \tag{Rp}$$

pour tout t permettent de transmettre la connaissance de l'attaquant d'une phase à la suivante.

Cette extension a été présentée dans [BAF08, section 8] et [Bla08a, section 9.3]. Une application de cette extension sera mentionnée dans la section suivante.

2.2.6 Preuves d'équivalences

La preuve d'équivalences de processus est la méthode de preuve introduite initialement avec le spi calcul [AG99, AG98] et le pi calcul appliqué [AF01], dans le cadre de preuves manuelles. Intuitivement, deux processus sont équivalents quand l'attaquant ne peut pas les distinguer. La preuve de telles équivalences étant difficile à automatiser, nous nous sommes intéressés seulement à certains cas particuliers d'équivalences plus faciles à traiter, mais quand même importants en pratique.

Nous nous sommes tout d'abord intéressés à la preuve du secret fort (dans le cas sans théorie équationnelle) [Bla04a, Bla04b] : le secret fort signifie que l'attaquant ne peut pas distinguer deux versions du protocole qui utilisent des valeurs différentes du secret. Pour prouver le secret fort, nous nous ramenons à une propriété de trace : nous montrons qu'aucun test (application de destructeur, communication sur un canal) qui donne un résultat différent pour différentes valeurs du secret n'est accessible. Nous codons cette propriété d'accessibilité à l'aide de clauses de Horn, avec un prédicat supplémentaire utilisé pour tester si une unification réussit pour certaines valeurs du secret et pas pour d'autres.

En collaboration avec Martín Abadi et Cédric Fournet [BAF05, BAF08], nous nous sommes également intéressés à une classe plus générale d'équivalences, mais dont la preuve est aussi plus coûteuse : les équivalences entre deux processus P et Q qui ne diffèrent que par les termes qu'ils contiennent. Ces équivalences sont là encore prouvées en se ramenant à une propriété de trace, sur un processus qui représente à la fois P et Q. Cette idée étend la technique de Pottier et Simonet [PS02, Pot02] pour le flot d'information (sans cryptographie) au cas des protocoles cryptographiques.

Plus formellement, on peut définir l'équivalence entre les processus P et Q comme suit. (Cette présentation s'inspire de celle de la thèse de Mathieu Baudet [Bau07], en particulier pour correspondre à la sémantique définie à la section 2.1.4, tandis que notre présentation initiale [BAF05, BAF08] utilisait une sémantique avec équivalence structurelle.)

Définition 2.6 Soit *Init* un ensemble fini de noms (qui représente la connaissance initiale de l'attaquant). On considère uniquement les configurations $\mathcal{C} = E, \mathcal{P}$ telles que $\mathit{Init} \subseteq E$.

On dit qu'une configuration $\mathcal{C} = E, \mathcal{P}$ *émet sur* N, et on note $\mathcal{C} \downarrow_N$, si et seulement s'il existe M et P tels que $\overline{N}\langle M \rangle.P \in \mathcal{P}$.

Si Q est un *Init*-attaquant et $\mathcal{C} = E, \mathcal{P}$ une configuration, on définit $\mathcal{C} \mid Q = E, \mathcal{P} \cup \{Q\}$.

L'*équivalence observationnelle* $\approx_{Init}$ est la plus grande relation symétrique $\mathcal{R}$ sur les configurations telle que $\mathcal{C}\ \mathcal{R}\ \mathcal{C}'$ implique

1. pour tout $a \in Init$, si $\mathcal{C} \downarrow_a$, alors $\mathcal{C}' \rightarrow^* \downarrow_a$;
2. si $\mathcal{C} \rightarrow \mathcal{C}_1$, alors il existe $\mathcal{C}'_1$ telle que $\mathcal{C}' \rightarrow^* \mathcal{C}'_1$ et $\mathcal{C}_1\ \mathcal{R}\ \mathcal{C}'_1$;
3. pour tout *Init*-attaquant Q, $(\mathcal{C} \mid Q)\ \mathcal{R}\ (\mathcal{C}' \mid Q)$.

On dit que $P \approx_{Init} P'$ si et seulement si $Init \cup \text{fn}(P), \{P\} \approx_{Init} Init \cup \text{fn}(P'), \{P'\}$.

On définit d'abord l'équivalence observationnelle sur les configurations sémantiques. Le point 1 de cette définition garantit que si une configuration $\mathcal{C}$ émet sur un canal public, alors $\mathcal{C}'$ aussi, sinon l'attaquant pourrait les distinguer immédiatement. Le point 2 exprime que l'équivalence est préservée par réduction, alors que le point 3 exprime qu'elle est préservée en présence d'un *Init*-attaquant. Enfin, on définit l'équivalence de deux processus à partir de l'équivalence sur les configurations. (La connaissance de l'attaquant *Init* est ici explicite, par analogie avec les définitions utilisées pour le secret et les correspondances, alors que, dans la plupart des travaux sur ce sujet, elle correspond aux noms libres des processus.)

On introduit maintenant un nouveau calcul qui permet de représenter des paires de processus qui ne diffèrent que par les termes qu'ils contiennent, et qu'on appelle *biprocessus*. La grammaire de ce calcul est une extension de la grammaire de la figure 2.1, avec le cas supplémentaire $\mathsf{diff}[M, M']$ pour les termes. Les *Init*-attaquants sont des processus sans diff. Étant donné un biprocessus P, on définit deux processus $\text{fst}(P)$ et $\text{snd}(P)$, comme suit : $\text{fst}(P)$ est obtenu en remplaçant toutes les occurrences de $\mathsf{diff}[M, M']$ par M dans P, et $\text{snd}(P)$ est obtenu en remplaçant $\mathsf{diff}[M, M']$ par M' dans P. On définit $\text{fst}(M)$ et $\text{snd}(M)$ de façon similaire. Notre but est de montrer que les processus $\text{fst}(P)$ et $\text{snd}(P)$ sont observationnellement équivalents.

Définition 2.7 Un biprocessus clos P *satisfait la Init-équivalence* si et seulement si $\text{fst}(P) \approx_{Init} \text{snd}(P)$.

La sémantique des biprocessus est définie comme dans la figure 2.3, sauf que les règles (Red I/O), (Red Destr 1) et (Red Destr 2) sont les suivantes :

$$E, \mathcal{P} \cup \{\, \overline{N}\langle M\rangle.Q, N'(x).P \,\} \rightarrow E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\} \quad \text{(Red I/O)}$$
$$\text{si } \text{fst}(N) = \text{fst}(N') \text{ et } \text{snd}(N) = \text{snd}(N')$$
$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P\{\mathsf{diff}[M, M']/x\} \,\} \quad \text{(Red Destr 1)}$$
$$\text{si } g(\text{fst}(M_1), \dots, \text{fst}(M_n)) \rightarrow M \text{ et } g(\text{snd}(M_1), \dots, \text{snd}(M_n)) \rightarrow M'$$
$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} \quad \text{(Red Destr 2)}$$
$$\text{s'il n'existe aucun } M \text{ tel que } g(\text{fst}(M_1), \dots, \text{fst}(M_n)) \rightarrow M$$
$$\text{et il n'existe aucun } M' \text{ tel que } g(\text{snd}(M_1), \dots, \text{snd}(M_n)) \rightarrow M'$$

Par cette sémantique, un biprocessus P se réduit quand ses deux composantes $\text{fst}(P)$ et $\text{snd}(P)$ se réduisent de la même façon : une communication est exécutée quand le canal est le même pour les deux composantes ; une application de destructeur réussit (resp. échoue) quand elle réussit (resp. échoue) pour les deux composantes.

Quand les deux composantes ne se réduisent pas de la même façon, on dit que la configuration $\mathcal{C} = E, \mathcal{P}$ *diverge*, et on note $\mathcal{C} \uparrow$ (vocabulaire et notation de [Bau07]) :

$$E, \mathcal{P} \cup \{\, \overline{N}\langle M\rangle.Q, N'(x).P \,\} \uparrow \quad \text{(Div I/O)}$$
$$\text{si } (\text{fst}(N) = \text{fst}(N')) \not\Leftrightarrow (\text{snd}(N) = \text{snd}(N'))$$
$$E, \mathcal{P} \cup \{\, \mathsf{let}\ x = g(M_1, \dots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \,\} \uparrow \quad \text{(Div Destr)}$$
$$\text{si } (\exists M, g(\text{fst}(M_1), \dots, \text{fst}(M_n)) \rightarrow M) \not\Leftrightarrow (\exists M', g(\text{snd}(M_1), \dots, \text{snd}(M_n)) \rightarrow M')$$

Si aucune configuration accessible ne diverge, alors les deux composantes du biprocessus P considéré se réduisent toujours de la même façon. Dans ce cas, le biprocessus satisfait l'équivalence, comme le montre le théorème suivant :

Théorème 2.5 *Soit P un biprocessus clos. Si, pour tout Init-attaquant Q, il n'existe aucune configuration $\mathcal{C}$ telle que $Init \cup \mathrm{fn}(P), \{P, Q\} \rightarrow^* \mathcal{C} \uparrow$, alors P satisfait la Init-équivalence.*

Grâce au théorème 2.5, il suffit de prouver une propriété de trace sur les biprocessus pour obtenir l'équivalence. Cette condition n'est cependant pas nécessaire : par exemple, si $P \approx_{Init} P'$, le biprocessus $\mathsf{if}\ \mathsf{diff}[\mathrm{true}, \mathrm{false}] = \mathrm{true}\ \mathsf{then}\ P\ \mathsf{else}\ P'$ satisfait la *Init*-équivalence, mais le théorème 2.5 ne nous permet pas de le prouver (car ce biprocessus diverge immédiatement). La condition sur les traces des biprocessus est codée en clauses de Horn comme précédemment, mais pour représenter la sémantique des biprocessus on utilise, à place de $\mathrm{attacker}(p)$ et $\mathrm{message}(p, p')$, des faits $\mathrm{attacker}'(p_1, p_2)$ et $\mathrm{message}'(p_1, p'_1, p_2, p'_2)$, où les composantes d'indice 1 correspondent à $\mathrm{fst}(P)$ et celles d'indice 2 à $\mathrm{snd}(P)$. Le fait $\mathrm{attacker}'(p_1, p_2)$ signifie que, par les mêmes actions, l'attaquant obtient p_1 en interaction avec $\mathrm{fst}(P)$ et p_2 en interaction avec $\mathrm{snd}(P)$. Le fait $\mathrm{message}'(p_1, p'_1, p_2, p'_2)$ signifie que, par les mêmes actions, $\mathrm{fst}(P)$ envoie le message p'_1 sur le canal p_1 tandis que $\mathrm{snd}(P)$ envoie le message p'_2 sur le canal p_2. On utilise également le fait $\mathrm{input}'(p_1, p_2)$ pour exprimer qu'une réception sur le canal p_1 peut-être exécutée par $\mathrm{fst}(P)$ tandis que $\mathrm{snd}(P)$ exécute une réception sur p_2 (ce qui permet à l'attaquant de tester l'égalité de ces canaux avec ceux utilisés dans une émission). Le prédicat nounif, déjà mentionné à la section 2.2.2, permet d'exprimer l'échec d'une application de destructeur. On peut ainsi coder en clauses la propriété de trace souhaitée sur les biprocessus, et la prouver par résolution, ce qui permet d'appliquer le théorème 2.5.

Dans sa thèse [Bau07], Mathieu Baudet a étudié cette méthode de preuve d'équivalence de processus. Il a en particulier montré, dans un cadre similaire au nôtre, la décidabilité de l'hypothèse du théorème 2.5 pour les processus sans réplication.

Une application importante de ces preuves d'équivalences est l'étude des protocoles qui utilisent des secrets faibles, comme des mots de passe. Ces protocoles sont sujets à des attaques par devinette, dans lesquelles l'attaquant devine le mot de passe (par exemple en essayant tous les mots d'un dictionnaire), puis vérifie qu'il a correctement deviné. Cette vérification peut être effectuée soit en ligne, en interagissant avec les participants du protocole, ce qu'on empêche simplement en limitant le nombre d'essais autorisés, soit hors ligne, en calculant sur les messages interceptés, sans interaction avec les autres participants.

On peut modéliser les attaques par devinette hors-ligne en combinant la notion d'équivalence observationnelle, les scénarios à plusieurs phases (section 2.2.5) et les primitives définies par des équations (section 2.1.5), car pour se protéger contre ces attaques il est souvent nécessaire qu'on ne puisse pas détecter l'échec éventuel du déchiffrement.

Dans la phase 0, l'attaquant peut interagir avec le protocole, mais le secret faible w est considéré comme impossible à deviner. Dans la phase 1, l'attaquant devine une valeur du secret faible. L'absence d'attaques par devinette hors ligne est caractérisée par une équivalence : l'attaquant ne peut pas distinguer le secret faible w utilisé dans la phase 0 d'une valeur fraîche w'.

Définition 2.8 Soit P un processus clos sans préfixe de phase et *Init* un ensemble de noms représentant la connaissance initiale de l'attaquant. On dit que P empêche les attaques par devinette hors ligne contre w si $(\nu w)(0 : P \mid 1 : (\nu w')\overline{c}\langle \mathsf{diff}[w, w'] \rangle)$ satisfait la *Init*-équivalence.

Nous avons prouvé à l'aide de ProVerif que les protocoles EKE [BM92] et Augmented EKE [BM93] satisfont cette propriété. Cette définition est dans la lignée des travaux de Cohen, Corin et al., Delaune et Jacquemard, Drieslma et al., et Lowe [Low02, Coh02, CMAFE03, DJ04, CDE04, DMV05]. Lowe [Low02] utilise le vérificateur de modèles FDR pour traiter un nombre borné de sessions. Delaune et Jacquemard [DJ04] donnent une procédure de décision pour ce

cas. Corin et al. [CDE04] donnent une définition fondée sur une équivalence comme la nôtre mais ne considèrent pas la première phase active et analysent une seule session.

2.3 Résultats

Le vérificateur ProVerif est disponible sur Internet à l'adresse `http://www.proverif.ens.fr/`. Il a été appliqué avec succès à de nombreux protocoles de la littérature, pour prouver des propriétés de secret et d'authentification : versions erronées et corrigées des protocoles de Needham-Schroeder à clé publique [NS78, Low96] et à clé partagée [NS78, BAN89, NS87], Woo-Lam à clé publique [WL92, WL97] et à clé partagée [WL92, AN95, AN96, WL97, GJ03], Denning-Sacco [DS81, AN96], Yahalom [BAN89], Otway-Rees [OR87, AN96, Pau98], Skeme [Kra96]. Les seuls cas de non-terminaison concernent certaines versions erronées du protocole de Woo-Lam à clé partagée. Les autres protocoles ont été vérifiés chacun en moins d'une seconde sur un Pentium M 1.8 GHz [Bla08a].

Les propriétés d'équivalences ont également été utilisées pour prouver le secret fort dans la version corrigée du protocole de Needham-Schroeder à clé publique [Low96] et les protocoles Otway-Rees [OR87], Yahalom [BAN89] et Skeme [Kra96], la sécurité des protocoles à mots de passe EKE [BM92] et Augmented EKE [BM93], l'authenticité du protocole Wide-Mouth-Frog [AG99] (version pour une session) [Bla04a, BAF08]. Le temps d'exécution va de moins d'une seconde à 15 s sur ces tests, sur un Pentium M 1.8 GHz.

De plus, il a aussi été utilisé dans des études de cas plus substantielles :

- En collaboration avec Martín Abadi [AB05b], nous l'avons appliqué à la vérification d'un protocole de courrier électronique certifié [AGHP02]. Nous utilisons les propriétés de correspondances pour montrer que le récepteur reçoit le message si et seulement si l'émetteur a un accusé de réception. (Nous utilisons des arguments manuels simples pour prendre en compte le fait que l'arrivée des messages envoyés est garantie.) Une des versions testées inclut la couche de transport du protocole SSH pour établir un canal sécurisé. (Temps total d'exécution : 6 min sur un Pentium M 1.8 GHz.)
- En collaboration avec Martín Abadi et Cédric Fournet [ABF07], nous avons étudié le protocole JFK (*Just Fast Keying*) [ABB+04], qui était un des candidats au remplacement d'IKE comme protocole d'échange de clés dans IPSec. Nous combinons à la fois des preuves manuelles et l'utilisation de ProVerif pour prouver des correspondances et des équivalences. (Temps total d'exécution : 3 min sur un Pentium M 1.8 GHz.)
- En collaboration avec Avik Chaudhuri [BC08], nous avons étudié le système de fichiers sécurisé Plutus [KRS+03] à l'aide de ProVerif, ce qui nous a permis de découvrir et corriger certaines faiblesses de l'article initial.

D'autres auteurs ont également utilisé ProVerif pour vérifier des protocoles ou construire d'autres outils :

- Karthik Bhargavan et al. [BFGP03, BFG04, BCFG04] l'ont utilisé pour construire l'outil de vérification de services Web TulaFale : les services Web sont des protocoles qui transmettent des messages XML ; TulaFale traduit ces protocoles dans le format d'entrée de ProVerif, et utilise ProVerif pour prouver les propriétés de sécurité souhaitées.
- Karthik Bhargavan et al. [BFGT06, BFG06, BFGS08] utilisent ProVerif pour vérifier des implantations de protocoles dans le langage F# (un langage fonctionnel de l'environnement .NET de Microsoft) : un sous-ensemble de F# suffisant pour exprimer des protocoles cryptographiques est traduit dans le format d'entrée de ProVerif.
- Kevin Lux et al. [LMBG05] ont conçu un service Web de courrier électronique certifié, et ont vérifié le protocole en utilisant TulaFale.
- Steve Kremer et Mark Ryan [KR04] déterminent en utilisant ProVerif si un protocole permet de construire une attaque à clair connu ou une attaque à clair ou chiffré choisi contre une primitive de chiffrement.

- Steve Kremer et Mark Ryan [KR05] l'ont également utilisé pour vérifier un protocole de vote électronique.
- Ran Canetti et Jonathan Herzog [CH06] l'utilisent pour prouver des protocoles dans le modèle calculatoire : ils montrent que pour une classe restreinte de protocoles qui utilisent seulement le chiffrement à clé publique, une preuve dans le modèle de Dolev-Yao implique la sécurité dans le modèle calculatoire, dans le cadre de la composabilité universelle. L'authentification est vérifiée par des propriétés de correspondances, alors que le secret des clés correspond au secret fort.
- Himanshu Khurana et Hyung-Seok Hahm [KH06] ont proposé un nouveau protocole pour des listes de discussion certifiées et l'ont vérifié avec ProVerif.
- Jens Chr. Godskesen [God06] a vérifié le protocole de routage pour réseaux ad-hoc ARAN (*Authenticated Routing for Adhoc Networks*).
- Michael Backes, Matteo Maffei et Dominique Unruh [BMU08] modélisent les protocoles *zero-knowledge* dans le pi calcul appliqué, et utilisent ProVerif pour vérifier le protocole DAA (*Direct Anonymous Attestation*). Ils ont montré la correction de cette modélisation vis-à-vis du modèle calculatoire [BU08].
- Michael Backes, Catalin Hritcu et Matteo Maffei [BHM08] formalisent les principales propriétés des protocoles de vote électronique (que les votes ne peuvent pas être modifiés, que seuls les inscrits peuvent voter, et seulement une fois, et la résistance aux coercitions) de façon à faciliter leur vérification automatique. Ils utilisent alors ProVerif pour les vérifier.

2.4 Conclusion

Le vérificateur automatique de protocoles ProVerif a été conçu pour fournir un bon compromis entre précision et efficacité, en garantissant la correction des propriétés de sécurité prouvées vis-à-vis du modèle de protocoles considéré, le modèle de Dolev-Yao. Il effectue des approximations, nécessaires pour pouvoir traiter un nombre non-borné de sessions, mais il reste extrêmement précis : il donne une analyse relationnelle, dans laquelle la principale approximation est l'oubli du nombre de répétitions de chaque action. Il permet de traiter des primitives cryptographiques variées, définies par des règles de réécriture ou des équations (même si certaines théories équationnelles comme celle du ou exclusif ne peuvent pas être traitées) et de prouver des propriétés de sécurité variées (secret, correspondances, certaines équivalences).

Ses principales limitations sont qu'il ne termine pas toujours (même s'il termine sur la grande classe des protocoles étiquetés et s'il termine la plupart du temps en pratique) et qu'il se fonde sur le modèle de Dolev-Yao, moins réaliste que le modèle calculatoire. Le vérificateur CryptoVerif, décrit au chapitre suivant, résout ce dernier problème en fournissant des preuves valides dans le modèle calculatoire, mais il est à un stade de développement moins avancé que ProVerif.

Chapitre 3

Vérification des protocoles dans le modèle calculatoire

Sommaire

3.1 Langage de représentation des jeux **40**
3.2 Équivalence observationnelle **44**
3.3 Transformations de jeux **44**
3.3.1 Transformations syntaxiques 44
3.3.2 Utiliser les hypothèses de sécurité sur les primitives 45
3.4 Propriétés de sécurité **49**
3.4.1 Secret 49
3.4.2 Correspondances 50
3.5 Stratégie de preuve **52**
3.6 Résultats **53**
3.7 Conclusion **54**

Ce chapitre présente le vérificateur de protocoles CryptoVerif. La principale originalité de ce vérificateur est qu'il produit directement des preuves valides dans le modèle calculatoire. Les preuves produites sont des preuves par suites de jeux, comme celles utilisées d'habitude par les cryptographes dans des preuves manuelles. Comme indiqué dans la section 1.6, une preuve par jeux consiste en une suite de jeux dont le premier correspond au protocole à prouver. Chaque autre jeu est obtenu à partir du précédent par des transformations telles que l'attaquant a une probabilité négligeable de distinguer deux jeux consécutifs. Dans le dernier jeu, l'attaquant a une probabilité négligeable de casser la propriété de sécurité à prouver, de par la forme même du jeu, sans faire intervenir d'hypothèse cryptographique. On en déduit alors que l'attaquant a une probabilité négligeable de casser la propriété souhaitée dans le jeu initial.

Les jeux sont formalisés dans un calcul de processus probabiliste polynomial, conçu pour faciliter les preuves automatiques. Contrairement aux calculs de processus utilisés dans le modèle formel, ce calcul ne fournit aucun choix non-déterministe, ce qui est important pour ne pas donner à l'attaquant la possibilité de deviner immédiatement les secrets. CryptoVerif fournit une méthode générique permettant de spécifier les hypothèses de sécurité de beaucoup de primitives cryptographiques, dont chiffrement à clé publique et à clé partagée, signatures, codes d'authentification de messages, fonctions de hachage. Il produit des preuves valides pour un nombre de sessions polynomial dans le paramètre de sécurité, en présence d'un attaquant actif. Il peut également évaluer la probabilité de succès d'une attaque en fonction de la probabilité de casser chaque primitive cryptographique et du nombre de sessions. Il peut prouver des propriétés de secret et de correspondances. (Ces dernières permettent de vérifier l'authentification, comme dans le modèle de Dolev-Yao.)

$M, N ::=$	termes
i	indice de réplication
$x[M_1, \ldots, M_m]$	accès à une variable
$f(M_1, \ldots, M_m)$	application de fonction
$Q ::=$	processus d'entrée
0	processus nul
$Q \mid Q'$	composition parallèle
$!^{i \leq n} Q$	réplication n fois
$\mathsf{newChannel}\ c; Q$	restriction de canal
$c[M_1, \ldots, M_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$	réception
$P ::=$	processus de sortie
$\overline{c[M_1, \ldots, M_l]}\langle N_1, \ldots, N_k \rangle; Q$	émission
$\mathsf{new}\ x[i_1, \ldots, i_m] : T; P$	nombre aléatoire
$\mathsf{let}\ x[i_1, \ldots, i_m] : T = M\ \mathsf{in}\ P$	affectation
$\mathsf{if}\ \mathsf{defined}(M_1, \ldots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditionnelle
$\mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat}$ $\mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$	recherche dans un tableau

FIG. 3.1 – Syntaxe du calcul de processus

Nous introduisons tout d'abord le langage utilisé pour représenter les protocoles et les différents jeux des preuves cryptographiques. Puis nous formalisons la notion d'indistinguabilité entre deux jeux, nous décrivons les transformations de jeux utilisées par CryptoVerif et les propriétés de sécurité prouvées sur le dernier jeu. Nous résumons la stratégie utilisée pour organiser les différentes transformations de jeux et les résultats obtenus sur des exemples de protocoles.

3.1 Langage de représentation des jeux

La syntaxe de ce langage est donnée dans la figure 3.1. Ce langage a été inspiré par le pi calcul et les calculs de processus de [LMMS98, LMMS99, MRST06] et de [Lau05]. Pour plus de simplicité, nous illustrons ce langage sur un exemple très simple de protocole, et renvoyons le lecteur à l'article [Bla08b] pour une présentation formelle.

On note η le paramètre de sécurité, qui détermine en particulier la longueur des clés. Ce langage utilise des types T, qui correspondent pour chaque valeur du paramètre de sécurité à un ensemble de chaînes de bits $I_\eta(T)$. Un type T est dit *de longueur fixée* quand $I_\eta(T)$ est l'ensemble de toutes les chaînes de bits d'une certaine longueur (qui peut dépendre de η). Un type T est dit *grand* quand $\frac{1}{I_\eta(T)}$ est négligeable. ($f(\eta)$ est négligeable quand pour tout polynôme q, il existe $\eta_0 \in \mathbb{N}$ tel que pour tout $\eta \geq \eta_0$, $f(\eta) \leq \frac{1}{q(\eta)}$.) On utilise les types *bool*, tel que $I_\eta(bool) = \{\text{true}, \text{false}\}$, où false $= 0$ et true $= 1$; *bitstring*, tel que $I_\eta(bitstring)$ est l'ensemble de toutes les chaînes de bits et $bitstring_\perp$ tel que $I_\eta(bitstring_\perp)$ est un symbole spécial $\perp$ union l'ensemble des chaînes de bits.

Les primitives cryptographiques et autres fonctions mathématiques sont modélisées par des symboles de fonction f. Chaque symbole de fonction f est muni d'une déclaration de type $f : T_1 \times \ldots \times T_m \to T$. Pour chaque valeur de η, f correspond à une fonction $I_\eta(f)$ de $I_\eta(T) \times \ldots \times I_\eta(T_m)$ vers $I_\eta(T)$, calculable en temps polynomial dans la longueur de ses arguments et la valeur de η. Par exemple, on représente les codes d'authentification de messages et le chiffrement symétrique par les fonctions définies ci-dessous. Ces définitions utilisent les notations

suivantes, habituelles en cryptographie. Si S est un ensemble fini, $x \stackrel{R}{\leftarrow} S$ choisit un élément aléatoire uniformément dans S et le stocke dans x. Si $\mathcal{A}$ est un algorithme probabiliste, $x \leftarrow \mathcal{A}(x_1, \dots, x_m)$ dénote l'expérience qui choisit un aléa r et stocke dans x le résultat de l'exécution de $\mathcal{A}(x_1, \dots, x_m)$ avec l'aléa r. Sinon, $x \leftarrow M$ est une simple instruction d'affectation.

Définition 3.1 Soient T_{mr}, T_{mk} et T_{ms} des types qui correspondent respectivement à des nombres aléatoires, des clés et des codes d'authentification de messages ; T_{mr} est un type de longueur fixée. Un code d'authentification de messages [BKR00] est constitué de trois symboles de fonction :

- mkgen : $T_{mr} \rightarrow T_{mk}$ où $I_\eta(\mathrm{mkgen}) = \mathrm{mkgen}_\eta$ est l'algorithme de génération de clés qui prend en argument une chaîne de bits aléatoire et retourne une clé. (Habituellement, mkgen est probabiliste ; ici, on sépare le choix des nombres aléatoires des calculs, donc mkgen prend un argument supplémentaire qui représente l'aléa.)
- mac : *bitstring* $\times T_{mk} \rightarrow T_{ms}$ où $I_\eta(\mathrm{mac}) = \mathrm{mac}_\eta$ est l'algorithme de MAC (*message authentication code*, code d'authentification de message) qui prend en argument un message et une clé et retourne le MAC correspondant. (On suppose ici que mac est déterministe ; on pourrait facilement coder un mac probabiliste en ajoutant un argument supplémentaire comme pour mkgen.)
- check : *bitstring* $\times T_{mk} \times T_{ms} \rightarrow$ *bool* où $I_\eta(\mathrm{check}) = \mathrm{check}_\eta$ est l'algorithme de vérification tel que $\mathrm{check}_\eta(m, k, t) = \mathrm{true}$ si et seulement si t est un MAC valide du message m sous la clé k. (Comme mac est déterministe, $\mathrm{check}_\eta(m, k, t)$ est typiquement $\mathrm{mac}_\eta(m, k) = t$.)

On a $\forall m \in I_\eta(\mathit{bitstring}), \forall r \in I_\eta(T_{mr}), \mathrm{check}_\eta(m, \mathrm{mkgen}_\eta(r), \mathrm{mac}_\eta(m, \mathrm{mkgen}_\eta(r))) = \mathrm{true}$.

Un MAC est UF-CMA (*unforgeable under chosen message attacks*, inforgeable sous des attaques à messages choisis) si et seulement si, pour tout polynôme q,

$$\max_{\mathcal{A}} \Pr\left[r \stackrel{R}{\leftarrow} I_\eta(T_{mr}); k \leftarrow \mathrm{mkgen}_\eta(r); (m, t) \leftarrow \mathcal{A}^{\mathrm{mac}_\eta(.,k), \mathrm{check}_\eta(.,k,.)} : \mathrm{check}_\eta(m, k, t)\right]$$

est négligeable, où l'adversaire $\mathcal{A}$ est n'importe quelle machine de Turing probabiliste, qui s'exécute en temps $q(\eta)$, avec accès aux oracles $\mathrm{mac}_\eta(., k)$ et $\mathrm{check}_\eta(., k, .)$, et $\mathcal{A}$ n'a pas appelé l'oracle $\mathrm{mac}_\eta(., k)$ sur le message m.

Intuitivement, quand le MAC est UF-CMA, l'attaquant a une probabilité négligeable de forger un MAC quand il n'a pas la clé k. On représente le chiffrement symétrique de façon analogue.

Définition 3.2 Soient T_r et T'_r des types à longueur fixée ; soient T_k et T_e des types. Un schéma de chiffrement symétrique [BDJR97] est constitué de trois symboles de fonction kgen : $T_r \rightarrow T_k$, enc : *bitstring* $\times T_k \times T'_r \rightarrow T_e$, et dec : $T_e \times T_k \rightarrow \mathit{bitstring}_\perp$, avec $I_\eta(\mathrm{kgen}) = \mathrm{kgen}_\eta$, $I_\eta(\mathrm{enc}) = \mathrm{enc}_\eta$, $I_\eta(\mathrm{dec}) = \mathrm{dec}_\eta$, tels que pour tout $m \in I_\eta(\mathit{bitstring})$, $r \in I_\eta(T_r)$, et $r' \in I_\eta(T'_r)$, $\mathrm{dec}_\eta(\mathrm{enc}_\eta(m, \mathrm{kgen}_\eta(r), r'), \mathrm{kgen}_\eta(r)) = m$.

Soit $LR(x, y, b) = x$ si $b = 0$ et $LR(x, y, b) = y$ si $b = 1$, défini seulement si x et y sont des chaînes de bits de même longueur. Un schéma de chiffrement symétrique est IND-CPA (*indistinguishable under chosen plaintext attacks*, indistinguable sous des attaques à clairs choisis) si et seulement si pour tout polynôme q,

$$\max_{\mathcal{A}} 2 \Pr\left[b \stackrel{R}{\leftarrow} \{0, 1\}; r \stackrel{R}{\leftarrow} I_\eta(T_r); k \leftarrow \mathrm{kgen}_\eta(r); b' \leftarrow \mathcal{A}^{r' \stackrel{R}{\leftarrow} I_\eta(T'_r); \mathrm{enc}_\eta(LR(.,.,b), k, r')} : b' = b\right] - 1$$

est négligeable, où l'adversaire $\mathcal{A}$ est n'importe quelle machine de Turing probabiliste, qui s'exécute en temps $q(\eta)$, avec accès à l'oracle de chiffrement gauche-droit qui, étant donné deux chaînes de bits a_0 et a_1 de même longueur, retourne $r' \stackrel{R}{\leftarrow} I_\eta(T'_r); \mathrm{enc}_\eta(LR(a_0, a_1, b), k, r')$, c'est-à-dire chiffre a_0 si $b = 0$ et a_1 si $b = 1$.

Intuitivement, quand le schéma de chiffrement est IND-CPA, l'attaquant a une probabilité négligeable de distinguer si on a chiffré a_0 ou a_1, quand il n'a pas la clé k.

A l'aide des primitives de MAC et de chiffrement, on peut construire le protocole très simple suivant :

$$A \to B : e, \mathrm{mac}(e, x_{mk}) \quad \text{où } e = \mathrm{enc}(x'_k, x_k, x''_r) \text{ et } x''_r, x'_k \text{ sont des nombres aléatoires frais}$$

A et B sont supposés partager une clé de chiffrement symétrique x_k et une clé de MAC x_{mk}. A crée une clé fraîche x'_k et l'envoie à B chiffrée sous x_k. Un MAC est ajouté au message, pour garantir son intégrité. Le but du protocole est que x'_k soit une clé secrète partagée entre A et B.

Ce protocole peut être modélisé dans notre langage par le processus Q_0 suivant :

$$\begin{aligned}
Q_0 &= \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in} \\
&\quad \mathsf{new}\ x'_r : T_{mr}; \mathsf{let}\ x_{mk} : T_{mk} = \mathrm{mkgen}(x'_r)\ \mathsf{in}\ \overline{c}\langle\rangle; (Q_A \mid Q_B) \\
Q_A &= !^{i \leq n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x''_r : T'_r; \\
&\quad \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x''_r)\ \mathsf{in}\ \overline{c_A[i]}\langle x_m, \mathrm{mac}(x_m, x_{mk})\rangle \\
Q_B &= !^{i' \leq n} c_B[i'](x'_m, x_{ma}); \mathsf{if}\ \mathrm{check}(x'_m, x_{mk}, x_{ma})\ \mathsf{then} \\
&\quad \mathsf{let}\ \mathrm{i}_\bot(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle
\end{aligned}$$

Le processus Q_0 est supposé s'exécuter en présence qu'un attaquant, qui modélise également le réseau. Q_0 attend tout d'abord de recevoir un message envoyé par l'attaquant sur le canal *start*, par la réception $\mathit{start}()$. Il choisit ensuite un nombre aléatoire x_r uniformément distribué dans T_r, par la construction $\mathsf{new}\ x_r : T_r$. Il calcule alors la clé de chiffrement x_k à partir de ce nombre aléatoire par l'algorithme de génération de clés kgen. La clé de MAC x_{mk} est choisie de façon similaire. Ensuite, Q_0 émet un message vide sur le canal c ; après avoir envoyé ce message, le contrôle passe au processus qui reçoit le message, qui fait partie de l'attaquant.

Plusieurs processus sont alors disponibles, définis par $Q_A \mid Q_B$; ces processus représentent les rôles de A et B dans le protocole. Le processus $Q_A \mid Q_B$ est la composition parallèle de Q_A et Q_B ; il met à disposition simultanément les processus définis dans Q_A et dans Q_B. Le calcul étant purement probabiliste, cette composition parallèle diffère de celle des calculs de processus utilisés dans le modèle formel : il n'y a pas de préemption entre processus ; le contrôle ne change de processus qu'au moment des communications, où il passe du processus émetteur au processus récepteur du message. Soient Q'_A et Q'_B tels que $Q_A = !^{i \leq n} Q'_A$ et $Q_B = !^{i' \leq n} Q'_B$. La réplication $!^{i \leq n} Q'_A$ représente n copies du processus Q'_A, indicées par l'indice de réplication i. (Le symbole n correspond à un entier $I_\eta(n)$ pour chaque valeur du paramètre de sécurité η ; $I_\eta(n)$ est polynomial en η.) Le processus Q'_A commence par une réception sur le canal $c_A[i]$: le canal est indicé par i pour que l'attaquant puisse choisir à quelle copie de Q'_A il envoie le message. La situation est similaire pour Q'_B. L'attaquant peut alors exécuter chaque copie de Q'_A ou Q'_B en envoyant un message sur le canal approprié $c_A[i]$ ou $c_B[i']$.

Après avoir reçu un message sur $c_A[i]$, Q'_A choisit aléatoirement une clé fraîche x'_k et des jetons aléatoires x''_r utilisés dans l'algorithme de chiffrement. Il chiffre alors la clé x'_k sous la clé x_k, et stocke le résultat dans x_m. La variable x'_k est de type T_k, le type des clés, alors que l'algorithme de chiffrement attend une chaîne de bits quelconque, de type *bitstring*. On utilise alors une fonction k2b pour convertir une donnée de type T_k en une donnée de type *bitstring* : k2b est l'injection naturelle de T_k dans *bitstring*. Elle est *poly-injective*, c'est-à-dire qu'elle est injective et que sa fonction réciproque peut être calculée en temps polynomial. Enfin, Q'_A envoie sur le canal $c_A[i]$ le message formé de x_m et de son MAC sous x_{mk}, comme spécifié dans le protocole. Le contrôle passe alors au processus qui reçoit ce message, qui fait partie de l'attaquant. Ce processus est censé faire suivre ce message à Q'_B sur le canal $c_B[i']$, mais il peut aussi agir différemment pour attaquer le protocole.

Le processus Q'_B attend le message x'_m, x_{ma} sur le canal $c_B[i']$. Quand il reçoit ce message, il vérifie que le MAC x_{ma} est un MAC correct de $x_{m'}$ avec la fonction check et, si oui, il déchiffre x'_m avec la clé x_k. Le déchiffrement retourne le symbole spécial $\perp$ quand il échoue, et une chaîne de bits de type *bitstring* quand il réussit. La fonction $\mathrm{i}_\perp$ est l'injection naturelle de *bitstring* vers *bitstring*$_\perp$, de sorte que, quand $\mathrm{i}_\perp(x) = \mathrm{dec}(x'_{m'}, x_k)$, le déchiffrement a réussi, et le clair est x. On traduit x de type *bitstring* vers le type T_k en utilisant la réciproque de k2b, quand x est bien de type T_k, c'est-à-dire quand x est la forme $\mathrm{k2b}(x''_k)$. Alors, x''_k contient normalement une clé x'_k choisie par A. On va chercher à montrer que cette clé x''_k reste secrète, c'est-à-dire que l'attaquant ne peut pas la distinguer d'une clé aléatoire. Le processus Q'_B conclut en envoyant un message vide sur $c_B[i']$, pour repasser le contrôle à l'attaquant.

Dans ce calcul, toutes les variables définies sous une réplication sont implicitement des tableaux. Par exemple, la variable x_m définie sous la réplication $!^{i \leq n}$ est implicitement un tableau indicé par i : x_m est une abréviation de $x_m[i]$. De même, x'_k, x''_r, x'_m, x_{ma}, x''_k sont respectivement des abréviations de $x'_k[i]$, $x''_r[i]$, $x'_m[i']$, $x_{ma}[i']$, $x''_k[i']$. L'utilisation de tableaux permet de mémoriser toutes les valeurs des variables dans les différentes copies des processus, ce qui permet de conserver en mémoire l'ensemble de l'état du système. Dans notre calcul, les tableaux remplacent les listes souvent utilisées par les cryptographes dans leurs preuves. Par exemple, dans la preuve, les messages dont on a calculé le MAC sous x_{mk} seraient stockés dans une liste, et l'inforgeabilité des MACs montrerait que si la vérification du MAC réussit, alors le message considéré est dans cette liste. Dans notre calcul, ces messages sont stockés dans le tableau x_m. Notre calcul comprend également une construction de recherche dans les tableaux : find $u_1 \leq n_1, \ldots, u_m \leq n_m$ suchthat defined$(M_1, \ldots, M_l) \wedge M$ then P else P' cherche des indices $u_1, \ldots, u_m$ tels que $M_1, \ldots, M_l$ sont définis et M est vrai. Quand de tels indices sont trouvés, P est exécuté, sinon P' est exécuté. Par exemple, find $u \leq n$ suchthat defined$(x_m[u]) \wedge x_m[u] = N$ then P cherche un indice u tel que $x_m[u]$ est défini et égal à N. Cette construction se généralise à des find à plusieurs branches. On note $\widetilde{i}$ un m-uplet $i_1, \ldots, i_m$. L'ordre et les indices de tableaux sur les n-uplets sont considérés composante par composante, donc par exemple $u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ sera éventuellement abrégé $\widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$. La construction find $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P cherche une branche $j \in [1, m]$ telle qu'il existe des valeurs de $u_{j1}, \ldots, u_{jm_j}$ pour lesquelles $M_{j1}, \ldots, M_{jl_j}$ sont définis et M_j est vrai. En cas de succès, P_j est exécuté. (S'il y a plusieurs choix, ils sont exécutés avec la même probabilité.) En cas d'échec pour toutes les branches, P est exécuté. Ici, la construction find n'apparaît pas dans le jeu initial, mais sera introduite par des transformations de jeux.

Enfin, notre calcul inclut une construction supplémentaire, la restriction newChannel $c; Q$ qui permet de restreindre la portée du canal c au processus Q ; le canal c est alors privé.

Comme détaillé dans [Bla08b], un processus doit être *bien formé* : cette condition implique que les chaînes de bits sont du type attendu et que les tableaux sont utilisés correctement (que chaque cellule d'un tableau est affectée au plus une fois, et que les variables sont lues seulement après avoir été initialisées). CryptoVerif vérifie que le processus initial est bien formé, et les transformations de jeux utilisées préservent la bonne formation.

La sémantique des processus est définie formellement par une relation de réduction probabiliste (voir [Bla08b, Annexe B]). Tous les processus de notre calcul s'exécutent en temps polynomial probabiliste. On note $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a \rangle]$ la probabilité que le processus Q émette la chaîne de bits a sur le canal c au cours de son exécution. Un contexte est un processus qui contient un trou $[\,]$. Un contexte d'évaluation C est un contexte formé à partir de $[\,]$, newChannel $c; C$, $Q \mid C$, et $C \mid Q$. On utilise un contexte d'évaluation pour représenter l'attaquant. On note $C[Q]$ le processus obtenu en remplaçant le trou $[\,]$ du contexte C par le processus Q. On note $\mathrm{var}(Q)$ l'ensemble des variables du processus Q. On utilise aussi la notation $\mathrm{var}(\cdot)$ pour des contextes et des termes. On note $\mathrm{fc}(Q)$ l'ensemble des canaux libres (non-restreints) de Q.

3.2 Équivalence observationnelle

Informellement, deux processus sont observationnellement équivalents quand l'attaquant a une probabilité négligeable de les distinguer. Notre définition d'équivalence observationnelle est adaptée à partir des définitions pour des calculs précédents comme [MRST06].

Définition 3.3 (Équivalence observationnelle) Soient Q et Q' deux processus et V un ensemble de variables. On suppose que Q et Q' sont bien formés et que les variables de V sont définies dans Q et Q', avec les mêmes types.

Un contexte d'évaluation C est dit *acceptable* pour Q, Q', V si et seulement si $\mathrm{var}(C) \cap (\mathrm{var}(Q) \cup \mathrm{var}(Q')) \subseteq V$ et $C[Q]$ est bien formé. (Alors $C[Q']$ l'est aussi.)

On dit que Q et Q' sont *observationnellement équivalents* avec les variables publiques V, et on note $Q \approx^V Q'$, si et seulement si, pour tout contexte d'évaluation C acceptable pour Q, Q', V, pour tout canal c, pour toute chaîne de bits a, $|\Pr[C[Q] \rightsquigarrow_\eta \overline{c}\langle a\rangle] - \Pr[C[Q'] \rightsquigarrow_\eta \overline{c}\langle a\rangle]|$ est négligeable.

Intuitivement, le but de l'attaquant représenté par le contexte C est de distinguer Q de Q'. Quand il réussit, il effectue une émission différente, par exemple $\overline{c}\langle 0\rangle$ quand il a reconnu Q et $\overline{c}\langle 1\rangle$ quand il a reconnu Q'. Quand $Q \approx^V Q'$, le contexte a une probabilité négligeable de distinguer Q de Q'.

La condition inhabituelle sur les variables de C vient de la présence des tableaux et de la construction associée find qui donne à C un accès direct aux variables de Q et Q' : le contexte C est autorisé à accéder aux variables de Q et Q' seulement quand elles sont dans V. Le résultat suivant est facile à prouver :

Lemme 3.1 *$\approx^V$ est une relation d'équivalence, et $Q \approx^V Q'$ implique $C[Q] \approx^{V'} C[Q']$ pour tout contexte d'évaluation C acceptable pour Q, Q', V et tout $V' \subseteq V \cup (\mathrm{var}(C) \setminus (\mathrm{var}(Q) \cup \mathrm{var}(Q')))$.*

On note $Q \approx_0^V Q'$ le cas particulier dans lequel, pour tout contexte d'évaluation C acceptable pour Q, Q', V, pour tout canal c, pour toute chaîne de bits a, $\Pr[C[Q] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C[Q'] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$. Quand V est vide, on écrit $Q \approx Q'$ au lieu de $Q \approx^V Q'$ et $Q \approx_0 Q'$ au lieu de $Q \approx_0^V Q'$.

A partir d'un processus Q_0 correspondant au protocole à prouver, CryptoVerif construit une suite de processus observationnellement équivalents $Q_0 \approx^V Q_1 \approx^V \ldots \approx^V Q_m$, grâce aux transformations de jeux résumées dans la section suivante. Par transitivité de $\approx^V$, $Q_0 \approx^V Q_m$, et donc en prouvant une propriété de sécurité sur Q_m, on peut en déduire que cette propriété reste vraie (à probabilité négligeable près) sur Q_0.

3.3 Transformations de jeux

Dans cette section, nous décrivons les transformations de jeux qui permettent de transformer le processus qui représente le protocole initial en un processus sur lequel la propriété de sécurité souhaitée peut être prouvée directement, par les critères donnés dans la section 3.4. Ces transformations sont paramétrées par l'ensemble V des variables auxquelles le contexte peut accéder. Ces transformations transforment un processus Q_0 en un processus Q'_0 tel que $Q_0 \approx^V Q'_0$.

3.3.1 Transformations syntaxiques

RemoveAssign(x) : suppression des affectations sur x. Quand x est défini par une affectation let $x[i_1, \ldots, i_l] : T = M$ in P, on remplace x par sa valeur M. Cette transformation n'est pas complètement évidente quand x est accédé par l'intermédiaire de find : si x a plusieurs définitions, on ne sait pas laquelle est concernée, donc le remplacement n'est pas possible pour ces accès à x. Il faut aussi veiller à respecter la sémantique des conditions defined, et préserver l'invariant que si on accède à une variable, alors on est sûr qu'elle est définie, soit parce que,

syntaxiquement, elle a été définie avant avec les mêmes indices, soit parce que l'accès est gardé par une condition defined adéquate.

Exemple 3.1 Dans le processus de la section 3.1, la transformation **RemoveAssign**(x_{mk}) remplace x_{mk} par $\text{mkgen}(x'_r)$ dans tout le processus, et supprime l'affectation let $x_{mk} : T_{mk} = \text{mkgen}(x'_r)$. Après ce remplacement, $\text{mac}(x_m, x_{mk})$ devient $\text{mac}(x_m, \text{mkgen}(x'_r))$ et $\text{check}(x'_m, x_{mk}, x_{ma})$ devient $\text{check}(x'_m, \text{mkgen}(x'_r), x_{ma})$, ce qui fait apparaître des termes requis dans la section 3.3.2. La situation est similaire pour **RemoveAssign**(x_k).

SArename(x) : renommage de x pour qu'il ait une seule affectation (*single assignment rename*). Cette transformation renomme les variables pour qu'elles aient une seule définition dans le jeu ; ceci est utile pour distinguer les cas suivant quelle définition de x a défini $x[\widetilde{i}]$. Cette transformation est appliquée seulement quand $x \notin V$. Quand x a $m > 1$ définitions, chaque définition de x est renommée en une variable différente $x_1, \ldots, x_m$. Pour les accès au tableau x, une recherche dans le tableau x est remplacée par des recherches dans les tableaux $x_1, \ldots, x_m$.

Simplify : simplification. La procédure de simplification des jeux est assez complexe. L'idée principale est de collecter toutes les égalités qui sont vraies à chaque point de programme, puis d'utiliser un prouveur équationnel fondé sur un algorithme proche de la complétion de Knuth-Bendix [KB70] pour déduire d'autres égalités et simplifier le jeu. Les égalités collectées comprennent :

- Des équations définies par l'utilisateur, de la forme $\forall x_1 : T_1, \ldots, \forall x_m : T_m, M$, qui signifient que M est vrai pour tout $x_1, \ldots, x_m$ de types $T_1, \ldots, T_m$. Par exemple, pour les schémas de MAC et de chiffrement des définitions 3.1 et 3.2, on a :

$$\forall r : T_{mr}, \forall m : \textit{bitstring}, \text{check}(m, \text{mkgen}(r), \text{mac}(m, \text{mkgen}(r))) = \text{true} \qquad \text{(mac)}$$
$$\forall m : \textit{bitstring}; \forall r : T_r, \forall r' : T'_r, \text{dec}(\text{enc}(m, \text{kgen}(r), r'), \text{kgen}(r)) = \text{i}_\perp(m) \qquad \text{(enc)}$$

  On exprime la poly-injectivité de la fonction k2b de l'exemple de la section 3.1 par

$$\forall x : T_k, \forall y : T_k, (\text{k2b}(x) = \text{k2b}(y)) = (x = y) \qquad \forall x : T_k, \text{k2b}^{-1}(\text{k2b}(x)) = x \qquad \text{(k2b)}$$

  où k2b^{-1} est un symbole de fonction qui représente l'inverse de k2b. On a des formules similaires pour $\text{i}_\perp$.
- Des équations qui viennent du processus. Par exemple, dans le processus if M then P else P', on a $M = \text{true}$ dans P et $M = \text{false}$ dans P'.
- La faible probabilité de collision entre nombres aléatoires. Par exemple, quand x est défini par new $x : T$ et T est un grand type, $x[M_1, \ldots, M_m] = x[M'_1, \ldots, M'_m]$ implique $M_1 = M'_1$, $\ldots$, $M_m = M'_m$ à probabilité négligeable près.

Le prouveur combine ces propriétés pour simplifier les termes, et utilise les termes simplifiés pour simplifier les processus. Par exemple, si M se simplifie en true, alors if M then P else P' se simplifie en P. De même, une branche de find est supprimée quand la condition associée se simplifie en false.

Des détails sur la procédure de simplification peuvent être trouvés dans [Bla08b, annexe C] et la preuve de la proposition suivante dans [Bla08b, annexe E.1].

Proposition 3.1 *Soit Q_0 un processus bien formé et Q'_0 le processus obtenu à partir de Q_0 par une des transformations ci-dessus. Alors Q'_0 est bien formé et $Q_0 \approx^V Q'_0$.*

3.3.2 Utiliser les hypothèses de sécurité sur les primitives

La sécurité des primitives cryptographiques est définie en utilisant des équivalences observationnelles données comme axiomes. Ce formalisme permet de spécifier de nombreuses primitives

de façon générique. Ces équivalences sont ensuite utilisées par le prouveur pour transformer un jeu en un autre jeu observationnellement équivalent, comme expliqué ci-dessous.

Les hypothèses sur les primitives cryptographiques sont spécifiées en utilisant des équivalences de la forme $(G_1, \ldots, G_m) \approx (G'_1, \ldots, G'_m)$ où G est défini par la grammaire suivante, avec $l \geq 0$ et $m \geq 1$:

$G ::=$	groupe de fonctions
$\quad !^{i \leq n}\mathsf{new}\ y_1 : T_1; \ldots; \mathsf{new}\ y_l : T_l; (G_1, \ldots, G_m)$	réplication, restrictions
$\quad (x_1 : T_1, \ldots, x_l : T_l) \rightarrow FP$	fonction

$FP ::=$	processus fonctionnel
$\quad M$	terme
$\quad \mathsf{new}\ x[\widetilde{i}] : T; FP$	nombre aléatoire
$\quad \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ FP$	affectation
$\quad \mathsf{find}\ (\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}\ \mathsf{suchthatdefined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ FP_j)\ \mathsf{else}\ FP$	recherche dans un tableau

Intuitivement, $(x_1 : T_1, \ldots, x_l : T_l) \rightarrow FP$ représente une fonction qui prend en argument des valeurs $x_1, \ldots, x_l$ de types $T_1, \ldots, T_l$ et retourne un résultat calculé par FP. L'équivalence observationnelle $(G_1, \ldots, G_m) \approx (G'_1, \ldots, G'_m)$ exprime que l'attaquant a une probabilité négligeable de distinguer les fonctions du membre gauche des fonctions correspondantes du membre droit. Formellement, les fonctions peuvent être encodées comme des processus qui reçoivent leurs arguments et renvoient leur résultat sur des canaux [Bla08b, section 3.2]. Pour plus de simplicité, on confond ici les fonctions et leur codage comme processus.

Par exemple, la sécurité d'un MAC (définition 3.1) est représentée par l'équivalence $L \approx R$ où

$$\begin{aligned}
L = {} & !^{i'' \leq n''}\mathsf{new}\ r : T_{mr}; (\\
& \quad !^{i \leq n}(x : \mathit{bitstring}) \rightarrow \mathrm{mac}(x, \mathrm{mkgen}(r)), \\
& \quad !^{i' \leq n'}(m : \mathit{bitstring}, ma : T_{ms}) \rightarrow \mathrm{check}(m, \mathrm{mkgen}(r), ma)) \\
R = {} & !^{i'' \leq n''}\mathsf{new}\ r : T_{mr}; (\\
& \quad !^{i \leq n}(x : \mathit{bitstring}) \rightarrow \mathrm{mac}'(x, \mathrm{mkgen}'(r)), \\
& \quad !^{i' \leq n'}(m : \mathit{bitstring}, ma : T_{ms}) \rightarrow \\
& \qquad \mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[u]) \wedge (m = x[u]) \\
& \qquad\quad \wedge\, \mathrm{check}'(m, \mathrm{mkgen}'(r), ma)\ \mathsf{then}\ \mathrm{true}\ \ \mathsf{else}\ \mathrm{false})
\end{aligned} \tag{mac$_{\mathrm{eq}}$}$$

où mac′, check′ et mkgen′ sont des symboles de fonction de mêmes types que mac, check et mkgen respectivement. (On utilise des symboles de fonction différents dans les membres gauche et droit uniquement pour empêcher l'application répétée de la transformation induite par cette équivalence : sinon, le membre droit serait une instance du membre gauche et la transformation pourrait être répétée indéfiniment. Comme on ajoute ces symboles de fonction, on ajoute aussi l'équation

$$\forall r : T_{mr}, \forall m : \mathit{bitstring}, \mathrm{check}'(m, \mathrm{mkgen}'(r), \mathrm{mac}'(m, \mathrm{mkgen}'(r))) = \mathrm{true} \tag{mac$'$}$$

qui répète (mac) pour mac′, check′ et mkgen′.) Intuitivement, l'équivalence $L \approx R$ laisse inchangés les calculs de MACs (si l'on ignore l'utilisation de symboles de fonction avec prime dans R), et permet de remplacer la vérification de MAC $\mathrm{check}(m, \mathrm{mkgen}(r), ma)$ par une recherche dans le tableau x des messages dont le MAC a été calculé avec la clé $\mathrm{mkgen}(r)$: si m est trouvé dans le tableau x et $\mathrm{check}(m, \mathrm{mkgen}(r), ma)$, on retourne true ; sinon, la vérification échoue (à probabilité négligeable près), donc on retourne false. (Si la vérification réussissait alors que m

n'est pas dans le tableau x, l'attaquant aurait forgé un MAC.) Bien sûr, la forme de L requiert que r soit utilisé seulement pour calculer ou vérifier des MACs, pour que l'équivalence soit correcte. Formellement, le résultat suivant montre la correction de notre modélisation. C'est une conséquence assez directe de la définition 3.1, et il est prouvé dans [Bla08b, annexe E.3].

Proposition 3.2 *Si* (mkgen, mac, check) *est un code d'authentification de messages UF-CMA,* $I_\eta(\mathrm{mkgen}') = I_\eta(\mathrm{mkgen})$*,* $I_\eta(\mathrm{mac}') = I_\eta(\mathrm{mac})$ *et* $I_\eta(\mathrm{check}') = I_\eta(\mathrm{check})$*, alors* $L \approx R$*.*

De façon analogue, si (kgen, enc, dec) est un schéma de chiffrement symétrique (définition 3.2), alors on a l'équivalence suivante :

$$\begin{aligned} &!^{i' \leq n'} \mathsf{new}\ r : T_r; !^{i \leq n}(x : \mathit{bitstring}) \rightarrow \mathsf{new}\ r' : T'_r; \mathrm{enc}(x, \mathrm{kgen}(r), r') \\ \approx\ &!^{i' \leq n'} \mathsf{new}\ r : T_r; !^{i \leq n}(x : \mathit{bitstring}) \rightarrow \mathsf{new}\ r' : T'_r; \mathrm{enc}'(\mathrm{Z}(x), \mathrm{kgen}'(r), r') \end{aligned} \qquad (\mathrm{enc_{eq}})$$

où enc' et kgen' sont des symboles de fonction de mêmes types que enc et kgen respectivement, et $\mathrm{Z} : \mathit{bitstring} \rightarrow \mathit{bitstring}$ est une fonction qui retourne un chaîne de bits contenant uniquement des zéros et de la même longueur que son argument. En utilisant des équations comme $\forall x : T, \mathrm{Z}(\mathrm{T2b}(x)) = \mathrm{Z}_T$, on peut prouver que $\mathrm{Z}(\mathrm{T2b}(x))$ ne dépend pas de x quand x est d'un type de longueur fixée et $\mathrm{T2b} : T \rightarrow \mathit{bitstring}$ est l'injection naturelle. L'équivalence $\mathrm{enc_{eq}}$ exprime intuitivement qu'on peut remplacer (à probabilité négligeable près) le chiffrement x par le chiffrement de $\mathrm{Z}(x)$, chaîne de bits de même longueur que x. La représentation d'autres primitives cryptographiques dans CryptoVerif peut-être trouvée dans [Bla08b, annexe D.3], ainsi que dans [BP06, BJST08]. Les équivalences qui formalisent les hypothèses de sécurité sur les primitives sont conçues et prouvées correctes manuellement à partir d'hypothèses de sécurité dans une forme plus standard, comme dans l'exemple du MAC. Ces preuves manuelles sont faites seulement une fois pour chaque primitive, et l'équivalence obtenue peut être réutilisée pour prouver de nombreux protocoles automatiquement.

Ces équivalences $L \approx R$ sont utilisées afin de transformer un processus Q_0 observationnellement équivalent à $C[L]$ en un processus Q'_0 observationnellement équivalent à $C[R]$ pour un certain contexte C. Pour détecter que $Q_0 \approx^V_0 C[L]$, CryptoVerif utilise des conditions syntaxiques suffisantes. Essentiellement, il s'agit de montrer que tous les usages des variables secrètes de L peuvent être codés comme des appels à des fonctions de L. Dans l'exemple du MAC, tous les usages de r doivent être codés comme appels aux fonctions (oracles) de MAC et de vérification. Formellement, ces conditions sont assez complexes, et détaillées dans [Bla08b, section 3.2 et annexe D.1]. Une fois cette vérification effectuée, le processus Q_0 est transformé en Q'_0, essentiellement en remplaçant les appels aux fonctions de L par des appels aux fonctions correspondantes de R. Comme $L \approx R$, par le lemme 3.1, on a $C[L] \approx^V C[R]$ donc on obtient $Q_0 \approx^V_0 C[L] \approx^V C[R] \approx^V_0 Q'_0$. Le résultat suivant est prouvé dans [Bla08b, annexe E.4] ; il montre la correction de la transformation.

Proposition 3.3 *Soit* Q_0 *un processus bien formé et* Q'_0 *le processus obtenu à partir de* Q_0 *par la transformation ci-dessus. Alors* Q'_0 *est bien formé et, si* $L \approx R$*, alors* $Q_0 \approx^V Q'_0$*.*

Exemple 3.2 Pour traiter l'exemple de la section 3.1, on donne à CryptoVerif en entrée l'indication que T_{mr}, T_r, T'_r et T_k sont des types à longueur fixée ; les déclarations de types pour les fonctions $\mathrm{mkgen}, \mathrm{mkgen}' : T_{mr} \rightarrow T_{mk}$, $\mathrm{mac}, \mathrm{mac}' : \mathit{bitstring} \times T_{mk} \rightarrow T_{ms}$, $\mathrm{check}, \mathrm{check}' : \mathit{bitstring} \times T_{mk} \times T_{ms} \rightarrow \mathit{bool}$, $\mathrm{kgen}, \mathrm{kgen}' : T_r \rightarrow T_k$, $\mathrm{enc}, \mathrm{enc}' : \mathit{bitstring} \times T_k \times T'_r \rightarrow T_e$, $\mathrm{dec} : T_e \times T_k \rightarrow \mathit{bitstring}_\perp$, $\mathrm{k2b} : T_k \rightarrow \mathit{bitstring}$, $\mathrm{i}_\perp : \mathit{bitstring} \rightarrow \mathit{bitstring}_\perp$, $\mathrm{Z} : \mathit{bitstring} \rightarrow \mathit{bitstring}$ et la constante $\mathrm{Z}_k : \mathit{bitstring}$; les équations (mac), (mac'), (2.1) et $\forall x : T_k, \mathrm{Z}(\mathrm{k2b}(x)) = \mathrm{Z}_k$ (qui exprime que toutes les clés ont la même longueur) ; l'indication que k2b et $\mathrm{i}_\perp$ sont poly-injectives (qui crée les équations (k2b) et des équations similaires pour $\mathrm{i}_\perp$) ; les équivalences $L \approx R$ pour le MAC ($\mathrm{mac_{eq}}$) et le chiffrement ($\mathrm{enc_{eq}}$) ; et le processus Q_0 de la section 3.1.

CryptoVerif applique tout d'abord **RemoveAssign**(x_{mk}) au processus Q_0, comme décrit dans l'exemple 3.1. Le processus est alors transformé en utilisant la sécurité du MAC. On obtient le processus Q'_0 suivant :

$$\begin{aligned}
&Q'_0 = \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in}\ \mathsf{new}\ x'_r : T_{mr}; \overline{c}\langle\rangle; (Q'_A \mid Q'_B)\\
&Q'_A = !^{i \le n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x''_r : T'_r; \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x''_r)\ \mathsf{in}\\
&\quad \overline{c_A[i]}\langle x_m, \mathrm{mac}'(x_m, \mathrm{mkgen}'(x'_r))\rangle\\
&Q'_B = !^{i' \le n} c_B[i'](x'_m, x_{ma});\\
&\quad \mathsf{find}\ u \le n\ \mathsf{suchthat}\ \mathsf{defined}(x_m[u]) \wedge x'_m = x_m[u] \wedge \mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})\ \mathsf{then}\\
&\quad\quad (\mathsf{if}\ \mathrm{true}\ \mathsf{then}\ \mathsf{let}\ \mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle)\\
&\quad \mathsf{else}\\
&\quad\quad (\mathsf{if}\ \mathrm{false}\ \mathsf{then}\ \mathsf{let}\ \mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle)
\end{aligned}$$

La définition initiale de x'_r est supprimée et remplacée par une nouvelle définition, qu'on appelle encore x'_r. Le terme $\mathrm{mac}(x_m, \mathrm{mkgen}(x'_r))$ est remplacé par $\mathrm{mac}'(x_m, \mathrm{mkgen}'(x'_r))$. Le terme $\mathrm{check}(x'_m, \mathrm{mkgen}(x'_r), x_{ma})$ devient $\mathsf{find}\ u \le n\ \mathsf{suchthat}\ \mathsf{defined}(x_m[u]) \wedge x'_m = x_m[u] \wedge \mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})\ \mathsf{then}\ \mathrm{true}\ \mathsf{else}\ \mathrm{false}$, ce qui donne Q'_B après transformation des fonctions en processus. Le processus cherche le message x'_m dans le tableau x_m, qui contient les messages dont le MAC a été calculé avec la clé $\mathrm{mkgen}(x'_r)$. Si le MAC de x'_m n'a jamais été calculé, la vérification échoue toujours (elle retourne false) par définition de la sécurité du MAC (à probabilité négligeable près). Sinon, elle retourne true quand $\mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})$.

Après application de **Simplify**, Q'_A reste inchangé et Q'_B devient

$$\begin{aligned}
&Q''_B = !^{i' \le n} c_B[i'](x'_m, x_{ma});\\
&\quad \mathsf{find}\ u \le n\ \mathsf{suchthat}\ \mathsf{defined}(x_m[u], x'_k[u]) \wedge x'_m = x_m[u] \wedge \mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})\ \mathsf{then}\\
&\quad \mathsf{let}\ x''_k : T_k = x'_k[u]\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle
\end{aligned}$$

Tout d'abord, les tests if true then ... et if false then ... sont simplifiés. Le terme $\mathrm{dec}(x'_m, x_k)$ est simplifié sachant que $x'_m = x_m[u]$ par la condition du find, $x_m[u] = \mathrm{enc}(\mathrm{k2b}(x'_k[u]), x_k, x''_r[u])$ par l'affectation qui définit x_m, $x_k = \mathrm{kgen}(x_r)$ par l'affectation qui définit x_k, et $\mathrm{dec}(\mathrm{enc}(m, \mathrm{kgen}(r), r'), \mathrm{kgen}(r)) = \mathrm{i}_\perp(m)$ par (2.1). Donc on a $\mathrm{dec}(x'_m, x_k) = \mathrm{i}_\perp(\mathrm{k2b}(x'_k[u]))$. Par injectivité de $\mathrm{i}_\perp$ et k2b, l'affectation à x''_k devient simplement $x''_k = x'_k[u]$, en utilisant les équations $\forall x : \mathit{bitstring}, \mathrm{i}_\perp^{-1}(\mathrm{i}_\perp(x)) = x$ et $\forall x : T_k, \mathrm{k2b}^{-1}(\mathrm{k2b}(x)) = x$.

Après avoir appliqué **RemoveAssign**(x_k), on applique la sécurité du chiffrement : le terme $\mathrm{enc}(\mathrm{k2b}(x'_k), \mathrm{kgen}(x_r), x''_r)$ devient $\mathrm{enc}'(\mathrm{Z}(\mathrm{k2b}(x'_k)), \mathrm{kgen}(x_r), x''_r)$. Après simplification, il devient $\mathrm{enc}'(\mathrm{Z}_k, \mathrm{kgen}(x_r), x''_r)$, en utilisant $\forall x : T_k, \mathrm{Z}(\mathrm{k2b}(x)) = \mathrm{Z}_k$ (qui exprime que toutes les clés ont la même longueur).

On obtient donc le jeu suivant :

$$\begin{aligned}
&Q''_0 = \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{new}\ x'_r : T_{mr}; \overline{c}\langle\rangle; (Q''_A \mid Q''_B)\\
&Q''_A = !^{i \le n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x''_r : T'_r; \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{Z}_k, \mathrm{kgen}(x_r), x''_r)\ \mathsf{in}\\
&\quad \overline{c_A[i]}\langle x_m, \mathrm{mac}'(x_m, \mathrm{mkgen}'(x'_r))\rangle
\end{aligned}$$

où Q''_B reste comme ci-dessus.

Utiliser les tableaux au lieu de listes simplifie cette transformation : on n'a pas besoin d'ajouter des instructions qui insèrent les valeurs dans la liste, car toutes les variables sont toujours implicitement dans des tableaux. De plus, s'il y a plusieurs occurrences de $\mathrm{mac}(x_i, k)$ avec la même clé dans le processus initial, chaque $\mathrm{check}(m_j, k, ma_j)$ est remplacé par un find avec une branche pour chaque occurrence de mac. De ce fait, CryptoVerif distingue automatiquement

les cas suivant l'occurrence de mac d'où vient le MAC vérifié ma_j, c'est-à-dire qu'il distingue les cas suivant la valeur de i telle que $m_j = x_i$. Typiquement, distinguer ces cas est utile dans les étapes suivantes de la preuve du protocole. (Une situation similaire se produit pour les autres primitives cryptographiques spécifiées en utilisant find.)

3.4 Propriétés de sécurité

Cette section définit les propriétés de secret et de correspondances (qui incluent l'authentification) et explique comment CryptoVerif les vérifie.

3.4.1 Secret

On définit deux notions de secret. La première exprime qu'une variable (ou chaque élément d'un tableau) est indistinguable d'un nombre aléatoire.

Définition 3.4 (Secret pour une session) Soit x une variable de type T définie dans Q sous les réplications $!^{i_1 \leq n_1} \ldots !^{i_m \leq n_m}$. Le processus Q *préserve le secret de x pour une session* si $Q \mid Q_x \approx Q \mid Q'_x$, où

$$Q_x = c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \mathsf{if\ defined}(x[u_1, \ldots, u_m]) \mathsf{\ then\ } \overline{c}\langle x[u_1, \ldots, u_m]\rangle$$
$$Q'_x = c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \mathsf{if\ defined}(x[u_1, \ldots, u_m]) \mathsf{\ then\ new\ } y : T; \overline{c}\langle y\rangle$$

$c \notin \mathrm{fc}(Q)$ et $u_1, \ldots, u_m, y \notin \mathrm{var}(Q)$.

Intuitivement, l'attaquant ne peut pas distinguer un processus qui émet la valeur du secret d'un processus qui émet un nombre aléatoire. L'attaquant exécute une seule requête de test, modélisée par Q_x et Q'_x.

Proposition 3.4 (Secret pour une session) *Soit Q un processus tel qu'il existe un ensemble de variables S tel que 1) les définitions de x sont soit des restrictions* new $x[\widetilde{i}] : T$ *et $x \in S$, soit des affectations* let $x[\widetilde{i}] : T = z[M_1, \ldots, M_l]$ *où z est défini par des restrictions* new $z[i'_1, \ldots, i'_l] : T$*, et $z \in S$, et 2) tous les accès à des variables $y \in S$ dans Q sont de la forme "*let $y'[\widetilde{i}] : T' = y[M_1, \ldots, M_l]$*" avec $y' \in S$. Alors $Q \mid Q_x \approx_0 Q \mid Q'_x$, donc Q préserve le secret de x pour une session.*

Intuitivement, seules les variables de S dépendent des restrictions qui définissent x ; les messages envoyés et le flot de contrôle du processus ne dépendent pas de x, donc l'attaquant n'a aucune information sur x.

Le secret à proprement parler exprime que les éléments d'un tableau sont indistinguables de nombres aléatoires indépendants.

Définition 3.5 (Secret) Soit x une variable de type T définie dans Q sous les réplications $!^{i_1 \leq n_1} \ldots !^{i_m \leq n_m}$. Le processus Q *préserve le secret de x* si $Q \mid R_x \approx Q \mid R'_x$, où

$$R_x = !^{i \leq n} c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \mathsf{if\ defined}(x[u_1, \ldots, u_m]) \mathsf{\ then\ } \overline{c}\langle x[u_1, \ldots, u_m]\rangle$$
$$R'_x = !^{i \leq n} c(u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \mathsf{if\ defined}(x[u_1, \ldots, u_m]) \mathsf{\ then}$$
$$\quad \mathsf{find\ } u' \leq n \mathsf{\ suchthat\ defined}(y[u'], u_1[u'], \ldots, u_m[u']) \wedge u_1[u'] = u_1 \wedge \ldots \wedge u_m[u'] = u_m$$
$$\quad \mathsf{then\ } \overline{c}\langle y[u']\rangle \mathsf{\ else\ new\ } y : T; \overline{c}\langle y\rangle$$

$c \notin \mathrm{fc}(Q)$ et $u_1, \ldots, u_m, u', y \notin \mathrm{var}(Q)$.

Intuitivement, l'attaquant ne peut pas distinguer un processus qui émet la valeur du secret pour plusieurs indices d'un processus qui émet des nombres aléatoires indépendants. Dans cette définition, l'attaquant peut exécuter plusieurs requêtes de test, modélisées par R_x et R'_x. Cela correspond à la définition de sécurité "réel ou aléatoire" (*real-or-random*) [AFP06]. (Cette définition est plus forte que la définition standard avec une seule requête de test et plusieurs requêtes qui révèlent toujours $x[u_1, \ldots, u_m]$ [AFP06].)

Informellement, CryptoVerif prouve le secret en montrant, en plus de l'hypothèse de la proposition 3.4, que chaque élément du tableau x vient d'une exécution distincte d'une restriction (et donc que les éléments de x sont des nombres aléatoires indépendants).

Lemme 3.2 *Si $Q \approx^{\{x\}} Q'$ et Q préserve le secret de x pour une session alors Q' préserve le secret de x pour une session. Le même résultat est vrai pour le secret.*

On peut alors appliquer la technique suivante. Quand on veut prouver que Q_0 préserve le secret de x (pour une session), on transforme Q_0 par les transformations décrites dans la section 3.3 avec $V = \{x\}$. Par les propositions 3.1 et 3.3, on obtient un processus Q'_0 tel que $Q_0 \approx^V Q'_0$. On utilise la proposition 3.4 ou une proposition similaire pour le secret pour montrer que Q'_0 préserve le secret de x (pour une session) et on conclut finalement que Q_0 préserve aussi le secret de x (pour une session) grâce au lemme 3.2.

Exemple 3.3 Après les transformations de l'exemple 3.2, le seul accès à x'_k dans le processus considéré est $\mathsf{let}\ x''_k : T_k = x'_k[u]$ et x''_k n'est pas utilisé. Alors, par la proposition 3.4, ce processus préserve le secret de x''_k pour une session (avec $S = \{x'_k, x''_k\}$). Le lemme 3.2 montre que le processus de la section 3.1 préserve aussi le secret de x''_k pour une session. Cependant, ce processus ne préserve pas le secret de x''_k, car l'attaquant peut forcer plusieurs sessions de B à utiliser la même clé x''_k, en rejouant le message envoyé par A. De ce fait, CryptoVerif ne prouve pas le secret de x''_k pour cet exemple.

Les critères donnés dans cette section peuvent sembler restrictifs, mais en fait, ils devraient être suffisants pour tous les protocoles, pourvu que les transformations précédentes soient assez puissantes pour transformer le protocole en un processus plus simple, sur lequel ces critères peuvent être appliqués.

3.4.2 Correspondances

Afin de formaliser les correspondances, on étend notre calcul de processus avec des événements $\mathsf{event}\ e(M_1, \ldots, M_m)$, de façon similaire à ce qui a été fait dans le modèle de Dolev-Yao dans la section 2.2.4. Ces événements ne changent pas l'état du système. Les correspondances sont des propriétés de la forme "si certains événements ont été exécutés, alors d'autres événements ont été exécutés". Plus précisément, on définit les formules logiques suivantes :

$\phi ::=$	formule
M	terme
$\mathsf{event}(e(M_1, \ldots, M_m))$	événement
$\phi_1 \wedge \phi_2$	conjonction
$\phi_1 \vee \phi_2$	disjonction

Les termes $M, M_1, \ldots, M_m$ dans les formules ne doivent pas contenir d'accès à des tableaux et leurs variables sont supposées distinctes des variables des processus. La formule M est vraie quand le terme M s'évalue en true ; la formule $\mathsf{event}(e(M_1, \ldots, M_n))$ est vraie quand l'événement $e(M_1, \ldots, M_n)$ a été exécuté. La conjonction et la disjonction sont définies comme d'habitude. Les formules notées ψ sont des conjonctions d'événements. On définit les correspondances informellement comme suit :

Définition 3.6 On note $\mathcal{E}$ une suite d'événements $e(a_1, \dots, a_m)$.

La suite d'événements $\mathcal{E}$ satisfait la correspondance $\psi \Rightarrow \phi$ si et seulement si, pour toute valeur des variables de ψ, si $\mathcal{E}$ satisfait ψ, alors il existe des valeurs des variables de ϕ qui n'apparaissent pas dans ψ telles que $\mathcal{E}$ satisfait ϕ.

Le processus Q satisfait la correspondance $\psi \Rightarrow \phi$ avec les variables publiques V si et seulement si pour tout contexte d'évaluation C acceptable pour Q, Q, V, la probabilité que $C[Q]$ exécute une suite d'événements $\mathcal{E}$ qui ne satisfait pas $\psi \Rightarrow \phi$ est négligeable.

Dans cette définition, l'attaquant est représenté par le contexte C.

Exemple 3.4 La correspondance

$$\mathsf{event}(e_1(x)) \wedge \mathsf{event}(e_2(x)) \Rightarrow \mathsf{event}(e_3(x)) \vee (\mathsf{event}(e_4(x, y)) \wedge \mathsf{event}(e_5(y, z)))$$

signifie qu'à probabilité négligeable près, pour tout x, si $e_1(x)$ et $e_2(x)$ ont été exécutés, alors $e_3(x)$ a été exécuté ou il existe y tel que $e_4(x, y)$ et $e_5(x, y)$ ont été exécutés.

La notion d'équivalence observationnelle est adaptée pour que, quand $Q \approx^V Q'$, les processus Q et Q' exécutent les mêmes événements à probabilité négligeable près, en présence d'un contexte d'évaluation acceptable pour Q, Q', V. On peut alors prouver le résultat suivant, analogue du lemme 3.2 pour les correspondances :

Lemme 3.3 *Si $Q \approx^V Q'$ et Q satisfait la correspondance c avec les variables publiques V, alors Q' la satisfait aussi.*

Les transformations de jeux de la section 3.3 laissent les événements inchangés, de sorte qu'elles transforment un processus Q en un processus Q' tel que $Q \approx^V Q'$ pour la définition d'équivalence observationnelle adaptée aux événements. On peut alors appliquer la même technique que pour le secret : pour prouver que Q_0 satisfait une correspondance c avec les variables publiques V, on le transforme en un processus Q'_0 tel que $Q_0 \approx^V Q'_0$ par les transformations de la section 3.3, et on prouve que le processus Q'_0 satisfait la correspondance c avec les variables publiques V. Alors Q_0 la satisfait aussi par le lemme 3.3. La technique utilisée pour prouver une correspondance sur Q'_0 est détaillée dans [Bla07] ; on présente ici un exemple simple.

Exemple 3.5 Considérons l'exemple de processus de la section 3.1 auquel on a ajouté des événements comme suit :

$$\begin{aligned}
&Q_0 = \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in} \\
&\qquad \mathsf{new}\ x'_r : T_{mr}; \mathsf{let}\ x_{mk} : T_{mk} = \mathrm{mkgen}(x'_r)\ \mathsf{in}\ \overline{c}\langle\rangle; (Q_A \mid Q_B) \\
&Q_A = !^{i \le n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x''_r : T'_r; \\
&\qquad \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x''_r)\ \mathsf{in\ event}\ e_A(x'_k); \overline{c_A[i]}\langle x_m, \mathrm{mac}(x_m, x_{mk})\rangle \\
&Q_B = !^{i' \le n} c_B[i'](x'_m, x_{ma}); \mathsf{if}\ \mathrm{check}(x'_m, x_{mk}, x_{ma})\ \mathsf{then} \\
&\qquad \mathsf{let}\ \mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in\ event}\ e_B(x''_k); \overline{c_B[i']}\langle\rangle
\end{aligned}$$

On souhaite prouver la correspondance suivante :

$$\mathsf{event}(e_B(x)) \Rightarrow \mathsf{event}(e_A(x)) \tag{3.1}$$

c'est-à-dire montrer que, si B a terminé le protocole avec la clé x (B a exécuté l'événement $e_B(x)$), alors A a choisi la clé x et l'a envoyée à B (A a exécuté l'événement $e_A(x)$).

Pour cela, on transforme Q_0 comme dans le cas du secret. Après avoir exécuté les transformations **RemoveAssign**(x_{mk}), sécurité du MAC et **Simplify**, on obtient le processus Q'_0 suivant comme décrit dans l'exemple 3.2 :

$$Q'_0 = \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in\ new}\ x'_r : T_{mr}; \overline{c}\langle\rangle; (Q'_A \mid Q'_B)$$

$Q'_A = !^{i \leq n} c_A[i]();$ $\mathsf{new}\ x'_k : T_k;$ $\mathsf{new}\ x''_r : T'_r;$ $\mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x''_r)\ \mathsf{in}$
$\mathsf{event}\ e_A(x'_k); \overline{c_A[i]}\langle x_m, \mathrm{mac}'(x_m, \mathrm{mkgen}'(x'_r))\rangle$
$Q'_B = !^{i' \leq n} c_B[i'](x'_m, x_{ma});$
$\mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x_m[u], x'_k[u]) \wedge x'_m = x_m[u] \wedge \mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})\ \mathsf{then}$
$\mathsf{let}\ x''_k : T_k = x'_k[u]\ \mathsf{in\ event}\ e_B(x''_k); \overline{c_B[i']}\langle\rangle$

On peut alors prouver (3.1) sur ce processus. Si l'événement $e_B(x)$ a été exécuté, alors il a été exécuté dans une certaine instance numéro i' de Q'_B, avec $x''_k[i'] = x$. (On rappelle que les variables définies sous des réplications sont implicitement des tableaux.) Puisque le point de programme event $e_B(x''_k)$ a été atteint dans cette instance de Q'_B, la condition du find est vraie, donc $x_m[u[i']]$ et $x'_k[u[i']]$ sont définies et, par définition de x''_k, $x''_k[i'] = x'_k[u[i']]$. Puisque $x'_k[u[i']]$ est définie, l'instance numéro $i = u[i']$ de Q'_A a été exécutée. Comme le contrôle ne change de processus qu'au moment des envois de messages, $e_A(x'_k[i])$ a été exécuté, et $e_A(x'_k[i]) = e_A(x'_k[u[i']]) = e_A(x''_k[i']) = e_A(x)$, ce qui prouve la correspondance souhaitée.

Les conditions de find sont souvent les points clés qui permettent de prouver les correspondances : elles permettent de prouver qu'une variable est définie, et donc que sa définition, située dans un processus en parallèle, a été exécutée. C'est ce qui permet de montrer qu'un événement a été exécuté.

CryptoVerif peut également prouver des correspondances injectives, dans lesquelles chaque exécution d'un événement de ψ correspond à un événement distinct de ϕ. Pour montrer que deux exécutions d'un événement sont distinctes, on montre que les indices de réplication associés sont distincts [Bla07]. Dans l'exemple ci-dessus, la correspondance n'est pas injective : deux exécutions de $e_B(x)$ peuvent correspondre à une seule exécution de $e_A(x)$ car l'attaquant peut rejouer le message envoyé de A à B.

Les propriétés de correspondances permettent de prouver l'authentification mutuelle et, combinées avec le secret, elles permettent de prouver qu'un protocole d'échange de clés authentifié est correct [Bla07, section 7].

3.5 Stratégie de preuve

Jusqu'à maintenant, nous avons décrit les transformations de jeux disponibles. Nous expliquons maintenant comment ces transformations sont organisées afin de prouver des protocoles.

Au début de la preuve et après chaque transformation cryptographique (c'est-à-dire une transformation de la section 3.3.2), CryptoVerif simplifie le jeu par **Simplify** et teste si la propriété de sécurité souhaitée est prouvée, comme décrit dans la section 3.4. Dans ce cas, il s'arrête.

Afin d'effectuer les transformations cryptographiques et les autres transformations syntaxiques, la stratégie de preuve est fondée sur l'idée du conseil. Plus précisément, CryptoVerif essaie d'exécuter chacune des transformations cryptographiques disponibles. Quand une telle transformation échoue, elle retourne des transformations syntaxiques qui pourraient lui permettre de réussir. (Ce sont les transformations conseillées.) CryptoVerif essaie alors d'exécuter ces transformations syntaxiques. Si elles échouent, elles peuvent à leur tour suggérer d'autres transformations conseillées, qui sont alors exécutées. Quand les transformations syntaxiques réussissent finalement, on essaie à nouveau la transformation cryptographique, qui peut réussir ou échouer, peut-être avec de nouvelles transformations conseillées, et ainsi de suite.

Par exemple, supposons qu'on essaie d'exécuter une transformation cryptographique qui nécessite de reconnaître un certain terme M de L, mais qu'on trouve dans Q_0 seulement une partie de M, les autres parties étant des accès à des variables $x[\ldots]$ alors que M contient des applications de fonctions. Dans ce cas, on conseille **RemoveAssign**(x). Par exemple, si Q_0

contient $\mathrm{enc}(M', x_k, x'_r)$ et on cherche $\mathrm{enc}(x_m, \mathrm{kgen}(x_r), x_{r'})$, on conseille **RemoveAssign**(x_k). Si Q_0 contient let $x_k = \mathrm{mkgen}(x_r)$ et on cherche $\mathrm{mac}(x_m, \mathrm{mkgen}(x_r))$, on conseille aussi **RemoveAssign**(x_k). (La transformation de l'exemple 3.1 est conseillée pour cette raison.) CryptoVerif utilise quelques autres critères pour conseiller des transformations [Bla08b, section 5].

3.6 Résultats

CryptoVerif a été testé sur des exemples de protocoles de la littérature. Ces protocoles ont été testés dans une configuration où les participants honnêtes acceptent d'exécuter le protocole avec l'attaquant. Dans ces exemples, le chiffrement à clé partagée est encodé comme chiffrement-puis-MAC où le chiffrement est IND-CPA et le MAC est UF-CMA, comme dans l'exemple de la section 3.1, le chiffrement à clé publique est supposé IND-CCA2 (*indistinguishable under adaptive chosen-ciphertext attacks*, indistinguable sous des attaques à chiffré choisi adaptatives) [BDPR98], le schéma de signature est supposé UF-CMA (*unforgeable under chosen message attacks*, inforgeable sous des attaques à messages choisis).

Les protocoles suivants ont été testés pour montrer le secret et le secret pour une session des clés échangées, et la propriété d'échange de clés authentifié : Otway-Rees [OR87], Yahalom [BAN89] avec et sans confirmation de la clé (la confirmation de la clé casse son secret), et les versions initiale et corrigée de Needham-Schroeder à clé partagée [NS78, NS87] avec et sans confirmation de la clé, Denning-Sacco à clé publique [DS81, AN96], Needham-Schroeder à clé publique [NS78, Low96] où la clé est soit un des nonces N_A ou N_B, soit $H(N_A, N_B)$. Ces protocoles ont aussi été testés pour l'authentification mutuelle ou dans un seul sens, suivant le but du protocole ; pour cette propriété, on a testé également les versions initiale et corrigée des protocoles de Woo-Lam à clé partagée [GJ01] et à clé publique [WL92, WL97] (qui n'échangent pas de clés). CryptoVerif réussit à prouver les propriétés qui sont correctes dans tous les cas sauf :

– la preuve du secret de N_A pour le protocole de Needham-Schroeder à clé publique (corrigé) échoue car CryptoVerif n'exploite pas le fait que N_A est accepté seulement après que tous les messages qui contiennent N_A ont été envoyés.
– la preuve de l'échange de clés authentifié échoue pour le protocole de Needham-Schroeder à clé publique (corrigé) quand la clé est $H(N_A, N_B)$, car CryptoVerif ne parvient pas à prouver certaines correspondances (mais la preuve d'authentification mutuelle réussit pour le protocole corrigé sans clé).
– la preuve d'une correspondance échoue pour la version initiale du protocole de Needham-Schroeder à clé partagée, car CryptoVerif ne réussit pas à prouver que $N_B[i] \neq N_B[i'] - 1$ à probabilité négligeable près, quand N_B est un nonce.

Pour les protocoles à clé publique, le mode manuel de CryptoVerif est utilisé : l'utilisateur indique les étapes principales de la preuve ; la stratégie de preuve automatique n'est pour l'instant pas suffisante pour prouver ces protocoles (en particulier, parce qu'elle ne distingue pas automatiquement les cas où l'interlocuteur est honnête ou malhonnête). Le temps total d'exécution pour tous ces tests est 2 min 45 s sur un AMD X2 4600, 2.4 GHz.

De plus, deux études de cas ont été effectuées :

– En collaboration avec David Pointcheval [BP06], nous avons prouvé la sécurité du schéma de signature FDH (*Full Domain Hash*) et de schémas de chiffrement de [BR93a]. Ces exemples utilisent des primitives cryptographiques de plus bas niveau, comme les permutations à sens unique à trappe, qui ne sont pas modélisées dans le modèle formel.
– En collaboration avec Aaron D. Jaggard, Andre Scedrov et Joe-Kai Tsay [BJST08], nous avons étudié le protocole Kerberos version 5 [NYHR05], avec et sans son extension à clé publique PKINIT [IET06].

Karthik Bhargavan et al. [BCF07] ont également commencé à utiliser CryptoVerif pour vérifier

des implantations de protocoles en F# dans le modèle calculatoire. CryptoVerif est disponible à l'adresse `http://www.cryptoverif.ens.fr/`.

3.7 Conclusion

Le vérificateur CryptoVerif a été le premier outil automatique à prouver des protocoles dans le modèle calculatoire, en fournissant des preuves par jeux, comme celles faites à la main par les cryptographes.

CryptoVerif permet de modéliser des primitives cryptographiques variées, en exprimant toutes les nuances sur leurs hypothèses de sécurité. Par exemple, on peut distinguer un MAC faiblement inforgeable (l'attaquant ne peut pas forger un MAC pour un nouveau message, comme dans la définition 3.1) d'un MAC fortement inforgeable (l'attaquant ne peut pas forger un nouveau MAC même s'il connaît un MAC pour le même message). On peut faire une distinction analogue pour les signatures. Différentes variantes de chiffrement à clé partagée peuvent être codées, dont le chiffrement bijectif déterministe PRP (*pseudo-random permutation*, permutation pseudo-aléatoire) ou SPRP (*super pseudo-random permutation*, permutation super-pseudo-aléatoire), et le chiffrement probabiliste IND-CPA, IND-CPA et INT-CTXT (*ciphertext integrity*, intégrité du chiffré) ou IND-CCA2 et INT-PTXT (*plaintext integrity*, intégrité du clair). On peut également représenter différentes variantes de chiffrement à clé publique, ainsi que des fonctions de hachage résistantes aux collisions ou des oracles aléatoires.

CryptoVerif produit des preuves valides pour un nombre de sessions polynomial dans le paramètre de sécurité, en présence d'un attaquant actif. Il prouve des propriétés de secret et de correspondances, qui permettent de montrer l'authentification mutuelle et l'échange de clés authentifié. Il fournit une borne sur la probabilité de succès d'une attaque [BP06]. CryptoVerif dispose d'une stratégie de preuve automatique, mais permet également à l'utilisateur de donner les étapes essentielles de la preuve d'un protocole, pour réussir à prouver des protocoles quand la stratégie automatique échoue, ou pour obtenir une meilleure réduction (une probabilité d'attaque plus faible). Il reste cependant beaucoup d'extensions à réaliser dans cet outil ; le chapitre suivant en mentionne quelques unes.

Récemment, Tšahhirov et Laud [TL07] ont développé un outil analogue, qui utilise une représentation des jeux par des graphes de dépendances. Cet outil est pour l'instant moins développé que CryptoVerif : il traite le chiffrement à clé publique et prouve des propriétés de secret, sans fournir de borne explicite sur la probabilité d'attaque. AVISPA inclut un module qui prouve les protocoles dans le modèle calculatoire [CHW06] en utilisant une vérification dans le modèle formel et le résultat de [CW05] qui montre que la sécurité dans le modèle formel implique la sécurité dans le modèle calculatoire pour les protocoles à clé publique (chiffrement et signatures).

Chapitre 4

Conclusion et perspectives

Ces dernières années, la vérification des protocoles cryptographiques a été un sujet de recherche très actif. Mes contributions dans ce domaine ont essentiellement consisté en la réalisation de deux outils automatiques de vérification des protocoles cryptographiques, ProVerif et CryptoVerif. Mes contributions ont été à la fois théoriques et pratiques : j'ai implanté des logiciels efficaces, mais j'ai aussi veillé à ce qu'ils reposent sur des fondements théoriques précis, en particulier en prouvant leur correction vis-à-vis d'une sémantique formelle du langage. Ces allers-retours entre la théorie et la pratique ont été particulièrement enrichissants.

Le vérificateur ProVerif repose sur le modèle formel des protocoles, dit modèle de Dolev-Yao. Sa principale caractéristique est qu'il peut prouver des protocoles pour un nombre non-borné de sessions, grâce à une représentation abstraite par des clauses de Horn. Cet outil arrive essentiellement à maturité : il a déjà été utilisé par de nombreux chercheurs, et mes travaux sur cet outil consisteront principalement à améliorer sa documentation et son interface, de façon à favoriser son adoption plus large.

Contrairement à la plupart des vérificateurs automatiques de protocoles, CryptoVerif travaille dans le modèle calculatoire et produit des preuves par jeux. Il permet donc d'obtenir des preuves dans un modèle plus réaliste et proches de celles habituellement écrites par les cryptographes. Ceci constitue une avancée très importante. Cependant, beaucoup d'extensions sont encore à réaliser avant d'obtenir un outil largement utilisable :

– Des améliorations de la stratégie de preuve seraient utiles afin d'obtenir plus souvent des preuves automatiques, en particulier pour les protocoles à clé publique.
– Les transformations cryptographiques devraient être étendues, pour traiter davantage de primitives, en particulier la mise en accord de clés de Diffie-Hellman, qui est utilisée dans beaucoup de protocoles importants.
– On devrait également traiter davantage d'équations, en particulier les symboles associatifs et commutatifs. (Ces équations sont en fait plus faciles à traiter dans CryptoVerif que dans ProVerif : dans ProVerif, on aurait besoin d'unification modulo les équations, alors que dans CryptoVerif, le filtrage modulo suffit.)
– Pour plus de facilité d'utilisation, on pourrait créer une bibliothèque des primitives cryptographiques les plus courantes avec leur codage, pour que l'utilisateur n'ait pas besoin de les coder lui-même.
– Il serait également intéressant de traiter plus d'exemples de protocoles, en particulier des protocoles qui ne sont pas habituellement analysés par des méthodes formelles. Ces exemples pourraient suggérer de nouvelles extensions à réaliser.

Un autre aspect sur lequel je n'ai pas encore travaillé, mais qui me paraît particulièrement important est la vérification d'implantations de protocoles, dans des langages de programmation standards. En effet, les travaux précédents vérifient des protocoles spécifiés dans des modèles comme le pi calcul appliqué ou des variantes, mais des erreurs peuvent être introduites au moment de l'implantation du protocole. Il est donc important de prouver les propriétés de

sécurité sur l'implantation du protocole. Pour cela, on peut distinguer deux approches :

- Une approche simple consiste à traduire le modèle en une implantation par un compilateur adapté, et dont on a prouvé la correction. Cette approche a été utilisée dans des outils comme [SPP01, Mil02, PSD04]. Une limitation de cette approche est qu'elle offre moins de flexibilité dans le codage du protocole qu'un langage de programmation habituel (ou bien il faut transformer un langage de modélisation en un vrai langage de programmation, ce qui est difficile).
- Une approche qui offre plus de flexibilité consiste à analyser l'implantation du protocole. Plusieurs travaux ont commencé à traiter ce problème.

  Goubault-Larrecq et Parrennes [GLP05] analysent des protocoles écrits en C et les traduisent en clauses de Horn, obtenant un modèle assez similaire à celui utilisé dans ProVerif. Ils utilisent ensuite le prouveur $\mathcal{H}_1$ de Goubault-Larrecq [GL05] pour prouver des propriétés sur le protocole.

  Bhargavan et al. [BFGT06] analysent des protocoles écrits dans un sous-ensemble de F# en les traduisant dans le langage d'entrée de ProVerif, et en utilisant ProVerif pour prouver les propriétés de sécurité. Récemment, ce travail a commencé à être adapté pour traduire le programme en F# vers le langage d'entrée de CryptoVerif [BCF07]. Contrairement aux autres travaux mentionnés ici, ceci permet donc d'obtenir des preuves dans le modèle calculatoire. Ces travaux analysent des implantations de référence écrites en F# dans le but de faciliter la vérification ; on vérifie que ces implantations de référence sont raisonnables en vérifiant leur interopérabilité avec d'autres implantations ; cependant, on ne peut pas encore analyser directement le code d'implantations écrites sans chercher à faciliter la vérification.

  Bengtson et al. [BBF+08] ont proposé un système de types pour prouver des propriétés de sécurité de protocoles implantés en F#, étendant ainsi aux implantations l'approche de Cryptic [GJ03, GJ04, GJ02] pour les modèles. Cette approche nécessite des annotations de types, qui facilitent la vérification automatique.

  Poll et Schubert [PS07] ont vérifié une implantation libre de SSH en Java, en utilisant ESC/Java2. C'est à ma connaissance le seul travail qui a vérifié une implantation qui n'a pas été construite pour cela. Cependant, ce travail présente des limitations : ESC/Java2 vérifie que l'implantation ne lance pas d'exceptions à l'exécution, et vérifie également que l'implantation respecte une spécification formelle de SSH par un automate fini, qui spécifie l'ordre des messages, mais pas leur contenu. De ce fait, les propriétés de sécurité du protocole ne sont pas vérifiées.

Des travaux importants sur la vérification d'implantations de protocoles ont donc déjà été réalisés, mais il reste encore beaucoup de travail pour atteindre l'objectif idéal à long terme de prouver automatiquement des implantations réellement utilisées dans le modèle calculatoire.

Chapitre 5

Activités d'enseignement et d'encadrement

5.1 Enseignement

J'ai commencé à enseigner dès la fin de mon DEA et je n'ai jamais cessé depuis. J'ai tout d'abord effectué des travaux dirigés d'informatique dans des grandes écoles et à l'université, puis j'ai participé à des cours de DEA (devenu depuis master recherche).

5.1.1 Travaux dirigés à l'École polytechnique

Ma première expérience de l'enseignement a été l'encadrement de travaux dirigés du cours de tronc commun "Algorithmes et programmation" de Jean-Jacques Lévy et Robert Cori à l'École polytechnique. Ce cours abordait les thèmes suivants : tableaux (et tris), récursivité (et tris), structures de données : listes, piles, files, arbres, graphes analyse syntaxique, modularité. J'ai effectué 48 heures par an en encadrant deux groupes d'un peu plus de 20 élèves, sous la responsabilité d'un chargé de TD ; chaque groupe effectuait 2 heures de TD par semaine, pendant 12 semaines.

- En 1996-97, j'ai enseigné, en tant que vacataire, le langage Pascal à des groupes de débutants, sous la responsabilité de Jean-Dominique Gascuel ; Fabrice Le Fessant et moi encadrions ces deux groupes.
- En 1997-98, j'ai enseigné, en tant que vacataire, le langage C à des groupes moyens, sous la responsabilité de Michel Mauny.
- En 1998-99, j'ai enseigné, en tant que scientifique du contingent au laboratoire d'informatique de l'École polytechnique, le langage Java à des groupes forts, sous la responsabilité de Philippe Chassignet.

5.1.2 Travaux dirigés à l'ENSTA

En 1997-98, j'ai également encadré un groupe de travaux dirigés dans le cours de Standard ML de Philippe Granger et Alain Deutsch (IN202) à l'ENSTA (École Nationale Supérieure des Techniques Avancées), au niveau débutant. J'ai effectué 15 heures de travaux dirigés, sous forme de vacations. Les sujets abordés dans ces travaux dirigés étaient les suivants : fonctions récursives, typage, types inductifs, structures et signatures, fonctions d'ordre supérieur, preuves de programmes ML, références.

5.1.3 Travaux dirigés à l'Université de Versailles

Dans le cadre de ma bourse de thèse (allocation couplée), j'ai été moniteur à l'université de Versailles. J'ai encadré des travaux dirigés de Java en DEUG MIAS 2e année, au 2e semestre,

en 1999-00 et 2000-01. J'ai encadré deux groupes d'une trentaine d'étudiants, chaque groupe effectuant 2 heures de TD par semaine. J'ai participé à la préparation des sujets de TD, d'examen et à la correction des examens. Les sujets abordés en TD étaient : tris, exceptions, fichiers, *applets*, interfaces graphiques. Ces TD comprenaient également un projet, à réaliser à maison par les étudiants et que j'ai corrigé pour mes groupes de TD.

5.1.4 Cours en DEA et Master

De 1999-00 à 2006-07, je suis intervenu 6 heures par an dans le cours d'interprétation abstraite au DEA Sémantique, Preuves et Programmation, devenu en 2000-01, DEA Programmation : Sémantique, Preuves et Langages, puis en 2004-05, Master Parisien de Recherche en Informatique (MPRI). L'intitulé et les responsables du cours ont varié suivant les années : jusqu'en 2001-02, "Analyse statique par Interprétation abstraite" de Radhia Cousot, Alain Deutsch et Arnaud Venet, en 2002-03, "Analyse statique par Interprétation abstraite", de Radhia Cousot, Laurent Mauborgne et moi, en 2003-04, "Analyse statique de propriétés numériques, de sécurité et de mobilité", de Radhia Cousot, Mathieu Martel et moi, puis à partir de 2004-05, "Interprétation abstraite : application à la vérification et à l'analyse statique", de Patrick et Radhia Cousot.

- De 1999-00 à 2002-03, j'ai présenté mes travaux de thèse sur l'analyse d'échappement pour ML et Java [Bla98, Bla03, Bla00]. Cette analyse statique par interprétation abstraite permet de prouver que la durée de vie de certaines données ne dépasse pas leur portée statique, ce qui permet ensuite d'effectuer des allocations en pile et, pour Java, d'éliminer des synchronisations sur des données locales à un seul *thread*.
- De 2003-04 à 2006-07, j'ai présenté mes travaux sur la vérification automatique de protocoles cryptographiques. Tous les ans, j'ai présenté la technique de vérification du secret et des propriétés de correspondances dans ProVerif [Bla01, AB05a, Bla08a]. En 2005-06, j'ai de plus présenté la vérification des équivalences de processus dans ProVerif [BAF08]. En 2006-07, j'ai de plus présenté une brève introduction à CryptoVerif [Bla08b].

Cette année (2007-08), j'ai été co-responsable avec Steve Kremer du cours "Protocoles cryptographiques : preuves formelles et calculatoires" (24 heures de cours au total, dont 12 que j'ai enseignées). Steve Kremer a présenté une introduction aux protocoles, les résultats de correction du modèle formel vis-à-vis du modèle calculatoire d'Abadi et Rogaway [AR02] (cas passif) et Cortier et Warinschi [CW05] (cas actif), ainsi que l'indécidabilité de la vérification des protocoles dans le cas général et la décidabilité pour un nombre borné de sessions [RT03]. J'ai présenté les vérificateurs de protocoles ProVerif et CryptoVerif. Cédric Fournet est intervenu sur la vérification de protocoles et de leur implantations dans le cas des services web (3 heures).

5.2 Encadrement

J'ai encadré six stagiaires, d'abord au Max-Planck-Institut für Informatik, à Sarrebruck, puis à l'École normale supérieure à Paris.

5.2.1 Reconstruction d'attaques contre des protocoles cryptographiques

Xavier Allamigeon a effectué son stage d'option scientifique de l'École polytechnique sous ma direction d'avril à juillet 2004 au Max-Planck-Institut für Informatik. Il a conçu et implanté dans ProVerif un algorithme de reconstruction d'attaques contre les protocoles cryptographiques. Il a prouvé sa correction, sa terminaison, et un résultat de complétude partielle : si la dérivation calculée par ProVerif correspond à une attaque, alors l'algorithme réussit à reconstruire cette attaque. Pour citer un exemple extrême, cet algorithme a permis de reconstruire une attaque impliquant 200 sessions en parallèle contre le protocole $f^{200}g^{200}$ [Mil99]. Ce travail a donné lieu à une publication [AB05c].

5.2.2 Analyse de protocoles présentés comme une liste de messages

Mehmet Kiraz a effectué son stage de master de l'université de la Sarre sous ma direction d'avril à octobre 2003. Il a réalisé une étude théorique en vue de traduire en clauses de Horn un protocole représenté comme une suite de messages. Cette traduction pose des problèmes délicats, dans la mesure où la liste de messages représente une exécution correcte du protocole, mais n'explicite pas comment les participants réagissent à des messages incorrects. Il faut donc déterminer quels tests les participants peuvent faire sur les messages reçus.

Dans le cadre de son stage d'option scientifique de l'École polytechnique, d'avril à juin 2005 à l'École normale supérieure, Yannick Gérault a étendu ce travail : tandis que Mehmet Kiraz avait considéré uniquement quelques primitives fixées (chiffrement à clé partagée et à clé publique), Yannick Gérault a considéré des primitives définies par des règles de réécriture arbitraires.

5.2.3 Analyse d'implantations de protocoles cryptographiques en Java

Plusieurs stagiaires ont travaillé sur un projet à long terme de vérification d'implantations de protocoles cryptographiques en Java.

Dans le cadre de son stage d'été de l'IIT Kanpur (juin et juillet 2002, au Max-Planck-Institut für Informatik), Shiv Pratap Raghuwanshi a réalisé une implantation du protocole SSH (*Secure SHell*) dans le langage Java, spécialement conçue pour faciliter la vérification automatique. Cette implantation est destinée à servir d'étude de cas pour la vérification d'implantations de protocoles.

Dans le cadre d'un stage au Max-Planck-Institut für Informatik (juin à août 2002), Emma Rabbidge a réalisé un *front-end* d'analyseur de *bytecode* Java. Ce *front-end* détermine les classes nécessaires à l'application considérée par clôture transitive, et transforme le *bytecode* en un code trois adresses en forme SSA (*Static Single Assignment*, où chaque variable est affectée en un seul point de programme) [CFR+91], en vue de faciliter son analyse.

Dans le cadre d'un stage long à l'École normale supérieure (janvier à juin 2006), Maël Primet a conçu et commencé à implanter un prototype d'analyseur de *bytecode* Java, destiné à traduire le programme en un ensemble de clauses de Horn, afin de prouver des propriétés de sécurité sur le programme Java, de la même façon qu'on les prouve dans ProVerif.

Bibliographie

[AB03] Martín ABADI et Bruno BLANCHET. – Secrecy types for asymmetric communication. *Theoretical Computer Science*, vol. 298, n° 3, avril 2003, pp. 387–415. – Special issue FoSSaCS'01.

[AB05a] Martín ABADI et Bruno BLANCHET. – Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, vol. 52, n° 1, janvier 2005, pp. 102–146. – Article joint en annexe.

[AB05b] Martín ABADI et Bruno BLANCHET. – Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, vol. 58, n° 1–2, octobre 2005, pp. 3–27. – Special issue SAS'03.

[AB05c] Xavier ALLAMIGEON et Bruno BLANCHET. – Reconstruction of attacks against cryptographic protocols. *In : 18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pp. 140–154, Aix-en-Provence, France, juin 2005. IEEE.

[Aba99] Martín ABADI. – Secrecy by typing in security protocols. *Journal of the ACM*, vol. 46, n° 5, septembre 1999, pp. 749–786.

[ABB+04] William AIELLO, Steven M. BELLOVIN, Matt BLAZE, Ran CANETTI, John IOANNIDIS, Keromytis KEROMYTIS et Omer REINGOLD. – Just Fast Keying : Key agreement in a hostile Internet. *ACM Transactions on Information and System Security*, vol. 7, n° 2, mai 2004, pp. 242–273.

[ABB+05] Alessandro ARMANDO, David BASIN, Yohan BOICHUT, Yannick CHEVALIER, Luca COMPAGNA, Jorge CUELLAR, Paul Hankes DRIELSMA, Pierre-Cyrille HÉAM, Olga KOUCHNARENKO, Jacopo MANTOVANI, Sebastian MÖDERSHEIM, David VON OHEIMB, Michaël RUSINOWITCH, Judson SANTIAGO, Mathieu TURUANI, Luca VIGANÓ et Laurent VIGNERON. – The AVISPA tool for automated validation of Internet security protocols and applications. *In : Computer Aided Verification, 17th International Conference, CAV 2005*, éd. par Kousha ETESSAMI et Sriram K. RAJAMANI, *Lecture Notes on Computer Science*, volume 3576, pp. 281–285, Edinburgh, Scotland, juillet 2005. Springer.

[ABF07] Martín ABADI, Bruno BLANCHET et Cédric FOURNET. – Just fast keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, n° 3, juillet 2007, pp. 1–59.

[ABHS05] Pedro ADÃO, Gergei BANA, Jonathan HERZOG et Andre SCEDROV. – Soundness of formal encryption in the presence of key-cycles. *In : Proceedings of the 10th European Symposium On Research In Computer Security (ESORICS 2005)*, éd. par Sabrina DE CAPITANI DI VIMERCATI, Paul SYVERSON et Dieter GOLLMANN, *Lecture Notes on Computer Science*, volume 3679, pp. 374–396, Milan, Italy, septembre 2005. Springer.

[ABW06] Martín ABADI, Mathieu BAUDET et Bogdan WARINSCHI. – Guessing attacks and the computational soundness of static equivalence. *In : Proceedings of the 9th International Conference on Foundations of Software Science and Computation*

Structures (FoSSaCS'06), éd. par Luca ACETO et Anna INGÓLFSDÓTTIR, *Lecture Notes on Computer Science*, volume 3921, pp. 398–412, Vienna, Austria, mars 2006. Springer.

[AC06] Martín ABADI et Véronique CORTIER. – Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science*, vol. 367, n° 1–2, novembre 2006, pp. 2–32.

[ACG03] Alessandro ARMANDO, Luca COMPAGNA et Pierre GANTY. – SAT-based model-checking of security protocols using planning graph analysis. *In : FME 2003 : Formal Methods, International Symposium of Formal Methods Europe*, éd. par Keijiro ARAKI, Stefania GNESI et Dino MANDRIOLI, *Lecture Notes on Computer Science*, volume 2805, pp. 875–893, Pisa, Italy, septembre 2003. Springer.

[AD07] Myrto ARAPINIS et Marie DUFLOT. – Bounding messages for free in security protocols. *In : 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, éd. par V. ARVIND et Sanjiva PRASAD, *Lecture Notes on Computer Science*, volume 4855, pp. 376–387, New Delhi, India, décembre 2007. Springer.

[AF01] Martín ABADI et Cédric FOURNET. – Mobile values, new names, and secure communication. *In : 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pp. 104–115, London, United Kingdom, janvier 2001. ACM Press.

[AFP06] Michel ABDALLA, Pierre-Alain FOUQUE et David POINTCHEVAL. – Password-based authenticated key exchange in the three-party setting. *IEE Proceedings Information Security*, vol. 153, n° 1, mars 2006, pp. 27–39.

[AG98] Martín ABADI et Andrew D. GORDON. – A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, vol. 5, n° 4, Winter 1998, pp. 267–303.

[AG99] Martín ABADI et Andrew D. GORDON. – A calculus for cryptographic protocols : The spi calculus. *Information and Computation*, vol. 148, n° 1, janvier 1999, pp. 1–70. – An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[AGHP02] Martín ABADI, Neal GLEW, Bill HORNE et Benny PINKAS. – Certified email with a light on-line trusted third party : Design and implementation. *In : 11th International World Wide Web Conference*, pp. 387–395, Honolulu, Hawaii, mai 2002. ACM Press.

[AJ01] Martín ABADI et Jan JÜRJENS. – Formal eavesdropping and its computational interpretation. *In : Theoretical Aspects of Computer Software (TACS'01)*, éd. par N. KOBAYASHI et B.C. PIERCE, *Lecture Notes on Computer Science*, volume 2215, pp. 82–94, Sendai, Japan, octobre 2001. Springer.

[AN95] Ross ANDERSON et Roger NEEDHAM. – Programming Satan's computer. *In : Computer Science Today : Recent Trends and Developments*, éd. par J. VAN LEEUVEN, *Lecture Notes on Computer Science*, volume 1000, pp. 426–440. Springer, 1995.

[AN96] Martín ABADI et Roger NEEDHAM. – Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, vol. 22, n° 1, janvier 1996, pp. 6–15.

[AR02] Martín ABADI et Phillip ROGAWAY. – Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, vol. 15, n° 2, 2002, pp. 103–127.

[AVI03] AVISPA. – Deliverable D2.3 : The intermediate format. – Available at `http://www.avispa-project.org.`, 2003.

[BAF05] Bruno BLANCHET, Martín ABADI et Cédric FOURNET. – Automated verification of selected equivalences for security protocols. *In : 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pp. 331–340, Chicago, IL, juin 2005. IEEE Computer Society.

[BAF08] Bruno BLANCHET, Martín ABADI et Cédric FOURNET. – Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, vol. 75, n° 1, février–mars 2008, pp. 3–51. – Article joint en annexe.

[BAN89] Michael BURROWS, Martín ABADI et Roger NEEDHAM. – A logic of authentication. *Proceedings of the Royal Society of London A*, vol. 426, 1989, pp. 233–271. – A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

[Bau07] Mathieu BAUDET. – *Sécurité des protocoles cryptographiques : aspects logiques et calculatoires*. – Thèse de PhD, Ecole Normale Supérieure de Cachan, janvier 2007.

[BBD+05] Chiara BODEI, Mikael BUCHHOLTZ, Pierpaolo DEGANO, Flemming NIELSON et Hanne Riis NIELSON. – Static validation of security protocols. *Journal of Computer Security*, vol. 13, n° 3, 2005, pp. 347–390.

[BBF+08] Jesper BENGTSON, Karthikeyan BHARGAVAN, Cédric FOURNET, Andy GORDON et Sergio MAFFEIS. – Refinement types for secure implementations. *In : 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pp. 17–32, Pittsburgh, PA, juin 2008. IEEE Computer Society.

[BBN04] Johannes BORGSTRÖM, Sébastien BRIAIS et Uwe NESTMANN. – Symbolic bisimulation in the spi calculus. *In : CONCUR 2004 : Concurrency Theory*, éd. par Philippa GARDNER et Nobuko YOSHIDA, *Lecture Notes on Computer Science*, volume 3170, pp. 161–176. Springer, août 2004.

[BC08] Bruno BLANCHET et Avik CHAUDHURI. – Automated formal analysis of a protocol for secure file sharing on untrusted storage. *In : IEEE Symposium on Security and Privacy*, pp. 417–431, Oakland, CA, mai 2008. IEEE.

[BCC+02] Bruno BLANCHET, Patrick COUSOT, Radhia COUSOT, Jérôme FERET, Laurent MAUBORGNE, Antoine MINÉ, David MONNIAUX et Xavier RIVAL. – Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. *In : The Essence of Computation : Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, éd. par T. MOGENSEN, D. A. SCHMIDT et I. H. SUDBOROUGH, pp. 85–108. – Springer, décembre 2002.

[BCC+03] Bruno BLANCHET, Patrick COUSOT, Radhia COUSOT, Jérôme FERET, Laurent MAUBORGNE, Antoine MINÉ, David MONNIAUX et Xavier RIVAL. – A static analyzer for large safety-critical software. *In : ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pp. 196–207, San Diego, California, juin 2003. ACM.

[BCF07] Karthikeyan BHARGAVAN, Ricardo CORIN et Cédric FOURNET. – Crypto-verifying protocol implementations in ML. *In : Workshop on Formal and Computational Cryptography (FCC'07)*, Venice, Italy, juillet 2007.

[BCFG04] Karthikeyan BHARGAVAN, Ricardo CORIN, Cédric FOURNET et Andrew GORDON. – Secure sessions for web services. *In : ACM Workshop on Secure Web Services (SWS'04)*, Washington DC, octobre 2004.

[BCK05] Mathieu BAUDET, Véronique CORTIER et Steve KREMER. – Computationally sound implementations of equational theories against passive adversaries. *In : Proceedings of the 32nd International Colloquium on Automata, Languages and*

Programming (ICALP'05), éd. par Luís Caires et Luís Monteiro, *Lecture Notes on Computer Science*, volume 3580, pp. 652–663, Lisboa, Portugal, juillet 2005. Springer.

[BCLM05] Stefano Bistarelli, Iliano Cervesato, Gabriele Lenzini et Fabio Martinelli. – Relating multiset rewriting and process algebras for security protocol analysis. *Journal of Computer Security*, vol. 13, n° 1, 2005, pp. 3–47.

[BCM07] Michael Backes, Agostino Cortesi et Matteo Maffei. – Causality-based abstraction of multiplicity in security protocols. *In : 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 355–369, Venice, Italy, juillet 2007. IEEE.

[BCT04] Gilles Barthe, Jan Cederquist et Sabrina Tarento. – A machine-checked formalization of the generic model and the random oracle model. *In : Second International Joint Conference on Automated Reasoning (IJCAR'04)*, éd. par David Basin et Michael Rusinowitch, *Lecture Notes on Computer Science*, volume 3097, pp. 385–399, Cork, Ireland, juillet 2004. Springer.

[BDJR97] Mihir Bellare, Anand Desai, E. Jokipii et Phillip Rogaway. – A concrete security treatment of symmetric encryption. *In : Proceedings of the 38th Symposium on Foundations of Computer Science (FOCS'97)*, pp. 394–403, Miami Beach, Florida, octobre 1997. IEEE. Full paper available at `http://www-cse.ucsd.edu/users/mihir/papers/sym-enc.html`.

[BDK07] Michael Backes, Markus Dürmuth et Ralf Küsters. – On simulatability soundness and mapping soundness of symbolic cryptography. *In : 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, éd. par V. Arvind et Sanjiva Prasad, *Lecture Notes on Computer Science*, volume 4855, pp. 108–120, New Delhi, India, décembre 2007. Springer.

[BDNN98] Chiara Bodei, Pierpaolo Degano, Flemming Nielson et Hanne Riis Nielson. – Control flow analysis for the π-calculus. *In : International Conference on Concurrency Theory (Concur'98)*, *Lecture Notes on Computer Science*, volume 1466, pp. 84–98. Springer, septembre 1998.

[BDP02] Michele Boreale, Rocco De Nicola et Rosario Pugliese. – Proof techniques for cryptographic processes. *SIAM Journal on Computing*, vol. 31, n° 3, 2002, pp. 947–986.

[BDPR98] Mihir Bellare, Anand Desai, David Pointcheval et Phillip Rogaway. – Relations among notions of security for public-key encryption schemes. *In : Advances in Cryptology – CRYPTO 1998*, éd. par H. Krawczyk, *Lecture Notes on Computer Science*, volume 1462, pp. 26–45, Santa Barbara, California, USA, août 1998. Springer.

[BFG04] Karthikeyan Bhargavan, Cédric Fournet et Andrew Gordon. – Verifying policy-based security for web services. *In : ACM Conference on Computer and Communications Security (CCS'04)*, pp. 268–277, Washington DC, octobre 2004. ACM.

[BFG06] Karthikeyan Bhargavan, Cédric Fournet et Andrew Gordon. – Verified reference implementations of WS-Security protocols. *In : 3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, éd. par Mario Bravetti, Manuel Núñez et Gianluigi Zavattaro, *Lecture Notes on Computer Science*, volume 4184, pp. 88–106, Vienna, Austria, septembre 2006. Springer.

[BFGP03] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon et Riccardo Pucella. – TulaFale : A security tool for web services. *In : Formal Methods for Components and Objects (FMCO 2003)*, *Lecture Notes on Computer Science*,

volume 3188, pp. 197–222, Leiden, The Netherlands, novembre 2003. Springer. Paper and tool available at `http://securing.ws/`.

[BFGS08] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon et Nikhil Swamy. – Verified implementations of the information card federated identity-management protocol. *In : ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pp. 123–135, Tokyo, Japan, mars 2008. ACM.

[BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon et Stephen Tse. – Verified interoperable implementations of security protocols. *In : 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pp. 139–152, Venice, Italy, juillet 2006. IEEE Computer Society.

[BFM05] Michele Bugliesi, Riccardo Focardi et Matteo Maffei. – Analysis of typed analyses of authentication protocols. *In : Proc. 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pp. 112–125, Aix-en-Provence, France, juin 2005. IEEE Comp. Soc. Press.

[BFM07] Michele Bugliesi, Riccardo Focardi et Matteo Maffei. – Dynamic types for authentication. *Journal of Computer Security*, vol. 15, n° 6, 2007, pp. 563–617.

[BFP+01] Olivier Baudron, Pierre-Alain Fouque, David Pointcheval, Guillaume Poupard et Jacques Stern. – Practical multi-candidate election system. *In : Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, pp. 274–283, Newport, Rhode Island, août 2001. ACM Press.

[BG01] L. Bachmair et H. Ganzinger. – Resolution theorem proving. *In : Handbook of Automated Reasoning*, éd. par A. Robinson et A. Voronkov, chap. 2, pp. 19–100. – North Holland, 2001.

[BG02] Michele Boreale et Daniele Gorla. – On compositional reasoning in the spi-calculus. *In : Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002*, éd. par M. Nielsen et U. Engberg, *Lecture Notes on Computer Science*, volume 2303, pp. 67–81, Grenoble, France, avril 2002. Springer.

[BHL06] Andrea Bittau, Mark Handley et Joshua Lackey. – The final nail in WEP's coffin. *In : IEEE Symposium on Security and Privacy*, pp. 386–400, Oakland, California, mai 2006. IEEE Computer Society.

[BHM08] Michael Backes, Catalin Hritcu et Matteo Maffei. – Automated verification of electronic voting protocols in the applied pi-calculus. *In : 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pp. 195–209, Pittsburgh, PA, juin 2008. IEEE Computer Society.

[BJST08] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov et Joe-Kai Tsay. – Computationally sound mechanized proofs for basic and public-key Kerberos. *In : ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pp. 87–99, Tokyo, Japan, mars 2008. ACM.

[BKR00] Mihir Bellare, Joe Kilian et Phillip Rogaway. – The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, vol. 61, n° 3, décembre 2000, pp. 362–399.

[BKV06] Yohan Boichut, Nikolai Kosmatov et Laurent Vigneron. – Validation of prouvé protocols using the automatic tool TA4SP. *In : Proceedings of the Third Taiwanese-French Conference on Information Technology (TFIT 2006)*, pp. 467–480, Nancy, France, mars 2006.

[BL06] Michael Backes et Peeter Laud. – Computationally sound secrecy proofs by mechanized flow analysis. *In : Proceedings of 13th ACM Conference on Computer*

and Communications Security (CCS'06), pp. 370–379, Alexandria, VA, novembre 2006. ACM.

[Bla98] Bruno Blanchet. – Escape analysis : Correctness proof, implementation and experimental results. *In : 25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*, pp. 25–37, San Diego, California, janvier 1998. ACM Press.

[Bla00] Bruno Blanchet. – *Analyse d'échappement. Applications à ML et Java^{TM}*. – Thèse de PhD, École Polytechnique, 7 décembre 2000.

[Bla01] Bruno Blanchet. – An efficient cryptographic protocol verifier based on Prolog rules. *In : 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pp. 82–96, Cape Breton, Nova Scotia, Canada, juin 2001. IEEE Computer Society.

[Bla03] Bruno Blanchet. – Escape analysis for Java^{TM}. theory and practice. *ACM Transactions on Programming Languages and Systems*, vol. 25, n° 6, novembre 2003, pp. 713–775.

[Bla04a] Bruno Blanchet. – Automatic proof of strong secrecy for security protocols. *In : IEEE Symposium on Security and Privacy*, pp. 86–100, Oakland, California, mai 2004.

[Bla04b] Bruno Blanchet. – *Automatic Proof of Strong Secrecy for Security Protocols*. – Rapport technique n° MPI-I-2004-NWG1-001, Saarbrücken, Germany, Max-Planck-Institut für Informatik, juillet 2004.

[Bla05] Bruno Blanchet. – Security protocols : From linear to classical logic by abstract interpretation. *Information Processing Letters*, vol. 95, n° 5, septembre 2005, pp. 473–479.

[Bla07] Bruno Blanchet. – Computationally sound mechanized proofs of correspondence assertions. *In : 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 97–111, Venice, Italy, juillet 2007. IEEE. Extended version available as ePrint Report 2007/128, `http://eprint.iacr.org/2007/128`.

[Bla08a] Bruno Blanchet. – Automatic verification of correspondences for security protocols. – Report arXiv:0802.3444v1, 2008. Article joint en annexe et disponible à `http://arxiv.org/abs/0802.3444v1`. Version sans preuves à paraître dans le *Journal of Computer Security*.

[Bla08b] Bruno Blanchet. – A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, vol. 5, n° 4, octobre–décembre 2008, pp. 193–207. – Article joint en annexe.

[BLMW07] Emmanuel Bresson, Yassine Lakhnech, Laurent Mazaré et Bogdan Warinschi. – A generalization of DDH with applications to protocol analysis and computational soundness. *In : Advances in Cryptology – CRYPTO 2007*, éd. par A. J. Menezes, *Lecture Notes on Computer Science*, volume 4622, pp. 482–499. Springer, août 2007.

[BLP06] Liana Bozga, Yassine Lakhnech et Michaël Périn. – Pattern-based abstraction for verifying secrecy in protocols. *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, n° 1, février 2006, pp. 57–76.

[BLR00] Philippa Broadfoot, Gavin Lowe et Bill Roscoe. – Automating data independence. *In : 6th European Symposium on Research in Computer Security (ESORICS 2000)*, *Lecture Notes on Computer Science*, volume 1895, pp. 175–190, Toulouse, France, octobre 2000. Springer.

[BM92] Steven M. Bellovin et Michael Merritt. – Encrypted Key Exchange : Password-based protocols secure against dictionary attacks. *In : Proceedings of the 1992*

IEEE Computer Society Symposium on Research in Security and Privacy, pp. 72–84, mai 1992.

[BM93] Steven M. Bellovin et Michael Merritt. – Augmented Encrypted Key Exchange : a password-based protocol secure against dictionary attacks and password file compromise. *In : Proceedings of the First ACM Conference on Computer and Communications Security*, pp. 244–250, novembre 1993.

[BMU08] Michael Backes, Matteo Maffei et Dominique Unruh. – Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. *In : 29th IEEE Symposium on Security and Privacy*, pp. 202–215, Oakland, CA, mai 2008. IEEE. Rapport technique disponible à `http://eprint.iacr.org/2007/289`.

[BMV03] David Basin, Sebastian Mödersheim et Luca Viganò. – An on-the-fly model-checker for security protocol analysis. *In : Computer Security – ESORICS 2003, 8th European Symposium on Research in Computer Security*, éd. par Einar Snekkenes et Dieter Gollman, *Lecture Notes on Computer Science*, volume 2808, pp. 253–270, Gjøvik, Norway, octobre 2003. Springer.

[BN00] Mihir Bellare et Chanathip Namprempre. – Authenticated encryption : Relations among notions and analysis of the generic composition paradigm. *In : Advances in Cryptology – ASIACRYPT'00*, éd. par T. Okamoto, *Lecture Notes on Computer Science*, volume 1976, pp. 531–545, Kyoto, Japan, décembre 2000. Springer.

[BN05] Johannes Borgström et Uwe Nestmann. – On bisimulations for the spi calculus. *Mathematical Structures in Computer Science*, vol. 15, n° 3, juin 2005, pp. 487–552.

[Bod00] Chiara Bodei. – *Security Issues in Process Calculi*. – Thèse de PhD, Università di Pisa, janvier 2000.

[Bol97] Dominique Bolignano. – Towards a mechanization of cryptographic protocol verification. *In : 9th International Conference on Computer Aided Verification (CAV'97)*, éd. par O. Grumberg, *Lecture Notes on Computer Science*, volume 1254, pp. 131–142. Springer, 1997.

[BP04] Michael Backes et Birgit Pfitzmann. – Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. *In : 17th IEEE Computer Security Foundations Workshop*, pp. 204–218, Pacific Grove, CA, juin 2004. IEEE.

[BP05a] Michael Backes et Birgit Pfitzmann. – Relating symbolic and cryptographic secrecy. *IEEE Transactions on Dependable and Secure Computing*, vol. 2, n° 2, avril 2005, pp. 109–123.

[BP05b] Bruno Blanchet et Andreas Podelski. – Verification of cryptographic protocols : Tagging enforces termination. *Theoretical Computer Science*, vol. 333, n° 1-2, mars 2005, pp. 67–90. – Special issue FoSSaCS'03.

[BP06] Bruno Blanchet et David Pointcheval. – Automated security proofs with sequences of games. *In : Advances in Cryptology – CRYPTO 2006*, éd. par Cynthia Dwork, *Lecture Notes on Computer Science*, volume 4117, pp. 537–554, Santa Barbara, CA, août 2006. Springer.

[BPR00] Mihir Bellare, David Pointcheval et Phillip Rogaway. – Authenticated key exchange secure against dictionary attacks. *In : Advances in Cryptology – Proceedings of EUROCRYPT '00*, éd. par B. Preneel, *Lecture Notes on Computer Science*, volume 1807, pp. 139–155, Bruges, Belgique, 2000. Springer.

[BPS07] Michael Backes, Birgit Pfiztmann et Andre Scedrov. – Key-dependent message security under active attacks—brsim/uc soundness of symbolic encryption with key cycles. *In : 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 112–124, Venice, Italy, juillet 2007. IEEE.

[BPW03a] Michael Backes, Birgit Pfitzmann et Michael Waidner. – A composable cryptographic library with nested operations. *In : 10th ACM conference on Computer and communication security (CCS'03)*, pp. 220–230, Washington D.C., octobre 2003. ACM.

[BPW03b] Michael Backes, Birgit Pfitzmann et Michael Waidner. – Symmetric authentication within a simulatable cryptographic library. *In : Computer Security - ESORICS 2003, 8th European Symposium on Research in Computer Security*, éd. par Einar Snekkenes et Dieter Gollman, *Lecture Notes on Computer Science*, volume 2808, pp. 271–290, Gjøovik, Norway, octobre 2003. Springer.

[BR93a] Mihir Bellare et Philip Rogaway. – Random oracles are practical : a paradigm for designing efficient protocols. *In : Computer and Communications Security (CCS'93)*, pp. 62–73. ACM Press, 1993.

[BR93b] Mihir Bellare et Phillip Rogaway. – Entity authentication and key distribution. *In : Advances in Cryptology – CRYPTO 1993*, éd. par Douglas R. Stinson, *Lecture Notes on Computer Science*, volume 773, pp. 232–249, Santa Barbara, California, août 1993. Springer.

[BR04] P. J. Broadfoot et A. W. Roscoe. – Embedding agents within the intruder to detect parallel attacks. *Journal of Computer Security*, vol. 12, n° 3/4, 2004, pp. 379–408.

[BR05] Michele Bugliesi et Sabina Rossi. – Non-interference proof techniques for the analysis of cryptographic protocols. *Journal of Computer Security*, vol. 13, n° 1, 2005, pp. 87–113.

[BR06] Mihir Bellare et Phillip Rogaway. – The security of triple encryption and a framework for code-based game-playing proofs. *In : Advances in Cryptology – Eurocrypt 2006 Proceedings*, éd. par S. Vaudenay, *Lecture Notes on Computer Science*, volume 4004, pp. 409–426, Saint Petersburg, Russia, mai 2006. Springer. Extended version available at `http://eprint.iacr.org/2004/331`.

[BU08] Michael Backes et Dominique Unruh. – Computational soundness of symbolic zero-knowledge proofs against active attackers. *In : 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pp. 255–269, Pittsburgh, PA, juin 2008. IEEE Computer Society.

[BWW00] Birgit Baum-Waidner et Michael Waidner. – Round-optimal and abuse-free optimistic multi-party contract signing. *In : Automata, Languages, and Programming, 27th International Colloquium, ICALP 2000*, éd. par Ugo Montanari, José D. P. Rolim et Emo Welzl, *Lecture Notes on Computer Science*, volume 1853, pp. 524–535, Geneva, Switzerland, juillet 2000. Springer.

[Can01] Ran Canetti. – Universally composable security : A new paradigm for cryptographic protocols. *In : Proceedings of the 42nd Symposium on Foundations of Computer Science (FOCS)*, pp. 136–145, Las Vegas, Nevada, octobre 2001. IEEE. An updated version is available at Cryptology ePrint Archive, `http://eprint.iacr.org/2000/067`.

[CC79] Patrick Cousot et Radhia Cousot. – Systematic design of program analysis frameworks. *In : 6th Annual ACM Symposium on Principles of Programming Languages*, pp. 269–282, San Antonio, Texas, 29-31 janvier 1979.

[CC05] Hubert Comon et Véronique Cortier. – Tree automata with one memory, set constraints and cryptographic protocols. *Theoretical Computer Science*, vol. 331, n° 1, février 2005, pp. 143–214.

[CCK+06] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira et Roberto Segala. – Time-bounded task-PIOAs : A framework for analyzing security protocols. *In : 20th Symposium on Distributed Computing (DISC)*, éd. par Shlomi Dolev, *Lecture Notes on Computer Science*, volume 4167, pp. 238–253, Stockholm, Sweden, septembre 2006. Springer.

[CDE04] R. Corin, J. M. Doumen et S. Etalle. – Analysing password protocol security against off-line dictionary attacks. *In : 2nd Int. Workshop on Security Issues with Petri Nets and other Computational Models (WISP)*, *Electronic Notes in Theoretical Computer Science*, juin 2004.

[CdH06] Ricardo Corin et Jerry den Hartog. – A probabilistic Hoare-style logic for game-based cryptographic proofs. *In : 33rd International Colloquium on Automata, Languages and Programming (ICALP), Track C (Security), Part II*, éd. par M. Bugliesi, B. Preneel, V. Sassone et I. Wegener, *Lecture Notes on Computer Science*, volume 4052, pp. 252–263, Venice, Italy, juillet 2006. Springer.

[CDKS00] I. Cervesato, N. Durgin, M. Kanovich et A. Scedrov. – Interpreting strands in linear logic. *In : 2000 Workshop on Formal Methods and Computer Security, 12th International Conference on Computer Aided Verification (CAV 2000) Satellite Workshop*, Chicago, Illinois, juillet 2000.

[CDL+99] I. Cervesato, N.A. Durgin, P.D. Lincoln, J.C. Mitchell et A. Scedrov. – A meta-notation for protocol analysis. *In : 12th IEEE Computer Security Foundation Workshop (CSFW-12)*, pp. 55–69, Mordano, Italy, juin 1999.

[CDL+05] Iliano Cervesato, Nancy Durgin, Patrick Lincoln, John C. Mitchell et Andre Scedrov. – A comparison between strand spaces and multiset rewriting for security protocol analysis. *Journal of Computer Security*, vol. 13, n° 2, 2005, pp. 265–316.

[CEL07] Judicaël Courant, Cristian Ene et Yassine Lakhnech. – Computationally sound typing for non-interference : The case of deterministic encryption. *In : 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, éd. par V. Arvind et Sanjiva Prasad, *Lecture Notes on Computer Science*, volume 4855, pp. 364–375, New Delhi, India, décembre 2007. Springer.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman et F. Kenneth Zadeck. – Efficiently computing static single assignment form and the control dependance graph. *ACM Transactions on Programming Languages and Systems*, vol. 13, n° 4, octobre 1991, pp. 451–490.

[CGH04] Ran Canetti, Oded Goldreich et Shai Halevi. – The random oracle methodology, revisited. *Journal of the ACM*, vol. 51, n° 4, juillet 2004, pp. 557–594.

[CH06] Ran Canetti et Jonathan Herzog. – Universally composable symbolic analysis of mutual authentication and key exchange protocols. *In : Proceedings, Theory of Cryptography Conference (TCC'06)*, éd. par Shai Halevi et Tal Rabin, *Lecture Notes on Computer Science*, volume 3876, pp. 380–403, New York, NY, mars 2006. Springer. Extended version available at `http://eprint.iacr.org/2004/334`.

[CHW06] Véronique Cortier, Heinrich Hördegen et Bogdan Warinschi. – Explicit randomness is not necessary when modeling probabilistic encryption. *In : Workshop on Information and Computer Security (ICS 2006)*, Timisoara, Romania, septembre 2006. Proceedings to appear.

[Cir01] Horatiu CIRSTEA. – Specifying authentication protocols using rewriting and strategies. *In : Practical Aspects of Declarative Languages (PADL'01)*, éd. par I.V. RAMAKRISHNAN, *Lecture Notes on Computer Science*, volume 1990, pp. 138–152, Las Vegas, Nevada, mars 2001. Springer.

[CJ97] John CLARK et Jeremy JACOB. – *A Survey of Authentication Protocol Literature : Version1.0.* – Rapport technique, University of York, Department of Computer Science, novembre 1997.

[CJM00] Edmund M. CLARKE, Somesh JHA et Will MARRERO. – Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, n° 4, 2000, pp. 443–487.

[CKKW06] Véronique CORTIER, Steve KREMER, Ralf KÜSTERS et Bogdan WARINSCHI. – Computationally sound symbolic secrecy in the presence of hash functions. *In : Proceedings of the 26th Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, éd. par Naveen GARG et S. ARUN-KUMAR, *Lecture Notes on Computer Science*, volume 4246, pp. 176–187, Kolkata, India, décembre 2006. Springer.

[CKRT03a] Yannick CHEVALIER, Ralf KÜSTERS, Michaël RUSINOWITCH et Mathieu TURUANI. – Deciding the security of protocols with Diffie-Hellman exponentiation and products in exponents. *In : FST TCS 2003 : Foundations of Software Technology and Theoretical Computer Science, 23rd Conference*, éd. par Paritosh K. PANDYA et Jaikumar RADHAKRISHNAN, *Lecture Notes on Computer Science*, volume 2914, pp. 124–135, Mumbai, India, décembre 2003. Springer.

[CKRT03b] Yannick CHEVALIER, Ralf KÜSTERS, Michaël RUSINOWITCH et Mathieu TURUANI. – An NP decision procedure for protocol insecurity with XOR. *In : 18th IEEE Symposium on Logic in Computer Science (LICS 2003)*, pp. 261–270, Ottawa, Canada, juin 2003. IEEE Computer Society.

[CKRT05] Yannick CHEVALIER, Ralf KÜSTERS, Michaël RUSINOWITCH et Mathieu TURUANI. – An NP decision procedure for protocol insecurity with XOR. *Theoretical Computer Science*, vol. 338, n° 1–3, juin 2005, pp. 247–274.

[CKS04] Rohit CHADHA, Steve KREMER et Andre SCEDROV. – Formal analysis of multiparty contract signing. *In : 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pp. 266–279, Asilomar, Pacific Grove, California, juin 2004. IEEE Computer Society.

[CLC03] Hubert COMON-LUNDH et Véronique CORTIER. – New decidability results for fragments of first-order logic and application to cryptographic protocols. *In : 14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)*, éd. par Robert NIEUWENHUIS, *Lecture Notes on Computer Science*, volume 2706, pp. 148–164, Valencia, Spain, juin 2003. Springer.

[CLC04] Hubert COMON-LUNDH et Véronique CORTIER. – Security properties : two agents are sufficient. *Science of Computer Programming*, vol. 50, n° 1–3, février 2004, pp. 51–71.

[CLS03] Hubert COMON-LUNDH et Vitaly SHMATIKOV. – Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. *In : Symposium on Logic in Computer Science (LICS'03)*, pp. 271–280, Ottawa, Canada, juin 2003. IEEE Computer Society.

[CMAFE03] Ricardo CORIN, Sreekanth MALLADI, Jim ALVES-FOSS et Sandro ETALLE. – Guess what ? here is a new tool that finds some new guessing attacks. *In : Workshop on Issues in the Theory of Security (WITS'03)*, éd. par Roberto GORRIERI, Warsaw, Poland, avril 2003.

[CMP05] Iliano CERVESATO, Catherine MEADOWS et Dusko PAVLOVIC. – An encapsulated authentication logic for reasoning about key distribution protocols. *In : Proc. 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pp. 48–61, Aix-en-Provence, France, juin 2005. IEEE Comp. Soc. Press.

[CMR01] Véronique CORTIER, Jon MILLEN et Harald RUESS. – Proving secrecy is easy enough. *In : 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pp. 97–108, Cape Breton, Nova Scotia, Canada, juin 2001. IEEE Computer Society.

[Coh02] Ernie COHEN. – Proving protocols safe from guessing. *In : Foundations of Computer Security*, Copenhagen, Denmark, juillet 2002.

[Coh03] Ernie COHEN. – First-order verification of cryptographic protocols. *Journal of Computer Security*, vol. 11, n° 2, 2003, pp. 189–216.

[Cor03] Véronique CORTIER. – *Vérification automatique des protocoles cryptographiques.* – Thèse de PhD, ENS de Cachan, mars 2003.

[Cre06] Cas J. F. CREMERS. – *Scyther - Semantics and Verification of Security Protocols.* – Ph.D. dissertation, Eindhoven University of Technology, novembre 2006.

[Cre08] Cas J. F. CREMERS. – On the Protocol Composition Logic PCL. *In : ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pp. 66–76, Tokyo, Japan, mars 2008. ACM.

[CRS05] David CHAUM, Peter Y. A. RYAN et Steve SCHNEIDER. – A practical voter-verifiable election scheme. *In : Computer Security – ESORICS 2005, 10th European Symposium on Research in Computer Security*, éd. par Sabrina De Capitani DI VIMERCATI, Paul SYVERSON et Dieter GOLLMAN, *Lecture Notes on Computer Science*, volume 3679, pp. 118–139, Milan, Italy, septembre 2005. Springer.

[CRZ07] Véronique CORTIER, Michaël RUSINOWITCH et Eugen ZĂLINESCU. – Relating two standard notions of secrecy. *Logical Methods in Computer Science*, vol. 3, n° 3, juillet 2007.

[CV01] Yannick CHEVALIER et Laurent VIGNERON. – A tool for lazy verification of security protocols. *In : 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pp. 373–376, Coronado Island, San Diego, CA, novembre 2001. IEEE Computer Society.

[CW05] Véronique CORTIER et Bogdan WARINSCHI. – Computationally sound, automated proofs for security protocols. *In : Proc. 14th European Symposium on Programming (ESOP'05)*, éd. par Mooly SAGIV, *Lecture Notes on Computer Science*, volume 3444, pp. 157–171, Edimbourg, U.K., avril 2005. Springer.

[CZ06] Véronique CORTIER et Eugen ZĂLINESCU. – Deciding key cycles for security protocols. *In : Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006*, éd. par Miki HERMANN et Andrei VORONKOV, *Lecture Notes on Computer Science*, volume 4246, pp. 317–331, Phnom Penh, Cambodia, novembre 2006. Springer.

[DDM+05] Anupam DATTA, Ante DEREK, John C. MITCHELL, Vitaly SHMATIKOV et Mathieu TURUANI. – Probabilistic polynomial-time semantics for a protocol security logic. *In : ICALP 2005 : the 32nd International Colloquium on Automata, Languages and Programming*, éd. par Luís CAIRES et Luís MONTEIRO, *Lecture Notes on Computer Science*, volume 3580, pp. 16–29, Lisboa, Portugal, juillet 2005. Springer.

[DDMP05] Anupam DATTA, Ante DEREK, John C. MITCHELL et Dusko PAVLOVIC. – A derivation system and compositional logic for security protocols. *Journal of Computer Security*, vol. 13, n° 3, 2005, pp. 423–482.

[DDMW06] Anupam Datta, Ante Derek, John C. Mitchell et Bogdan Warinschi. – Computationally sound compositional logic for key exchange protocols. *In : Proceedings of 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pp. 321–334, Venice, Italy, juillet 2006. IEEE Computer Society.

[DFG00] Antonio Durante, Riccardo Focardi et Roberto Gorrieri. – A compiler for analyzing cryptographic protocols using noninterference. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, n° 4, octobre 2000, pp. 488–528.

[DH76] W. Diffie et M. Hellman. – New directions in cryptography. *IEEE Transactions on Information Theory*, vol. IT-22, n° 6, novembre 1976, pp. 644–654.

[DJ04] Stéphanie Delaune et Florent Jacquemard. – A theory of dictionary attacks and its complexity. *In : 17th IEEE Computer Security Foundations Workshop*, pp. 2–15, Pacific Grove, CA, juin 2004. IEEE.

[DKR07] Stéphanie Delaune, Steve Kremer et Mark D. Ryan. – Symbolic bisimulation for the applied pi-calculus. *In : 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, éd. par V. Arvind et Sanjiva Prasad, *Lecture Notes on Computer Science*, volume 4855, pp. 133–145, New Delhi, India, décembre 2007. Springer.

[DLMS04] Nancy Durgin, Patrick Lincoln, John C. Mitchell et Andre Scedrov. – Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, vol. 12, n° 2, 2004, pp. 247–311.

[DM00] Grit Denker et Jonathan Millen. – CAPSL integrated protocol environment. *In : DARPA Information Survivability Conference and Exposition (DISCEX'00)*, pp. 207–221, Hilton Head, South Carolina, janvier 2000. IEEE.

[DMP03] Nancy Durgin, John C. Mitchell et Dusko Pavlovic. – A compositional logic for proving security properties of protocols. *Journal of Computer Security*, vol. 11, n° 4, 2003, pp. 677–721.

[DMT98] Grit Denker, Jose Meseguer et Carolyn Talcott. – Protocol specification and analysis in Maude. *In : Workshop on Formal Methods and Security Protocols*, éd. par N. Heintze et J. Wing, Indianapolis, Indiana, 25 juin 1998.

[DMV05] Paul Hankes Drielsma, Sebastian Mödersheim et Luca Viganò. – A formalization of off-line guessing for security protocol analysis. *In : Logic for Programming, Artificial Intelligence, and Reasoning : 11th International Conference, LPAR 2004*, éd. par Franz Baader et Andrei Voronkov, *Lecture Notes on Computer Science*, volume 3452, pp. 363–379, Montevideo, Uruguay, mars 2005. Springer.

[dN95] Hans de Nivelle. – *Ordering Refinements of Resolution.* – Thèse de PhD, Technische Universiteit Delft, octobre 1995.

[DR06] Tim Dierks et Eric Rescorla. – RFC 4346 : The Transport Layer Security (TLS) protocol, version 1.1. – avril 2006. `http://tools.ietf.org/html/rfc4346`.

[DS81] Dorothy E. Denning et Giovanni Maria Sacco. – Timestamps in key distribution protocols. *Communications of the ACM*, vol. 24, n° 8, août 1981, pp. 533–536.

[DSV03] Luca Durante, Riccardo Sisto et Adriano Valenzano. – Automatic testing equivalence verification of spi calculus specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, n° 2, avril 2003, pp. 222–284.

[DY83] Danny Dolev et Andrew C. Yao. – On the security of public key protocols. *IEEE Transactions on Information Theory*, vol. IT-29, n° 12, mars 1983, pp. 198–208.

[EMM06] Santiago ESCOBAR, Catherine MEADOWS et José MESEGUER. – A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, vol. 367, n° 1-2, 2006, pp. 162–202.

[Fer05] Jérôme FERET. – *Analysis of mobile systems by abstract interpretation.* – Thèse de PhD, École Polytechnique, février 2005.

[FGM00] Riccardo FOCARDI, Roberto GORRIERI et Fabio MARTINELLI. – Non interference for the analysis of cryptographic protocols. *In : Automata, Languages and Programming, 27th International Colloquium, ICALP'00*, éd. par Ugo MONTANARI, José D. P. ROLIM et Emo WELZL, *Lecture Notes on Computer Science*, volume 1853, pp. 354–372, Geneva, Switzerland, juillet 2000. Springer.

[FHG99] F. Javier Thayer FÁBREGA, Jonathan C. HERZOG et Joshua D. GUTTMAN. – Strand spaces : Proving security protocols correct. *Journal of Computer Security*, vol. 7, n° 2/3, 1999, pp. 191–230.

[GJ01] Andrew GORDON et Alan JEFFREY. – Authenticity by typing for security protocols. *In : 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pp. 145–159, Cape Breton, Nova Scotia, Canada, juin 2001. IEEE Computer Society.

[GJ02] Andrew GORDON et Alan JEFFREY. – Typing one-to-one and one-to-many correspondences in security protocols. *In : Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium, ISSS 2002*, éd. par M. OKADA, B. PIERCE, A. SCEDRIV, H. TOKUDA et A. YONEZAWA, *Lecture Notes on Computer Science*, volume 2609, pp. 263–282, Tokyo, Japan, novembre 2002. Springer.

[GJ03] Andrew GORDON et Alan JEFFREY. – Authenticity by typing for security protocols. *Journal of Computer Security*, vol. 11, n° 4, 2003, pp. 451–521.

[GJ04] Andrew GORDON et Alan JEFFREY. – Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, vol. 12, n° 3/4, 2004, pp. 435–484.

[GK00] Thomas GENET et Francis KLAY. – Rewriting for cryptographic protocol verification. *In : 17th International Conference on Automated Deduction (CADE-17)*, éd. par D. MCALLESTER, *Lecture Notes on Computer Science*, volume 1831, pp. 271–290, Pittsburgh, PA, juin 2000. Springer.

[GL00] Jean GOUBAULT-LARRECQ. – A method for automatic cryptographic protocol verification (extended abstract), invited paper. *In : Fifth International Workshop on Formal Methods for Parallel Programming : Theory and Applications (FMPP-TA'2000)*, éd. par J. ROLIM et OTHERS, *Lecture Notes on Computer Science*, volume 1800, pp. 977–984, Cancún, Mexique, mai 2000. Springer.

[GL05] Jean GOUBAULT-LARRECQ. – Deciding $\langle_1$ by resolution. *Information Processing Letters*, vol. 95, n° 3, août 2005, pp. 401–408.

[GL08] Jean GOUBAULT-LARRECQ. – Towards producing formally checkable security proofs, automatically. *In : 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pp. 224–238, Pittsburgh, PA, juin 2008. IEEE Computer Society.

[GLP05] Jean GOUBAULT-LARRECQ et Fabrice PARRENNES. – Cryptographic protocol analysis on real C code. *In : Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, éd. par Radhia COUSOT, *Lecture Notes on Computer Science*, volume 3385, pp. 363–379, Paris, France, janvier 2005. Springer.

[GM84] Shafi GOLDWASSER et Silvio MICALI. – Probabilistic encryption. *Journal of Computer and System Sciences*, vol. 28, 1984, pp. 270–299.

[GM99] Juan A. GARAY et Philip MACKENZIE. – Abuse-free multi-party contract signing. *In : Distributed Computing : 13th International Symposium, DISC'99*, éd. par

Prasad Jayanti, *Lecture Notes on Computer Science*, volume 1693, pp. 151–165, Bratislava, Slovak Republic, septembre 1999. Springer.

[GM03] Roberto Gorrieri et Fabio Martinelli. – Process algebraic frameworks for the specification and analysis of cryptographic protocols. *In : Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003*, éd. par Branislav Rovan et Peter Vojtáš, *Lecture Notes on Computer Science*, volume 2747, pp. 46–67, Bratislava, Slovakia, août 2003. Springer.

[GMP05] Alexey Gotsman, Fabio Massacci et Marco Pistore. – Towards an independent semantics and verification technology for the HLPSL specification language. *Electronic Notes in Theoretical Computer Science*, vol. 135, n° 1, juillet 2005, pp. 59–77.

[GMR88] Shafi Goldwasser, Silvio Micali et Ronald Rivest. – A digital signature scheme secure against adaptative chosen-message attacks. *SIAM Journal of Computing*, vol. 17, n° 2, avril 1988, pp. 281–308.

[GNY90] Li Gong, Roger Needham et Raphael Yahalom. – Reasoning about belief in cryptographic protocols. *In : Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pp. 234–248, Oakland, California, mai 1990. IEEE Computer Society.

[God06] Jens Chr. Godskesen. – Formal verification of the aran protocol using the applied pi-calculus. *In : Proceedings of the Sixth International IFIP WG 1.7 Workshop on Issues in the Theory of Security (WITS'06)*, pp. 99–113, Vienna, Austria, mars 2006.

[Hal05] Shai Halevi. – A plausible approach to computer-aided cryptographic proofs. – Cryptology ePrint Archive, Report 2005/181, juin 2005. Available at `http://eprint.iacr.org/2005/181`.

[Her03] Jonathan Herzog. – A computational interpretation of Dolev-Yao adversaries. *In : WITS'03 - Workshop on Issues in the Theory of Security*, éd. par Roberto Gorrieri, pp. 146–155, Warsaw, Poland, avril 2003.

[HLM03] Jonathan Herzog, Moses Liskov et Silvio Micali. – Plaintext awareness via key registration. *In : Advances in Cryptology – CRYPTO 2003*, éd. par Dan Boneh, *Lecture Notes on Computer Science*, volume 2729, pp. 548–564, Santa Barbara, California, août 2003. Springer.

[HLS00] James Heather, Gavin Lowe et Steve Schneider. – How to prevent type flaw attacks on security protocols. *In : 13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pp. 255–268, Cambridge, England, juillet 2000.

[HS05] James Heather et Steve Schneider. – A decision procedure for the existence of a rank function. *Journal of Computer Security*, vol. 13, n° 2, 2005, pp. 317–344.

[Hüt02] Hans Hüttel. – Deciding framed bisimilarity. *In : 4th International Workshop on Verification of Infinite-State Systems (INFINITY'02)*, pp. 1–20, Brno, Czech Republic, août 2002.

[IEE99] IEEE Computer Society. – *IEEE Standard 802.11 : IEEE Standard for Information technology–Telecommunications and information exchange between system–Local and metropolitan area networks–Specific requirements–Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[IEE04] IEEE Computer Society. – *IEEE Standard 802.11i : IEEE Standard for Information technology–Telecommunications and information exchange between system–Local and metropolitan area networks–Specific requirements–Part 11 : Wireless*

LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 6 : Medium Access Control (MAC) Security Enhancements, 2004.

[IET06] IETF. – Public key cryptography for initial authentication in Kerberos, 1996–2006. RFC 4556. Preliminary versions available as a sequence of Internet Drafts at `http://tools.ietf.org/wg/krb-wg/draft-ietf-cat-kerberos-pk-init/`.

[JLM05] Romain Janvier, Yassine Lakhnech et Laurent Mazaré. – Completing the picture : Soundness of formal encryption in the presence of active adversaries. *In : Proc. 14th European Symposium on Programming (ESOP'05)*, éd. par Mooly Sagiv, *Lecture Notes on Computer Science*, volume 3444, pp. 172–185, Edimbourg, U.K., avril 2005. Springer.

[JLM06] Romain Janvier, Yassine Lakhnech et Laurent Mazaré. – Relating the symbolic and computational models of security protocols using hashes. *In : Proceedings of the Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA'06)*, éd. par Pierpaolo Degano, Ralf Küsters, Luca Viganò et Steve Zdancewic, pp. 67–89, Seattle, Washington, août 2006.

[Kau05] Charlie Kaufman. – RFC 4306 : Internet Key Exchange (IKEv2) Protocol. – décembre 2005. `http://www.ietf.org/rfc/rfc4306.txt`.

[KB70] Donald E. Knuth et Peter B. Bendix. – Simple word problems in universal algebras. *In : Computational Problems in Abstract Algebra*, éd. par J. Leech, pp. 263–297. – Oxford, U.K., Pergamon Press, 1970.

[KH06] Himanshu Khurana et Hyung-Seok Hahm. – Certified mailing lists. *In : Proceedings of the ACM Symposium on Communication, Information, Computer and Communication Security (ASIACCS'06)*, pp. 46–58, Taipei, Taiwan, mars 2006. ACM.

[KR04] Steve Kremer et Mark D. Ryan. – Analysing the vulnerability of protocols to produce known-pair and chosen-text attacks. *In : Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SecCo 2004)*, éd. par R. Focardi et G. Zavattaro, *Electronic Notes in Theoretical Computer Science*, volume 128(5), pp. 87–104, août 2004.

[KR05] Steve Kremer et Mark D. Ryan. – Analysis of an electronic voting protocol in the applied pi calculus. *In : Programming Languages and Systems : 14th European Symposium on Programming, ESOP 2005*, éd. par Mooly Sagiv, *Lecture Notes on Computer Science*, volume 3444, pp. 186–200, Edimbourg, UK, avril 2005. Springer.

[Kra96] Hugo Krawczyk. – SKEME : A versatile secure key exchange mechanism for internet. *In : Internet Society Symposium on Network and Distributed Systems Security*, février 1996. Available at `http ://bilbo.isu.edu/sndss/sndss96.html`.

[KRS+03] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang et Kvin Fu. – Plutus : Scalable secure file sharing on untrusted storage. *In : 2nd Conference on File and Storage Technologies (FAST'03)*, pp. 29–42, San Francisco, CA, avril 2003. Usenix.

[KW96] Darell Kindred et Jeannette M. Wing. – Fast, automatic checking of security protocols. *In : USENIX 2nd Workshop on Electronic Commerce*, pp. 41–52, novembre 1996.

[Lau03] Peeter Laud. – Handling encryption in an analysis for secure information flow. *In : Programming Languages and Systems, 12th European Symposium on Programming, ESOP'03*, éd. par Pierpaolo Degano, *Lecture Notes on Computer Science*, volume 2618, pp. 159–173, Warsaw, Poland, avril 2003. Springer.

[Lau04] Peeter Laud. – Symmetric encryption in automatic analyses for confidentiality against active adversaries. *In : IEEE Symposium on Security and Privacy*, pp. 71–85, Oakland, California, mai 2004.

[Lau05] Peeter Laud. – Secrecy types for a simulatable cryptographic library. *In : 12th ACM Conference on Computer and Communications Security (CCS'05)*, pp. 26–35, Alexandria, VA, novembre 2005. ACM.

[LMBG05] Kevin D. Lux, Michael J. May, Nayan L. Bhattad et Carl A. Gunter. – WSEmail : Secure internet messaging based on web services. *In : International Conference on Web Services (ICWS'05)*, pp. 75–82, Orlando, Florida, juillet 2005. IEEE Computer Society.

[LMMS98] P. D. Lincoln, J. C. Mitchell, M. Mitchell et A. Scedrov. – A probabilistic poly-time framework for protocol analysis. *In : ACM Computer and Communication Security (CCS-5)*, pp. 112–121, San Francisco, California, novembre 1998.

[LMMS99] P. D. Lincoln, J. C. Mitchell, M. Mitchell et A. Scedrov. – Probabilistic polynomial-time equivalence and security protocols. *In : FM'99 World Congress On Formal Methods in the Development of Computing Systems*, éd. par J.M. Wing, J. Woodcock et J. Davies, *Lecture Notes on Computer Science*, volume 1708, pp. 776–793, Toulouse, France, septembre 1999. Springer.

[Low96] Gavin Lowe. – Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *In : Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes on Computer Science*, volume 1055, pp. 147–166. Springer, 1996.

[Low97] Gavin Lowe. – A hierarchy of authentication specifications. *In : 10th Computer Security Foundations Workshop (CSFW '97)*, pp. 31–43, Rockport, Massachusetts, juin 1997. IEEE Computer Society.

[Low02] Gavin Lowe. – Analyzing protocols subject to guessing attacks. *In : Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, janvier 2002.

[LV05] Peeter Laud et Varmo Vene. – A type system for computationally secure information flow. *In : 15th International Symposium on Fundamentals of Computation Theory (FCT'05)*, éd. par Maciej Liśkiewicz et Rüdiger Reischuk, *Lecture Notes on Computer Science*, volume 3623, pp. 365–377, Lübeck, Germany, août 2005. Springer.

[Lyn97] Christopher Lynch. – Oriented equational logic programming is complete. *Journal of Symbolic Computation*, vol. 21, n° 1, 1997, pp. 23–45.

[MCF87] Jonathan K. Millen, Sidney C. Clark et Sheryl B. Freedman. – The Interrogator : Protocol security analysis. *IEEE Transactions on Software Engineering*, vol. SE-13, n° 2, février 1987, pp. 274–288.

[Mea96] Catherine A. Meadows. – The NRL protocol analyzer : An overview. *Journal of Logic Programming*, vol. 26, n° 2, 1996, pp. 113–131.

[Mil95] Jonathan K. Millen. – The Interrogator model. *In : 1995 IEEE Symposium on Security and Privacy*, pp. 251–260, Oakland, California, mai 1995. IEEE Computer Society Press.

[Mil99] Jonathan Millen. – A necessarily parallel attack. *In : Workshop on Formal Methods and Security Protocols (FMSP'99)*, Trento, Italy, juillet 1999.

[Mil02] Giuseppe Milicia. – χ-spaces : Programming security protocols. *In : Proceedings of the 14th Nordic Workshop on Programming Theory (NWPT'02)*, Tallinn, Estonia, novembre 2002.

[MMS97] John C. Mitchell, Mark Mitchell et Ulrich Stern. – Automated analysis of cryptographic protocols using Murφ. *In : 1997 IEEE Symposium on Security and Privacy*, pp. 141–151, 1997.

[MMS03] P. Mateus, J. Mitchell et A. Scedrov. – Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. *In : CONCUR 2003 - Concurrency Theory, 14-th International Conference*, éd. par R. Amadio et D. Lugiez, *Lecture Notes on Computer Science*, volume 2761, pp. 327–349, Marseille, France, septembre 2003. Springer.

[MN02] Cathy Meadows et Paliath Narendran. – A unification algorithm for the group Diffie-Hellman protocol. *In : Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, janvier 2002.

[Mon99] David Monniaux. – Decision procedures for the analysis of cryptographic protocols by logics of belief. *In : 12th Computer Security Foundations Workshop*, pp. 44–54, Mordano, Italy, juin 1999. IEEE.

[Mon03] David Monniaux. – Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, vol. 47, n° 2–3, 2003, pp. 177–202.

[MPW92] Robin Milner, Joachim Parrow et David Walker. – A calculus of mobile processes, parts I and II. *Information and Computation*, vol. 100, septembre 1992, pp. 1–40 and 41–77.

[MR06] Aybek Mukhamedov et Mark Ryan. – Resolve-impossibility for a contract-signing protocol. *In : 19th Computer Security Foundations Workshop (CSFW'06)*, pp. 167–176, Venice, Italy, juillet 2006. IEEE Computer Society.

[MRST06] John C. Mitchell, Ajith Ramanathan, Andre Scedrov et V. Teague. – A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, vol. 353, n° 1–3, mars 2006, pp. 118–164.

[MS01] Jonathan Millen et Vitaly Shmatikov. – Constraint solving for bounded-process cryptographic protocol analysis. *In : Proc. 8th ACM Conference on Computer and Communications Security (CCS '01)*, pp. 166–175, 2001.

[MW04a] Daniele Micciancio et Bogdan Warinschi. – Completeness theorems for the Abadi-Rogaway logic of encrypted expressions. *Journal of Computer Security*, vol. 12, n° 1, 2004, pp. 99–129.

[MW04b] Daniele Micciancio et Bogdan Warinschi. – Soundness of formal encryption in the presence of active adversaries. *In : Theory of Cryptography Conference (TCC'04)*, éd. par Moni Naor, *Lecture Notes on Computer Science*, volume 2951, pp. 133–151, Cambridge, MA, USA, février 2004. Springer.

[NS78] Roger M. Needham et Michael D. Schroeder. – Using encryption for authentication in large networks of computers. *Communications of the ACM*, vol. 21, n° 12, décembre 1978, pp. 993–999.

[NS87] Roger M. Needham et Michael D. Schroeder. – Authentication revisited. *Operating Systems Review*, vol. 21, n° 1, 1987, p. 7.

[NYHR05] Clifford Neuman, Tom Yu, Sam Hartman et Kenneth Raeburn. – The Kerberos network authentication service (V5), juillet 2005. `http://www.ietf.org/rfc/rfc4120`.

[OR87] Dave Otway et Owen Rees. – Efficient and timely mutual authentication. *Operating Systems Review*, vol. 21, n° 1, 1987, pp. 8–10.

[Pau98] Larry C. Paulson. – The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, vol. 6, n° 1–2, 1998, pp. 85–128.

[Pot02] François Pottier. – A simple view of type-secure information flow in the π-calculus. *In : Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pp. 320–330, Cape Breton, Nova Scotia, juin 2002.

[PS02] François Pottier et Vincent Simonet. – Information flow inference for ML. *In : Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pp. 319–330, Portland, Oregon, janvier 2002.

[PS07] Erik Poll et Aleksy Schubert. – Verifying an implementation of SSH. *In : 7th International Workshop on Issues in the Theory of Security (WITS'07)*, Braga, Portugal, mars 2007.

[PSD04] Davide Pozza, Riccardo Sisto et Luca Durante. – Spi2Java : Automatic cryptographic protocol Java code generation from spi calculus. *In : 18th International Conference on Advanced Information Networking and Applications (AINA'04)*, volume 1, pp. 400–405, Fukuoka, Japan, mars 2004. IEEE Computer Society.

[RB99] A. W. Roscoe et P. J. Broadfoot. – Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, vol. 7, n° 2, 3, 1999, pp. 147–190.

[RMST04] Ajith Ramanathan, John Mitchell, Andre Scedrov et Vanessa Teague. – Probabilistic bisimulation and equivalence for security analysis of network protocols. *In : FOSSACS 2004 - Foundations of Software Science and Computation Structures*, éd. par I. Walukiewicz, *Lecture Notes on Computer Science*, volume 2987, pp. 468–483, Barcelona, Spain, mars 2004. Springer.

[RS03] R. Ramanujam et S.P. Suresh. – Tagging makes secrecy decidable with unbounded nonces as well. *In : FST TCS 2003 : Foundations of Software Technology and Theoretical Computer Science*, éd. par P.K. Pandya et J. Radhakrishnan, *Lecture Notes on Computer Science*, volume 2914, pp. 363–374, Mumbai, India, décembre 2003. Springer.

[RSA78] Ronald Rivest, Adi Shamir et Leonard Adleman. – A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, vol. 21, n° 2, février 1978, pp. 120–126.

[RT03] Michaël Rusinowitch et Mathieu Turuani. – Protocol insecurity with finite number of sessions is NP-complete. *Theoretical Computer Science*, vol. 299, n° 1–3, avril 2003, pp. 451–475.

[SA06] Geoffrey Smith et Rafael Alpízar. – Secure information flow with random assignment and encryption. *In : 4th ACM Workshop on Formal Methods in Security Engineering (FMSE'06)*, pp. 33–43, Alexandria, Virginia, novembre 2006.

[SBB+06] Christoph Sprenger, Michael Backes, David Basin, Birgit Pfitzmann et Michael Waidner. – Cryptographically sound theorem proving. *In : 19th IEEE Computer Security Foundations Workshop (CSFW-19)*, pp. 153–166, Venice, Italy, juillet 2006. IEEE.

[SBP01] Dawn Xiaodong Song, Sergey Berezin et Adrian Perrig. – Athena : a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, vol. 9, n° 1/2, 2001, pp. 47–74.

[Sch96] Bruce Schneier. – *Applied Cryptography, Second Edition*. – John Wiley & Sons, 1996.

[Sho01] Victor Shoup. – A proposal for an ISO standard for public-key encryption, décembre 2001. ISO/IEC JTC 1/SC27.

[Sho02] Victor Shoup. – OAEP reconsidered. *Journal of Cryptology*, vol. 15, n° 4, septembre 2002, pp. 223–249.

[Sho04] Victor Shoup. – Sequences of games : a tool for taming complexity in security proofs. – Cryptology ePrint Archive, Report 2004/332, novembre 2004. Available at `http://eprint.iacr.org/2004/332`.

[SPP01] Dawn Song, Adrian Perrig et Doantam Phan. – AGVI—Automatic Generation, Verification, and Implementation of security protocols. *In : Computer Aided Verification (CAV'01)*, éd. par Gérard Berry, Hubert Comon et Alain Finkel, *Lecture Notes on Computer Science*, volume 2102, pp. 241–245, Paris, France, juillet 2001. Springer.

[Sti05] Douglas R. Stinson. – *Cryptography : Theory and Practice, Third Edition.* – CRC Press, novembre 2005.

[SvO94] Paul Syverson et Paul C. van Oorschot. – On unifying some cryptographic protocol logics. *In : Proceedings 1994 IEEE Symposium on Research in Security and Privacy*, pp. 14–28, Oakland, California, mai 1994. IEEE Computer Society.

[Tar05] Sabrina Tarento. – Machine-checked security proofs of cryptographic signature schemes. *In : Proceedings of the 10th European Symposium On Research In Computer Security (ESORICS 2005)*, éd. par Sabrina de Capitani di Vimercati, Paul Syverson et Dieter Gollmann, *Lecture Notes on Computer Science*, volume 3679, pp. 140–158, Milan, Italy, septembre 2005. Springer.

[TL07] Ilja Tšahhirov et Peeter Laud. – Application of dependency graphs to security protocol analysis. *In : 3rd Symposium on Trustworthy Global Computing (TGC'07)*, éd. par Gilles Barthe et Cédric Fournet, *Lecture Notes on Computer Science*, volume 4912, Sophia-Antipolis, France, novembre 2007. Springer.

[Wei99] Christoph Weidenbach. – Towards an automatic analysis of security protocols in first-order logic. *In : 16th International Conference on Automated Deduction (CADE-16)*, éd. par Harald Ganzinger, *Lecture Notes in Artificial Intelligence*, volume 1632, pp. 314–328, Trento, Italy, juillet 1999. Springer.

[WFA] Wi-Fi Alliance. – *Wi-Fi Protected Access (WPA)*. `http://www.wi-fi.org`.

[WL92] Thomas Y. C. Woo et Simon S. Lam. – Authentication for distributed systems. *Computer*, vol. 25, n° 1, janvier 1992, pp. 39–52.

[WL93] Thomas Y. C. Woo et Simon S. Lam. – A semantic model for authentication protocols. *In : Proceedings IEEE Symposium on Research in Security and Privacy*, pp. 178–194, Oakland, California, mai 1993.

[WL97] Thomas Y. C. Woo et Simon S. Lam. – Authentication for distributed systems. *In : Internet Besieged : Countering Cyberspace Scofflaws*, éd. par Dorothy Denning et Peter Denning, pp. 319–355. ACM Press and Addison-Wesley, octobre 1997.

[WS96] David Wagner et Bruce Schneier. – Analysis of the SSL 3.0 protocol. *In : The Second USENIX Workshop on Electronic Commerce Proceedings*, pp. 29–40, Oakland, California, novembre 1996. USENIX Press.

[WY05] Xiaoyun Wang et Hongbo Yu. – How to break md5 and other hash functions. *In : Advances in Cryptology – EUROCRYPT 2005*, éd. par Ronald Cramer, *Lecture Notes on Computer Science*, volume 3494, pp. 19–35, Aarhus, Denmark, mai 2005. Springer.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin et Hongbo Yu. – Finding collisions in the full SHA-1. *In : Advances in Cryptology – CRYPTO 2005*, éd. par Victor Shoup, *Lecture Notes on Computer Science*, volume 3621, pp. 17–36, Santa Barbara, California, août 2005. Springer.

[Yao82] Andrew C. YAO. – Theory and applications of trapdoor functions. *In : Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS'82)*, pp. 80–91, 1982.

[Ylö06] Tatu YLÖNEN. – The Secure Shell (SSH) protocol architecture. – janvier 2006. `http://tools.ietf.org/html/rfc4251`.

Annexe A

Curriculum vitae

État civil

Bruno Blanchet
École normale supérieure, département d'informatique
45, rue d'Ulm
75005 Paris, France
Courrier électronique : `Bruno.Blanchet@ens.fr`

Né le 9 avril 1974, célibataire, nationalité française.

Expérience professionnelle

Depuis 2001
: Chargé de recherche au CNRS, affecté au laboratoire d'informatique de l'École normale supérieure, Paris, France (1re classe depuis 2005).

Formation

1997-1998 et 1999-2000
: Thèse avec Alain Deutsch à l'INRIA Rocquencourt sous la direction de Patrick Cousot, soutenue le 7 décembre 2000, à l'École polytechnique,
mention Très honorable avec félicitations, prix de thèse de l'École polytechnique.
Sujet : *Analyse d'échappement. Applications à ML et Java^{TM}.*

1998-1999
: Service national, scientifique du contingent au laboratoire d'informatique de l'École polytechnique.

1994-1998
: Élève de l'École normale supérieure (reçu 5e au concours d'entrée).

- 97-98 Première année de thèse.
- 96-97 Agrégation de mathématiques, rang 24e.
- 95-96 DEA de Sémantique, Preuves et Programmation, mention Très bien, rang 1er.
- 94-95 Licence et maîtrise d'Informatique, mention Très bien.

Séjours dans des laboratoires étrangers

Nov. 2001-août 2004
: Chef d'un groupe de recherche indépendant (*Nachwuchsgruppenleiter*), Max-Planck-Institut für Informatik, Sarrebruck, Allemagne, environ 2 à 3 semaines par mois au Max-Planck-Institut.

Août-oct. 2000
: Stage sous la direction de Martín Abadi, Bell Labs Research, Palo Alto, USA.
Sujet : *vérification de protocoles cryptographiques.*

Avril 2000
: Stage dans l'équipe de Reinhard Wilhelm, Université de la Sarre, Sarrebruck, Allemagne.
Sujet : *codage en ligne d'objets en Java.*

Activité éditoriale

Éditeur associé à l'International Journal of Applied Cryptography (IJACT), depuis 2006.
Membre des comités de programme de

2009 IEEE Computer Security Foundations Symposium (CSF'09)
ACM SIGPLAN Conference on Principles of Programming Languages (POPL'09)
2008 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'08)
IEEE Computer Security Foundations Symposium (CSF'08)
Workshop on Formal and Computational Cryptography (FCC'08), PC co-chair
2007 Workshop on Formal and Computational Cryptography (FCC'07)
Concurrency Theory (CONCUR'07)
IEEE Computer Security Foundations Symposium (CSF'07)
ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)
2006 Workshop on Formal and Computational Cryptography (FCC'06)
IEEE Computer Security Foundations Workshop (CSFW'06)
Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)
Foundations of Software Science and Computation Structures (FOSSACS'06)
2005 Mobile Code Safety and Program Verification Using Computational Logic Tools (MoveLog'05)
IEEE Computer Security Foundations Workshop (CSFW'05)
Abstract Interpretation for Object Oriented Languages (AIOOL'05)
European Symposium on Programming (ESOP'05)
2004 Concurrency Theory (CONCUR'04)
ACM Symposium on Applied Computing (SAC'04) Security Track
2003 International Workshop in Formal Methods (IWFM'03)

Commission de spécialistes

Membre de la commission de spécialistes informatique de l'École normale supérieure de Cachan, depuis 2005.

Jurys de thèses

2007 Mathieu Baudet, École normale supérieure de Cachan (examinateur)
Sécurité des protocoles cryptographiques : aspects logiques et calculatoires.

2005 Zhang Yu, École normale supérieure de Cachan (examinateur)
Relations logiques cryptographiques — Qu'est-ce que l'équivalence contextuelle des protocoles cryptographiques et comment la prouver ?

Encadrement de la recherche

2006 Maël Primet, stage long, École normale supérieure.
Vérification d'implantations de protocoles cryptographiques en Java.
2005 Yannick Gérault, stage scientifique, École polytechnique.
Analyse de protocoles cryptographiques définis par une suite de messages.
2004 Xavier Allamigeon, stage scientifique, École polytechnique.
Reconstruction d'attaques contre des protocoles cryptographiques.
2003 Mehmet Kiraz, stage de master d'informatique, Université de la Sarre.
Formalisation et vérification de descriptions informelles de protocoles cryptographiques.
2002 Emma Rabbidge.
Implantation d'un front-end *pour l'analyse du* bytecode *Java.*
Shiv Pratap Raghuwanshi, stage d'été, IIT Kanpur.
Implantation en Java du protocole SSH.

Enseignement

2007-2008 Cours “Protocoles cryptographiques : preuves formelles et calculatoires” avec Steve Kremer, au Master Parisien de Recherche en Informatique (MPRI), 12 heures
2003-2007 Intervention sur la vérification de protocoles cryptographiques dans le cours d'analyse statique de Patrick et Radhia Cousot au DEA de Programmation : Sémantique, Preuves et Langages, devenu Master Parisien de Recherche en Informatique (MPRI), 6 heures/an.
1999-2002 Intervention sur l'analyse d'échappement dans le cours d'analyse statique de Radhia Cousot au DEA Sémantique, Preuves et Programmation, 6 heures/an.
1999-2001 Travaux dirigés d'informatique à l'Université de Versailles, 64 heures/an.
1997-1998 Travaux dirigés d'informatique à l'ENSTA, 15 heures.
1996-1999 Travaux dirigés d'informatique à l'École polytechnique, 48 heures/an.

Publications

Revues internationales à comité de lecture

[1] Bruno Blanchet. – Automatic Verification of Correspondences for Security Protocols. *Journal of Computer Security.* À paraître.

[2] Bruno Blanchet. – A Computationally Sound Mechanized Prover for Security Protocols. *IEEE Transactions on Dependable and Secure Computing*, vol. 5, n° 4, octobre-décembre 2008, pp. 193–207.

[3] Bruno Blanchet, Martín Abadi et Cédric Fournet. – Automated Verification of Selected Equivalences for Security Protocols. *Journal of Logic and Algebraic Programming*, vol. 75, n° 1, février-mars 2008, pp. 3–51.

[4] Martín Abadi, Bruno Blanchet et Cédric Fournet. – Just Fast Keying in the Pi Calculus. *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, n° 3, juillet 2007, pp. 1–59.

[5] Martín Abadi et Bruno Blanchet. – Computer-Assisted Verification of a Protocol for Certified Email. *Science of Computer Programming*, vol. 58, n° 1-2, octobre 2005, pp. 3–27.

[6] Bruno BLANCHET. – Security Protocols : From Linear to Classical Logic by Abstract Interpretation. *Information Processing Letters*, vol. 95, n° 5, septembre 2005, pp. 473–479.

[7] Bruno BLANCHET et Andreas PODELSKI. – Verification of Cryptographic Protocols : Tagging Enforces Termination. *Theoretical Computer Science*, vol. 333, n° 1-2, mars 2005, pp. 67–90.

[8] Martín ABADI et Bruno BLANCHET. – Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, vol. 52, n° 1, janvier 2005, pp. 102–146.

[9] Bruno BLANCHET. – Escape Analysis for Java™. Theory and Practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, n° 6, novembre 2003, pp. 713–775.

[10] Martín ABADI et Bruno BLANCHET. – Secrecy Types for Asymmetric Communication. *Theoretical Computer Science*, vol. 298, n° 3, avril 2003, pp. 387–415.

Conférences invitées dans des congrès internationaux

[11] Bruno BLANCHET. – An Automatic Security Protocol Verifier based on Resolution Theorem Proving (tutorial invité). *In : 20th International Conference on Automated Deduction (CADE-20)*, Tallinn, Estonie, juillet 2005.

[12] Bruno BLANCHET. – Automatic Verification of Cryptographic Protocols : A Logic Programming Approach. *In : 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 1–3, Uppsala, Suède, août 2003. ACM.

[13] Bruno BLANCHET. – Abstracting Cryptographic Protocols by Prolog Rules. *In : 8th International Static Analysis Symposium (SAS'01)*, éd. par Patrick COUSOT, *Lecture Notes on Computer Science*, volume 2126, pp. 433–436, Paris, France, juillet 2001. Springer.

Actes de colloques internationaux à comité de programme

[14] Bruno BLANCHET et Avik CHAUDHURI. – Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage. *In : IEEE Symposium on Security and Privacy*, pp. 417–431, Oakland, CA, mai 2008. IEEE.

[15] Bruno BLANCHET, Aaron D. JAGGARD, Andre SCEDROV et Joe-Kai TSAY. – Computationally Sound Mechanized Proofs for Basic and Public-Key Kerberos. *In : ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pp. 87–99, Tokyo, Japon, mars 2008. ACM.

[16] Bruno BLANCHET. – Computationally Sound Mechanized Proofs of Correspondence Assertions. *In : 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 97–111, Venise, Italie, juillet 2007. IEEE.

[17] Bruno BLANCHET et David POINTCHEVAL. – Automated Security Proofs with Sequences of Games. *In : Advances in Cryptology – CRYPTO'06*, éd. par Cynthia Dwork, *Lecture Notes on Computer Science*, volume 4117, pp. 537-554, Santa Barbara, Californie, août 2006. Springer.

[18] Bruno BLANCHET. – A Computationally Sound Mechanized Prover for Security Protocols. *In : IEEE Symposium on Security and Privacy*, pp. 140-154, Oakland, Californie, mai 2006. IEEE Computer Society.

[19] Bruno BLANCHET, Martín ABADI et Cédric FOURNET. – Automated Verification of Selected Equivalences for Security Protocols. *In : 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pp. 331–340, Chicago, Illinois, juin 2005.

[20] Xavier Allamigeon et Bruno Blanchet. – Reconstruction of Attacks against Cryptographic Protocols. *In : 18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pp. 140–154, Aix-en-Provence, France, juin 2005.

[21] Bruno Blanchet. – Automatic Proof of Strong Secrecy for Security Protocols. *In : IEEE Symposium on Security and Privacy*, pp. 86–100, Oakland, Californie, mai 2004.

[22] Martín Abadi, Bruno Blanchet et Cédric Fournet. – Just Fast Keying in the Pi Calculus. *In : Programming Languages and Systems : 13th European Symposium on Programming (ESOP'04)*, éd. par David Schmidt, *Lecture Notes on Computer Science*, volume 2986, pp. 340–354, Barcelone, Espagne, mars 2004. Springer.

[23] Bruno Blanchet et Benjamin Aziz. – A Calculus for Secure Mobility. *In : Eighth Asian Computing Science Conference (ASIAN'03)*, éd. par Vijay Saraswat, *Lecture Notes on Computer Science*, volume 2896, pp. 188–204, Mumbai, Inde, décembre 2003. Springer.

[24] Martín Abadi et Bruno Blanchet. – Computer-Assisted Verification of a Protocol for Certified Email. *In : Static Analysis, 10th International Symposium (SAS'03)*, éd. par Radhia Cousot, *Lecture Notes on Computer Science*, volume 2694, pp. 316–335, San Diego, Californie, juin 2003. Springer.

[25] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival. – A Static Analyzer for Large Safety-Critical Software. *In : ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pp. 196–207, San Diego, Californie, juin 2003. ACM.

[26] Bruno Blanchet et Andreas Podelski. – Verification of Cryptographic Protocols : Tagging Enforces Termination. *In : Foundations of Software Science and Computation Structures (FoSSaCS'03)*, éd. par Andrew Gordon, *Lecture Notes on Computer Science*, volume 2620, pp. 136–152, Varsovie, Pologne, avril 2003. Springer.

[27] Bruno Blanchet. – From Secrecy to Authenticity in Security Protocols. *In : 9th International Static Analysis Symposium (SAS'02)*, éd. par Manuel Hermenegildo et Germán Puebla, *Lecture Notes on Computer Science*, volume 2477, pp. 342–359, Madrid, Espagne, septembre 2002. Springer.

[28] Martín Abadi et Bruno Blanchet. – Analyzing Security Protocols with Secrecy Types and Logic Programs. *In : 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pp. 33–44, Portland, Oregon, janvier 2002. ACM Press.

[29] Bruno Blanchet. – An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. *In : 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pp. 82–96, Cape Breton, Nova Scotia, Canada, juin 2001. IEEE Computer Society.

[30] Martín Abadi et Bruno Blanchet. – Secrecy Types for Asymmetric Communication. *In : Foundations of Software Science and Computation Structures (FoSSaCS'01)*, éd. par F. Honsell et M. Miculan, *Lecture Notes on Computer Science*, volume 2030, pp. 25–41, Gênes, Italie, avril 2001. Springer.

[31] Bruno Blanchet. – Escape Analysis for Object Oriented Languages. Application to Java^{TM}. *In : Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pp. 20–34, Denver, Colorado, novembre 1999.

[32] Bruno Blanchet. – Escape Analysis : Correctness Proof, Implementation and Experimental Results. *In : 25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*, pp. 25–37, San Diego, Californie, janvier 1998. ACM Press.

Workshops

[33] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov et Joe-Kai Tsay. – Computationally Sound Mechanized Proofs of Basic and Public-Key Kerberos. – octobre 2007. Shloss Dagstuhl seminar "Formal Protocol Verification Applied", Wadern, Germany.

[34] Bruno Blanchet. – A Computationally Sound Automatic Prover for Cryptographic Protocols *In : Workshop on the link between formal and computational models*, École normale supérieure, Paris, France, juin 2005.

[35] Bruno Blanchet. – Automatic Proof of Strong Secrecy for Security Protocols *In : Seminar "Language-Based Security"*, Schloss Dagstuhl, Wadern, Allemagne, octobre 2003.

[36] Bruno Blanchet et Benjamin Aziz. – A Calculus for Locations, Mobility, and Cryptography *In : Seminar "Reasoning about Shape"*, Schloss Dagstuhl, Wadern, Allemagne, mars 2003.

[37] Martín Abadi et Bruno Blanchet. – Secrecy Types for Asymmetric Communication. *In : Seminar "Security through Analysis and Verification"*, Schloss Dagstuhl, Wadern, Allemagne, décembre 2000.

Chapitre dans un ouvrage

[38] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival. – Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, chapitre invité. *In : The Essence of Computation : Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, éd. par T. Mogensen, D. A. Schmidt et I. H. Sudborough, pp. 85–108. – Springer, décembre 2002.

Logiciels

[39] Bruno Blanchet. – CryptoVerif, version 1.06. Vérificateur de protocoles cryptographiques dans le modèle calculatoire. Disponible à `http://www.cryptoverif.ens.fr/`. 2007.

[40] Bruno Blanchet et Xavier Allamigeon. – ProVerif, version 1.14. Vérificateur de protocoles cryptographiques dans le modèle formel. Disponible à `http://www.proverif.ens.fr/`. 2007.

[41] Bruno Blanchet. – Analyseur d'échappement pour l'allocation en pile dans Objective Caml, intégré dans le compilateur Ocaml 1.05, construit à partir d'un prototype d'Alain Deutsch. Disponible à `http://www.di.ens.fr/~blanchet/escape.html`. 2001.

Rapports

[42] Bruno Blanchet. – *Automatic Verification of Correspondences for Security Protocols.* – Rapport arXiv:0802.3444v1, février 2008. Disponible à `http://arxiv.org/abs/0802.3444v1`.

[43] Bruno Blanchet. – *Computationally sound mechanized proofs of correspondence assertions.* – Cryptology ePrint Archive, Rapport 2007/128, avril 2007. Disponible à `http://eprint.iacr.org/2007/128`.

[44] Bruno Blanchet et David Pointcheval. – *Automated security proofs with sequences of games.* – Cryptology ePrint Archive, Rapport 2006/069, février 2006. Disponible à `http://eprint.iacr.org/2006/069`.

[45] Bruno Blanchet. – *A computationally sound mechanized prover for security protocols.*– Cryptology ePrint Archive, Rapport 2005/401, novembre 2005. Disponible à `http://eprint.iacr.org/2005/401`.

[46] Bruno Blanchet. – *Automatic Proof of Strong Secrecy for Security Protocols.* – Rapport technique MPI-I-2004-NWG1-001, Max-Planck-Institut für Informatik, Sarrebruck, Allemagne, juillet 2004.

Mémoires

[47] Bruno Blanchet. – *Analyse d'échappement. Applications à ML et Java^{TM}.* – Thèse de doctorat, École polytechnique, 7 décembre 2000.

[48] Bruno Blanchet. – *Rapport de magistère MMFAI.* – ENS, octobre 1997.

[49] Bruno Blanchet. – *Garbage Collection statique.* – Rapport de DEA, INRIA, Rocquencourt, septembre 1996.

Annexe B

Articles joints

Vérification des protocoles dans le modèle formel

Martín Abadi et Bruno Blanchet. – Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, vol. 52, n° 1, janvier 2005, pp. 102–146.

Cet article présente un système de types pour vérifier des propriétés de secret de protocoles cryptographiques, codés dans un extension du pi calcul avec des symboles de fonction. Ce système de types fournit un traitement générique de nombreuses primitives cryptographiques, dont chiffrement à clé publique et à clé partagée, signatures, fonctions de hachage. Nous étudions plusieurs instances de ce système. Nous montrons en particulier qu'une des instances de ce système de types est équivalente à la méthode de vérification du secret fondée sur les clauses de Horn, utilisée par le vérificateur automatique ProVerif. Nous montrons également que cette instance du système de types est la plus précise : si une propriété de secret peut être prouvée par une instance quelconque du système de types, alors elle peut être prouvée par cette instance.

Bruno Blanchet. – Automatic Verification of Correspondences for Security Protocols. Rapport arXiv:0802.3444v1. Version sans preuves à paraître dans le *Journal of Computer Security.*

Cet article étend la méthode de vérification du secret fondée sur les clauses de Horn et présentée dans l'article précédent aux propriétés de correspondances. Les propriétés de correspondance sont des propriétés de la forme "si un certain événement a été exécuté, alors d'autres événements ont été exécutés". Ces propriétés sont utilisées en particulier pour formaliser l'authentification.

Cet article décrit également l'algorithme de résolution utilisé sur les clauses de Horn, sa preuve de correction, et montre sa terminaison sur une sous-classe de protocoles bien conçus, dans lesquels chaque chiffrement, signature, ... est distingué des autres par une étiquette constante.

Bruno Blanchet, Martín Abadi et Cédric Fournet. – Automated Verification of Selected Equivalences for Security Protocols. *Journal of Logic and Algebraic Programming*, vol. 75, n° 1, février-mars 2008, pp. 3–51.

Cet article étend également la méthode de vérification fondée sur les clauses de Horn, cette fois à la vérification d'équivalences de processus. Intuitivement, deux processus sont observationnellement équivalents quand l'attaquant ne peut pas les distinguer. Ces équivalences peuvent être utilisées pour spécifier de nombreuses propriétés de sécurité subtiles. Ici, nous nous concentrons sur les équivalences entre processus P et Q qui ne diffèrent que par le choix de certains termes. De telles équivalences

apparaissent souvent dans les applications, par exemple pour traiter les protocoles à mots de passe faibles. Nous montrons comment les traiter comme des prédicats sur les traces d'un processus qui représente à la fois P et Q.

Cet article présente également le traitement des primitives cryptographiques modélisées par une théorie équationnelle. Cela permet par exemple de représenter des primitives de chiffrement pour lesquelles le déchiffrement réussit toujours, ce qui est une propriété utile pour obtenir certaines équivalences.

Vérification des protocoles dans le modèle calculatoire

Bruno Blanchet. – A Computationally Sound Mechanized Prover for Security Protocols. *IEEE Transactions on Dependable and Secure Computing*, vol. 5, n° 4, octobre-décembre 2008, pp. 193–207.

Cet article présente le vérificateur automatique de protocoles CryptoVerif. Contrairement à la plupart des vérificateurs précédents, il n'est pas fondé sur le modèle formel, mais sur le modèle calculatoire. Il produit des preuves présentées comme des suites de jeux, comme celles utilisées par les cryptographes. Ces jeux sont formalisés dans un calcul de processus probabiliste polynomial. CryptoVerif fournit une méthode générique pour spécifier les hypothèses de sécurité sur les primitives cryptographiques, qui peut traiter en particulier chiffrement à clé partagée et à clé publique, signatures, codes d'authentification de messages, fonctions de hachage. Il produit des preuves valides pour un nombre de sessions polynomial dans le paramètre de sécurité, en présence d'un attaquant actif.

Analyzing Security Protocols with Secrecy Types and Logic Programs*

Martín Abadi
Computer Science Department
University of California, Santa Cruz
abadi@cs.ucsc.edu

Bruno Blanchet
CNRS, Département d'Informatique
École Normale Supérieure, Paris
Bruno.Blanchet@ens.fr

Abstract

We study and further develop two language-based techniques for analyzing security protocols. One is based on a typed process calculus; the other, on untyped logic programs. Both focus on secrecy properties. We contribute to these two techniques, in particular by extending the former with a flexible, generic treatment of many cryptographic operations. We also establish an equivalence between the two techniques.

1 Introduction

Concepts and methods from programming languages have long been useful in security (e.g., [47]). In recent years, they have played a significant role in understanding security protocols. They have given rise to programming calculi for these protocols (e.g., [7, 9, 11, 21, 24, 28, 29, 31, 42, 44, 52]). They have also suggested several approaches for reasoning about protocols, leading to theories as well as tools for formal protocol analysis. We describe some of these approaches below. Although several of them are incomplete (in the sense that they sometimes fail to establish security properties), they are applicable to many protocols, including infinite-state protocols, often with little effort. Thus, they provide an attractive alternative to finite-state model checking (e.g., [43]) and to human-guided theorem proving (e.g., [50]).

In this work we pursue these language-based approaches to protocol analysis and aim to clarify their interconnections. We examine and further develop two techniques that represent two popular, substantial, but largely disjoint lines of research. One technique relies on a typed process calculus, the other on untyped logic programs. We contribute to these two techniques, in particular by extending the former with a flexible, generic treatment of many cryptographic operations. We also establish an equivalence between the two techniques. We believe that this equivalence is surprising and illuminating.

The typed process calculus belongs in a line of research that exploits standard static-analysis ideas and adapts them with security twists. There are by now several type systems for processes in which types not only track the expected structure of values and processes but also give security information [1, 5, 20, 32, 33, 38, 39]. A related approach relies on control-flow analysis [18]; it has an algorithmic emphasis, but it is roughly equivalent to typing at least in important special cases [17]. Such static analyses have applications in a broader security context (e.g., [3, 37, 48, 53]); security protocols constitute a particularly challenging class of examples. To date, however, such static analyses have dealt case by case with operations on data, and in particular with cryptographic operations. In this paper, we develop a general treatment of these operations.

*This work was presented at the 29th Annual ACM Symposium on Principles of Programming Languages (2002). A preliminary version of this paper appears in the proceedings of that symposium.

In another line of research, security protocols are represented as logic programs, and they are analyzed symbolically with general provers [26, 54] or with special-purpose algorithms and tools [4, 13–16, 21–23, 25, 34–36, 51]. (See also [40] for some of the roots of this approach.) In some of this work [21, 23], the use of linear logic enables a rather faithful model of protocol state, reducing (or eliminating) the possibility of false alarms; on the other hand, the treatment of protocols with an unbounded number of sessions can become quite difficult. Partly for this reason, and partly because of the familiarity and relative simplicity of classical logic, algorithms and tools that rely on classical logic programs are prevalent. Superficially, these algorithms and tools are quite different from typing and control-flow analysis. However, in this paper we show that one of these tools can be viewed as an implementation of a type system.

More specifically, we develop a generic type system for a process calculus that extends the pi calculus [46] with constructor operations and corresponding destructor operations. These operations may be, for instance, tupling and projection, symmetric (shared-key) encryption and decryption, asymmetric (public-key) encryption and decryption, digital signatures and signature checking, and one-way hashing (with no corresponding destructor). As in the applied pi calculus [7], these operations are not hardwired. The applied pi calculus is even more general in that it does not require the classification of operations into constructors and destructors; we expect that it can be treated along similar lines but with more difficulty (see Sections 2 and 8). Our type system for the process calculus gives secrecy information. The basic soundness theorem for the type system, which we prove only once (rather than once per choice of operations), states that well-typed processes do not reveal their secrets.

We compare this generic type system with an automatic protocol checker. The checker takes as input a process and translates it into an abstract representation by logic-programming rules. This representation and its manipulation, but not the translation of processes, come from previous work [13], which develops an efficient tool for establishing secrecy properties of protocols. We show that establishing a secrecy property of a protocol with this checker corresponds to typing the protocol in a particular instance of the generic type system. This result implies a soundness property for the checker. Conversely, as a completeness property, we establish that the checker corresponds to the "best" instance of our generic type system: if a secrecy property can be established using any instance of the type system, then it can also be established by the checker.

Throughout this paper, we use the following concept of secrecy (e.g., [2]): a protocol P preserves the secrecy of data M if P never publishes M, or anything that would permit the computation of M, even in interaction with an adversary Q. For instance, M may be a cryptographic key; its secrecy means that no adversary can obtain the key by attacking P. Although this property allows the possibility that P reveals partial information about M, the property is attractive and often satisfactory.

For example, consider the following protocol (presented informally here, and studied more rigorously in the body of this paper):

$$\begin{array}{ll} \text{Message 1.} & A \to B : \mathit{pencrypt}((k, pK_A), pK_B) \\ \text{Message 2.} & B \to A : \mathit{pencrypt}((k, K_{AB}), pK_A) \\ \text{Message 3.} & A \to B : \mathit{sencrypt}(s, K_{AB}) \end{array}$$

This protocol establishes a session key K_{AB} between two parties A and B, then uses the key to transmit a secret s from A to B. It relies on a public-key encryption function *pencrypt*, on a shared-key encryption function *sencrypt*, and on public keys pK_A for A and pK_B for B. For *pencrypt* and *sencrypt*, the second argument is the encryption key, the first the plaintext being encrypted. First, A creates a challenge k (a nonce), sends it to B paired with A's public key, encrypted under B's public key. Then B replies with the same nonce and the session key K_{AB}, encrypted under A's public key. When A receives this message, it recognizes k; it is then confident that the key K_{AB} has been created by B. Finally, A sends the secret s under K_{AB}.

$M, N ::=$	terms
$\quad x, y, z$	variable
$\quad a, b, c, k, s$	name
$\quad f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\quad \overline{M}\langle N\rangle.P$	output
$\quad M(x).P$	input
$\quad 0$	nil
$\quad P \mid Q$	parallel composition
$\quad !P$	replication
$\quad (\nu a)P$	restriction
$\quad$ *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q	destructor application
$\quad$ *let* $x = M$ *in* P	local definition
$\quad$ *if* $M = N$ *then* P *else* Q	conditional

Figure 1: Syntax of the process calculus

Can an attacker obtain s? The answer to this question may partly depend on delicate points that the informal description of the protocol does not clarify, such as whether a public key can be mistaken for a shared key. Once we address those points through a formal description of the protocol, we can apply our analyses for establishing the secrecy of s or for identifying vulnerabilities.

The next section presents our process calculus, without types. Section 3 gives a (fairly standard) definition of secrecy. Section 4 presents our type system, and Section 5 gives the main soundness theorems for the type system and related results. As an application, Section 6 explains how the type system can be instantiated to handle shared-key and public-key encryption operations. Section 7 formalizes and studies the logic-programming protocol checker. Section 8 discusses an extension (to general equational theories). Section 9 concludes. An appendix contains some proofs.

2 The Process Calculus (Untyped)

This section introduces our process calculus, by giving its syntax and its operational semantics.

2.1 Syntax and Informal Semantics

The syntax of our calculus is summarized in Figure 1. It distinguishes a category of terms (data) and one of processes (programs). It assumes an infinite set of names and an infinite set of variables; a, b, c, k, s, and similar identifiers range over names, and x, y, and z range over variables. Names represent atomic data items, such as nonces and keys, while variables are formal parameters that can be replaced by any term (atomic or complex). The syntax also assumes a set of symbols for constructors and destructors, each with an arity; we often use f for a constructor and g for a destructor.

Constructors are used to build terms. Therefore, the terms are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$. On the other hand, destructors do not appear in terms, but only manipulate terms in processes. They are partial functions on terms that processes can apply. The process *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q tries to evaluate $g(M_1, \ldots, M_n)$; if this succeeds, then x is bound to the result and P is executed, else Q is executed. More precisely, the semantics of a destructor g of arity n is given by a partial function from n-tuples of terms to terms, such that $g(\sigma M_1, \ldots, \sigma M_n) = \sigma g(M_1, \ldots, M_n)$ if

$g(M_1, \ldots, M_n)$ is defined and σ is a substitution that maps names and variables to terms. We may isolate a minimal set $\text{def}(g)$ of equations $g(M'_1, \ldots, M'_n) = M'$ that define g, where $M'_1, \ldots, M'_n, M'$ are terms without free names, and all variables of M' occur in $M'_1, \ldots, M'_n$. Then $g(M_1, \ldots, M_n)$ is defined if and only if there exists a substitution σ and an equation $g(M'_1, \ldots, M'_n) = M'$ in $\text{def}(g)$ such that $M_i = \sigma M'_i$ for all $i \in \{1, \ldots, n\}$, and $g(M_1, \ldots, M_n) = \sigma M'$. This set of equations may be infinite, but it is usually finite and small in concrete examples.

Using these constructors and destructors, we can represent data structures, such as tuples, and cryptographic operations, for instance as follows:

- $\mathit{ntuple}(M_1, \ldots, M_n)$ is the tuple of the terms $M_1, \ldots, M_n$, where ntuple is a constructor. (We sometimes abbreviate $\mathit{ntuple}(M_1, \ldots, M_n)$ to $(M_1, \ldots, M_n)$.) The n projections are destructors ith_n for $i \in \{1, \ldots, n\}$, defined by

$$\mathit{ith}_n(\mathit{ntuple}(M_1, \ldots, M_n)) = M_i$$

- $\mathit{sencrypt}(M, N)$ is the symmetric (shared-key) encryption of the message M under the key N, where *sencrypt* is a constructor. The corresponding destructor *sdecrypt* is defined by

$$\mathit{sdecrypt}(\mathit{sencrypt}(M, N), N) = M$$

  Thus, $\mathit{sdecrypt}(M', N)$ returns the decryption of M' if M' is a message encrypted under N.

- In order to represent asymmetric (public-key) encryption, we may use two constructors *pk* and *pencrypt*: $\mathit{pk}(M)$ builds a public key from a secret M and $\mathit{pencrypt}(M, N)$ encrypts M under N. The corresponding destructor *pdecrypt* is defined by

$$\mathit{pdecrypt}(\mathit{pencrypt}(M, \mathit{pk}(N)), N) = M$$

- As for digital signatures, we may use a constructor *sign*, and write $\mathit{sign}(M, N)$ for M signed with the signature key N, and the two destructors *checksignature* and *getmessage* with the equations:

$$\mathit{checksignature}(\mathit{sign}(M, N), \mathit{pk}(N)) = M$$
$$\mathit{getmessage}(\mathit{sign}(M, N)) = M$$

- We may represent a one-way hash function by the constructor H. There is no corresponding destructor; so we model that the term M cannot be retrieved from its hash $H(M)$.

Thus, the process calculus supports many of the operations common in security protocols. It has limitations, though: for example, XOR cannot be directly represented by a constructor or by a destructor. We explain how we can treat such primitives in Section 8.

The other constructs in the syntax of Figure 1 are standard; most of them come from the pi calculus.

- The input process $M(x).P$ inputs a message on channel M, and executes P with x bound to the input message. The output process $\overline{M}\langle N\rangle.P$ outputs the message N on the channel M and then executes P. Here, we use an arbitrary term M to represent a channel: M can be a name, a variable, or a constructor application, but the process blocks if M does not reduce to a name at runtime. Our calculus is monadic (in that the messages are terms rather than tuples of terms), but a polyadic calculus can be simulated since tuples are terms. It is also synchronous (in that a process P is executed after the output of a message). As usual, we may omit P when it is 0.

- The nil process 0 does nothing.
- The process $P \mid Q$ is the parallel composition of P and Q.
- The replication $!P$ represents an unbounded number of copies of P in parallel.
- The restriction $(\nu a)P$ creates a new name a, and then executes P.
- The local definition $\mathit{let}\ x = M\ \mathit{in}\ P$ executes P with x bound to the term M.
- The conditional $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ executes P if M and N reduce to the same term at runtime; otherwise, it executes Q. As usual, we may omit an *else* clause when it consists of 0.

The name a is bound in the process $(\nu a)P$. The variable x is bound in P in the processes $M(x).P$, $\mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$, and $\mathit{let}\ x = M\ \mathit{in}\ P$. We write $\mathit{fn}(P)$ and $\mathit{fv}(P)$ for the sets of names and variables free in P, respectively. A process is closed if it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables. We write $\{M_1/x_1, \dots, M_n/x_n\}$ for the substitution that replaces $x_1, \dots, x_n$ with $M_1, \dots, M_n$, respectively. When σ is such a substitution and D is some expression, we may write σD or $D\sigma$ for the result of applying σ to D; the distinction is one of emphasis at most. Except when stated otherwise, substitutions always map variables (not names) to expressions.

As mentioned in the introduction, our calculus resembles the applied pi calculus [7]. Both calculi are extensions of the pi calculus with (fairly arbitrary) functions on terms. However, there are also important differences between these calculi. The first one is that we use destructors instead of the equational theories of the applied pi calculus. (Section 8 contains further material on equational theories.) The second difference is that our calculus has a built-in error-handling construct (the *else* clause of the destructor application), whereas in the applied pi calculus the error-handling must be done “by hand”. This error-handling construct makes typing easier.

2.2 An Example

As an example, we return to the exchange presented in the introduction, namely:

$$\begin{array}{ll} \text{Message 1.} & A \rightarrow B : \mathit{pencrypt}((k, pK_A), pK_B) \\ \text{Message 2.} & B \rightarrow A : \mathit{pencrypt}((k, K_{AB}), pK_A) \\ \text{Message 3.} & A \rightarrow B : \mathit{sencrypt}(s, K_{AB}) \end{array}$$

Next we show how to express this protocol in the process calculus. We return again to this example in later sections, and there we discuss its formal analysis.

Informal protocol descriptions, such as the one for this protocol, are often ambiguous [2], so several different process-calculus expressions may be reasonable counterparts to an informal description. We start with a relatively simple representation of the protocol, given in the following process P:

$$\begin{array}{rcl} P & \triangleq & (\nu sK_A)(\nu sK_B)\mathit{let}\ pK_A = pk(sK_A)\ \mathit{in} \\ & & \quad \mathit{let}\ pK_B = pk(sK_B)\ \mathit{in}\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(A \mid B) \\ A & \triangleq & (\nu k)\overline{e}\langle \mathit{pencrypt}((k, pK_A), pK_B)\rangle. \\ & & \quad e(z).\mathit{let}\ (x, y) = \mathit{pdecrypt}(z, sK_A)\ \mathit{in} \\ & & \quad \mathit{if}\ x = k\ \mathit{then}\ \overline{e}\langle \mathit{sencrypt}(s, y)\rangle \\ B & \triangleq & e(z).\mathit{let}\ (x, y) = \mathit{pdecrypt}(z, sK_B)\ \mathit{in} \\ & & \quad (\nu K_{AB})\overline{e}\langle \mathit{pencrypt}((x, K_{AB}), y)\rangle. \\ & & \quad e(z').\mathit{let}\ s' = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0 \end{array}$$

Here we write *let* $(x, y) = M$ *in* Q instead of *let* $z = M$ *in let* $x = 1th_2(z)$ *in let* $y = 2th_2(z)$ *in* Q, using pattern-matching on tuples. The keys sK_A and sK_B are the decryption keys that match pK_A and pK_B, respectively, and e is a public channel. The messages $\overline{e}\langle pK_A\rangle$ and $\overline{e}\langle pK_B\rangle$, which publish pK_A and pK_B on e, model the fact that these keys are public. This code corresponds to a basic, one-shot version of the protocol, in which A talks only to B and in which honest hosts that play the roles of A and B use different keys.

It is easy to extend the code to represent more elaborate, general versions of the protocol. For instance, the following process P' represents a version in which A and B run an unbounded number of sessions, A can talk to any host (whose public key A receives in x_{pK_B}), and the hosts that play the roles of A and B may have the same key:

$$
\begin{aligned}
P' &\triangleq (\nu sK_A)(\nu sK_B) let\ pK_A = pk(sK_A)\ in \\
&\quad let\ pK_B = pk(sK_B)\ in\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(!A' \mid !B' \mid !B'') \\
A' &\triangleq e(x_{pK_B}).(\nu k)\overline{e}\langle pencrypt((k, pK_A), x_{pK_B})\rangle. \\
&\quad e(z).let\ (x, y) = pdecrypt(z, sK_A)\ in\ if\ x = k\ then \\
&\quad\quad (if\ x_{pK_B} = pK_A\ then\ \overline{e}\langle sencrypt(s_A, y)\rangle \\
&\quad\quad \mid if\ x_{pK_B} = pK_B\ then\ \overline{e}\langle sencrypt(s_B, y)\rangle) \\
B' &\triangleq e(z).let\ (x, y) = pdecrypt(z, sK_B)\ in \\
&\quad (\nu K_{AB})\overline{e}\langle pencrypt((x, K_{AB}), y)\rangle. \\
&\quad e(z').let\ s' = sdecrypt(z', K_{AB})\ in\ 0 \\
B'' &\triangleq e(z).let\ (x, y) = pdecrypt(z, sK_A)\ in \\
&\quad (\nu K_{AB})\overline{e}\langle pencrypt((x, K_{AB}), y)\rangle. \\
&\quad e(z').let\ s' = sdecrypt(z', K_{AB})\ in\ 0
\end{aligned}
$$

Here B'' is much like B' but uses the same key as A'. (A separate definition of B'' is needed because, in the applied pi calculus, the syntactically different names sK_A and sK_B never mean the same. Of course, the code duplication can easily be avoided by using a variable parameter for the keys.)

This and other variants can be written rather directly as scripts in the input syntax of the automatic protocol checker, which is quite close to that of the process calculus. The following script illustrates this point:

```
(* First some declarations, with equations *)

(* Shared-key encryption *)

fun sencrypt/2.
reduc sdecrypt(sencrypt(x, y), y) = x.

(* Public-key encryption *)

fun pencrypt/2.
fun pk/1.
reduc pdecrypt(pencrypt(x, pk(y)), y) = x.

(* Declarations of free names *)

private free sA, sB.
free e.
```

```
(* A secrecy query, for protocol analysis *)

query attacker : sA;
      attacker : sB.

(* The processes *)

let processA' =
        in(e, xpkB);
        new k;
        out(e, pencrypt((k, pkA), xpkB));
        in(e, z);
        let (x, y) = pdecrypt(z, skA) in
        if x = k then
        (
          if xpkB = pkA then
          out(e, sencrypt(sA, y))
        )
        |
        (
          if xpkB = pkB then
          out(e, sencrypt(sB, y))
        ).

let processB' =
        in(e, z);
        let (x, y) = pdecrypt(z, skB) in
        new Kab;
        out(e, pencrypt((x, Kab), y));
        in(e, z2);
        let s2 = sdecrypt(z2, Kab) in
        0.

let processB'' =
        in(e, z);
        let (x, y) = pdecrypt(z, skA) in
        new Kab;
        out(e, pencrypt((x, Kab), y));
        in(e, z2);
        let s2 = sdecrypt(z2, Kab) in
        0.

process new skA;
        new skB;
        let pkA = pk(skA) in
        let pkB = pk(skB) in
        out(e, pkA);
        out(e, pkB);
        ((!processA') | (!processB') | (!processB''))
```

As can be seen from this example, writing a model of a protocol in the process calculus is much like programming it in a little language with concurrency, message passing on named

$$\frac{}{P \mid 0 \equiv P} \qquad \frac{}{P \mid Q \equiv Q \mid P} \qquad \frac{}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)}$$

$$\frac{}{!P \equiv P \mid !P}$$

$$\frac{}{(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P} \qquad \frac{a \notin fn(P)}{(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q}$$

$$\frac{P \equiv Q}{P \mid R \equiv Q \mid R} \qquad \frac{P \equiv Q}{!P \equiv !Q} \qquad \frac{P \equiv Q}{(\nu a)P \equiv (\nu a)Q}$$

$$\frac{}{P \equiv P} \qquad \frac{Q \equiv P}{P \equiv Q} \qquad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R}$$

$$\frac{}{\overline{a}\langle M\rangle.Q \mid a(x).P \ \rightarrow\ Q \mid P\{M/x\}} \quad \text{(Red I/O)}$$

$$\frac{g(M_1, \ldots, M_n) = M'}{let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q \rightarrow P\{M'/x\}} \quad \text{(Red Destr 1)}$$

$$\frac{g(M_1, \ldots, M_n) \text{ is not defined}}{let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q \rightarrow Q} \quad \text{(Red Destr 2)}$$

$$\frac{}{let\ x = M\ in\ P \rightarrow P\{M/x\}} \quad \text{(Red Let)}$$

$$\frac{}{if\ M = M\ then\ P\ else\ Q \ \rightarrow\ P} \quad \text{(Red Cond 1)}$$

$$\frac{M \neq N}{if\ M = N\ then\ P\ else\ Q \ \rightarrow\ Q} \quad \text{(Red Cond 2)}$$

$$\frac{P \ \rightarrow\ Q}{P \mid R \ \rightarrow\ Q \mid R} \quad \text{(Red Par)}$$

$$\frac{P \ \rightarrow\ Q}{(\nu a)P \ \rightarrow\ (\nu a)Q} \quad \text{(Red Res)}$$

$$\frac{P' \equiv P,\ P \ \rightarrow\ Q,\ Q \equiv Q'}{P' \ \rightarrow\ Q'} \quad \text{(Red} \equiv\text{)}$$

Figure 2: Structural congruence and reduction

channels, and high-level, "black-box" operations on data (including cryptographic functions). In this respect, the calculus resembles many of the other programming calculi for protocols mentioned in the introduction.

The literature contains additional examples that provide evidence of the effectiveness of this process calculus and related ones for the analysis of a range of protocols. In particular, we have recently used this process calculus in the study of a protocol for certified email [4, 8] and of the JFK protocol (a proposed replacement for IKE in IPsec) [6, 10].

2.3 Formal Semantics

The rules of Figure 2 axiomatize the reduction relation $\rightarrow$ for processes, thus defining the operational semantics of our calculus. As is often done in process calculi (e.g., [46]), auxiliary rules axiomatize the structural congruence relation $\equiv$. This relation is useful for transforming processes so that the reduction rules can be applied. Both $\equiv$ and $\rightarrow$ are defined only on closed processes.

We write $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. As in [5], we say that the process P outputs M immediately on c if and only if $P \equiv \overline{c}\langle M\rangle.Q \mid R$ for some processes Q and R. We say that the process P outputs M on c if and only if $P \rightarrow^* Q$ and Q outputs M immediately on c for some process Q.

3 A Definition of Secrecy

As indicated in the introduction, we use the following informal definition of secrecy: a protocol P preserves the secrecy of data M if P never publishes M, or anything that would permit the computation of M, even in interaction with an adversary Q. Equivalently, a protocol P preserves the secrecy of data M if P in parallel with an adversary Q will never output M on a public channel. The interaction between P and Q takes place by communication on shared channels. These primarily include public channels (such as those of the Internet), on which Q may eavesdrop, modify, and inject messages; they may also include other channels known to Q. In addition to these shared channels, P may use private channels for its internal computations.

Next we give a formal counterpart for this informal definition, in the context of our process calculus and relying on the operational semantics of Section 2.3.

We represent the adversary Q as a process of the calculus, with some hypotheses that characterize Q's initial capabilities. We formulate these hypotheses simply by using a set of names S. Intuitively, Q knows the names in S initially; in particular, these names may represent the cryptographic keys, communication channels, and nonces that Q knows initially. In the course of computation, Q may acquire some additional capabilities (for instance, additional cryptographic keys) not represented in S, by creating fresh names and receiving terms in messages.

In order to represent that Q may initially know complex terms rather than just names, we may let P begin with the output of these terms on a public channel $c \in S$, so the restriction that S is a set of names entails no loss of generality.

Definition 3.1 Let S be a finite set of names. The closed process Q is a S-adversary if and only if $\mathit{fn}(Q) \subseteq S$. The closed process P preserves the secrecy of M from S if and only if $P \mid Q$ does not output M on c for any S-adversary Q and any $c \in S$.

If P preserves the secrecy of M from S, then it clearly cannot output M on some $c \in S$, that is, on one of the channels known to the adversary. This guarantee corresponds to the informal requirement that P never publishes M on its own. Moreover, P cannot publish data that would enable an adversary to compute M, because the adversary could go on to output M on some $c \in S$.

For instance, the process $(\nu k)\overline{a}\langle \mathit{sencrypt}(s,k)\rangle$ preserves the secrecy of s from $\{a\}$. This process publishes an encryption of s on the channel a, but not the decryption key; hence s does not escape. Similarly, the process $(\nu a)\overline{a}\langle \mathit{sencrypt}(s,k)\rangle$ preserves the secrecy of s from $\{k\}$; here the key is published but the channel remains private. On the other hand, the process $\overline{a}\langle \mathit{sencrypt}(s,k)\rangle$ does not preserve the secrecy of s from $\{a,k\}$: the adversary

$$a(x).\overline{a}\langle \mathit{sdecrypt}(x,k)\rangle$$

can receive $\mathit{sencrypt}(s,k)$ on a, decrypt s, and resend it on a.

As an additional example, we may apply this definition of secrecy to the process P of the example of Section 2.2. We may ask whether P preserves the secrecy of s from $\{e\}$. This property would mean that an attacker with access to e cannot learn s. Section 6.1 shows that this property indeed holds.

Definitions along these lines are quite common in protocol analysis. They are particularly popular and useful for treating the secrecy of keys and other atomic data items. There are however alternatives, in particular some definitions based on the concept of noninterference.

According to those, a protocol parameter (such as the identity of a participant) is secret if an adversary cannot tell an instance of the protocol with one value of the parameter from an instance with a different value. The adversary may actually have these values, but ignore which is in use. In contrast, Definition 3.1 implies that, when a process P keeps the secrecy of a term M, the adversary does not have M. See [2] for further details and discussion.

4 The Type System

This section presents a general type system for our process calculus: Section 4.1 describes parameters and assumptions of the type system, and Section 4.2 describes its judgments and the type rules, which Figure 3 gives. The following sections include instances of this general type system.

4.1 Parameters and Requirements

The type system is parameterized by a set of types *Types* and a non-empty subset $T_{\mathrm{Public}} \subseteq \mathit{Types}$. These parameters will be fixed in each instance of the type system. Always, T_{Public} is intended as the set of types of data that can be known by the attacker. The set T_{Public} is crucial in formulating our secrecy results, in which we assume that the attacker has names with types in T_{Public} and prove that it does not have names with types not in T_{Public}.

The type system relies on a function $\mathit{conveys} : \mathit{Types} \to \mathcal{P}(\mathit{Types})$ that satisfies property (P0):

(P0) If $T \in T_{\mathrm{Public}}$, then $\mathit{conveys}(T) = T_{\mathrm{Public}}$.

Intuitively, $\mathit{conveys}(T)$ is the set of types of data that are conveyed by a channel of type T. (It is empty when elements of T cannot be used as channels.) Data conveyed by a public channel is public, since the adversary can obtain it by listening on the channel. Conversely, public data can appear on a public channel, since the adversary can send it.

The type system also relies on a partial function from types to types $O_f : \mathit{Types}^n \to \mathit{Types}$ for each constructor f of arity n, and a function from types to sets of types $O_g : \mathit{Types}^n \to \mathcal{P}(\mathit{Types})$ for each destructor g of arity n. Basically, these operators O_f and O_g give the types of constructor and destructor applications (much like type declarations for f and g), so they determine the type rules for constructors and destructors. As the type rules say, if $M_1, \ldots, M_n$ have respective types $T_1, \ldots, T_n$, f is a constructor of arity n, and $O_f(T_1, \ldots, T_n)$ is defined, then $f(M_1, \ldots, M_n)$ has type $O_f(T_1, \ldots, T_n)$. Similarly, if $M_1, \ldots, M_n$ have respective types $T_1, \ldots, T_n$, g is a destructor of arity n, and $g(M_1, \ldots, M_n)$ is defined, then $g(M_1, \ldots, M_n)$ has a type in $O_g(T_1, \ldots, T_n)$.

These constructor and destructor applications need not have unique or most general types (but terms do have unique types in a given environment). Constructors and destructors can accept arguments of different types, and return results whose types depend on the types of the arguments. In this sense, we may say that they are overloaded functions; this overloading subsumes some forms of subtyping and parametric polymorphism.

We require the following properties:

(P1) If for all $i \in \{1, \ldots, n\}$, $T_i \in T_{\mathrm{Public}}$, then $O_f(T_1, \ldots, T_n)$ is defined and $O_f(T_1, \ldots, T_n) \in T_{\mathrm{Public}}$.

(P2) If for all $i \in \{1, \ldots, n\}$, $T_i \in T_{\mathrm{Public}}$ and $T \in O_g(T_1, \ldots, T_n)$, then $T \in T_{\mathrm{Public}}$.

(P3) For each equation $g(M_1, \ldots, M_n) = M$ in $\mathrm{def}(g)$, if for all $i \in \{1, \ldots, n\}$, $E \vdash M_i : T_i$, then there exists $T \in O_g(T_1, \ldots, T_n)$ such that $E \vdash M : T$.

These properties are both reasonable and necessary for the soundness of the type system. The first two properties reflect that the result of applying a function to public terms should also be public, since the adversary can compute it. These properties are important in the proof of the Typability Lemma (Lemma 5.1.4 in Section 5). The third property essentially says that the definition of O_g on types is compatible with the definition of g on terms. This property is useful for type preservation when a destructor is applied, in the proof of the Subject Reduction Lemma (Lemma 5.1.3 in Section 5).

Thus, in summary, the type system is parameterized by:

- the set of types *Types*,
- the subset $T_{\text{Public}} \subseteq \mathit{Types}$,
- the function $\mathit{conveys} : \mathit{Types} \to \mathcal{P}(\mathit{Types})$,
- a partial function $O_f : \mathit{Types}^n \to \mathit{Types}$ for each constructor f of arity n, and
- a function $O_g : \mathit{Types}^n \to \mathcal{P}(\mathit{Types})$ for each destructor g of arity n,

and these parameters are subject to conditions (P0, P1, P2, P3).

As it stands, the type system is not parameterized by a subtyping relation. On the other hand, we sometimes find it convenient to use specific subtyping relations in instances of the type system, without however a "subsumption" rule [19]. This rule is present in many programming languages with subtyping (but not all: see for example Objective Caml). It would enables us to view every element of a type T as having a type T' whenever T is a subtype of T', and therefore to apply any function f that expects an argument of type T' to any element of type T. It would be fairly easy to add this rule, should one wish to do so; we have developed some of the corresponding theory. We have not needed this rule because, as explained above, our constructors and destructors can accept arguments of different types, and return results whose types depend on the types of the arguments. Therefore, a function f that expects an argument of type T' can be defined to handle arguments of any other type T as well.

The soundness of the type system depends on the proper definition of constructors and destructors. In particular, the result of a constructor must be a new term, not equal to any other term. For instance, the identity function cannot be a constructor (but it may be a destructor). Otherwise, taking the identity function as a constructor, we could type it with $O_{id}(T) = T' \in T_{\text{Public}}$ for all T, so it could convert a secret type into a public one, and this would lead to wrong secrecy proofs. Once the constructors and destructors are defined correctly and the required properties (P0, P1, P2, P3) hold, the type system provides secrecy guarantees, as we show in the next section.

4.2 Judgments and Rules

Figure 3 gives the rules of the type system. In the rules, the metavariable u ranges over names and variables (that is, over atomic terms), and T over types. The rules concern three judgments:

- $E \vdash \diamond$ means that E is a well-formed typing environment.
- $E \vdash M : T$ means that M is a term of type T in the environment E.
- $E \vdash P$ says that the process P is well-typed in the environment E.

The type rules for nil, parallel composition, replication, restriction, and local definition are standard. We use a Curry-style typing for restriction, so we do not mention a type of a explicitly in the construct (νa). (That is, we do not write $(\nu a : T)$ for some T.) This style of typing gives rise to a form of polymorphism: the type of a can change according to the environment. We

Well-formed environments:

$$\frac{}{\emptyset \vdash \diamond} \qquad \text{(Env } \emptyset\text{)}$$

$$\frac{E \vdash \diamond \qquad u \notin \mathit{dom}(E)}{E, u : T \vdash \diamond} \qquad \text{(Env atom)}$$

Terms:

$$\frac{E \vdash \diamond \qquad (u : T) \in E}{E \vdash u : T} \qquad \text{(Atom)}$$

$$\frac{E \vdash \diamond \qquad \forall i \in \{1, \dots, n\}, E \vdash M_i : T_i \qquad O_f(T_1, \dots, T_n) \text{ is defined}}{E \vdash f(M_1, \dots, M_n) : O_f(T_1, \dots, T_n)}$$

(Constructor application)

Processes:

$$\frac{E \vdash M : T \qquad E \vdash N : T' \qquad T' \in \mathit{conveys}(T) \qquad E \vdash P}{E \vdash \overline{M}\langle N \rangle.P} \qquad \text{(Output)}$$

$$\frac{E \vdash M : T \qquad \forall T' \in \mathit{conveys}(T), E, x : T' \vdash P}{E \vdash M(x).P} \qquad \text{(Input)}$$

$$\frac{E \vdash \diamond}{E \vdash 0} \qquad \text{(Nil)}$$

$$\frac{E \vdash P \qquad E \vdash Q}{E \vdash P \mid Q} \qquad \text{(Parallel composition)}$$

$$\frac{E \vdash P}{E \vdash !P} \qquad \text{(Replication)}$$

$$\frac{E, a : T \vdash P}{E \vdash (\nu a)P} \qquad \text{(Restriction)}$$

$$\frac{\forall i \in \{1, \dots, n\}, E \vdash M_i : T_i \qquad \forall T \in O_g(T_1, \dots, T_n), E, x : T \vdash P \qquad E \vdash Q}{E \vdash \mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q}$$

(Destructor application)

$$\frac{E \vdash M : T \qquad E, x : T \vdash P}{E \vdash \mathit{let}\ x = M\ \mathit{in}\ P} \qquad \text{(Local definition)}$$

$$\frac{E \vdash M : T \qquad E \vdash N : T' \qquad \text{if } T = T' \text{ then } E \vdash P \qquad E \vdash Q}{E \vdash \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q} \qquad \text{(Conditional)}$$

Figure 3: Type rules

could easily have a variant with explicit types on restrictions. The resulting type system would be less powerful, but our soundness results would still hold.

By the rule (Output), the process $\overline{M}\langle N\rangle.P$ is well-typed only if data of the type T' of N can be conveyed on a channel of the type T of M, that is, $T' \in \mathit{conveys}(T)$. Conversely, for typechecking the process $M(x).P$ via the rule (Input), the variable x is considered with all types $T' \in \mathit{conveys}(T)$ where T is the type of M. The universal quantification on the type of x is unusual; it arises because a channel may convey data of several types. In security protocols, this flexibility is important because a channel may convey data from the adversary and from honest participants, and types can help distinguish these two cases.

The rule (Constructor application) types $f(M_1, \ldots, M_n)$ according to the corresponding operator O_f. The rule (Destructor application) is similar to (Input); in $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$, the variable x is considered with all the possible types of $g(M_1, \ldots, M_n)$, that is, all elements of $O_g(T_1, \ldots, T_n)$.

Rule (Conditional) exploits the property that if two terms M and N have different types then they are certainly different. In this case, $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ may be well-typed without P being well-typed.

The constructs $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ and $\mathit{let}\ x = M\ \mathit{in}\ P$ can be defined as special cases from $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$, and their typing follows:

- Let *equals* be a binary destructor with $\mathit{equals}(M, M) = M$ (and $\mathit{equals}(M, N)$ undefined otherwise), $O_{\mathit{equals}}(T, T) = \{T\}$, and $O_{\mathit{equals}}(T, T') = \emptyset$ if $T \neq T'$. Then $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ can be defined and typed as $\mathit{let}\ x = \mathit{equals}(M, N)\ \mathit{in}\ P\ \mathit{else}\ Q$, where $x \notin \mathit{fv}(P)$.

- Let *id* be a unary destructor with $\mathit{id}(M) = M$ and $O_{\mathit{id}}(T) = \{T\}$. Then $\mathit{let}\ x = M\ \mathit{in}\ P$ can be defined and typed as $\mathit{let}\ x = \mathit{id}(M)\ \mathit{in}\ P\ \mathit{else}\ 0$.

Because of these encodings, we may omit the cases of $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ and $\mathit{let}\ x = M\ \mathit{in}\ P$ in various arguments and proofs. The encodings also suggest that the typing rule (Conditional) for $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ is more natural than might seem at first sight.

In the rules (Input) and (Destructor application), the type system uses universal quantifications over a possibly infinite set of types, and the rule (Restriction) involves picking a type from a possibly infinite set. These features are important for the richness of the type system. For example, had we attached a single type to the variable in (Destructor application), we could not have dealt with situations in which a process receives an encrypted message with a cleartext of a statically unknown type.

On the other hand, these features are challenging from an algorithmic perspective: typechecking and type inference are not easy to implement in general. Unless explicit types are given, typechecking requires guessing the types of restricted names. Even worse, typechecking requires considering bound variables with a potentially infinite set of types, and verifying hypotheses for each of those types. That is why the instance presented in Section 6.1 is intended primarily for manual proofs.

Despite these difficulties, typechecking is certainly not out of reach, as we show. First, we demonstrate that the set of types is finite in significant cases, by providing an example in Section 6.2. Moreover, the logic-programming protocol checker of Section 7 yields a practical implementation in cases in which the sets are infinite.

Finally, having a very general (infinitary) type system strengthens our relative completeness result of Section 7.3. This result shows that the checker can prove all secrecy properties that can be proved with any instance of the type system, even infinitary instances.

5 Properties of the Type System

Next we study the properties of the type system. We first establish a subject-reduction result and other basic lemmas, then use these results for proving a theorem about secrecy. Technically, we follow the same pattern as in the special case (protocols with asymmetric communication) treated in our previous work [5], but some of the proofs require new arguments.

5.1 Subject Reduction and Typability

Lemma 5.1.1 (Substitution) *If* $E, E' \vdash M : T$ *and* $E, x : T, E' \vdash M' : T'$ *then* $E, E' \vdash M'\{M/x\} : T'$. *If* $E, E' \vdash M : T$ *and* $E, x : T, E' \vdash P$ *then* $E, E' \vdash P\{M/x\}$.

Proof The proof is by induction on the derivations of $E, x : T, E' \vdash M' : T'$ and of $E, x : T, E \vdash P$. The treatment of all rules is straightforward. □

Lemma 5.1.2 (Subject congruence) *If* $E \vdash P$ *and* $P \equiv Q$ *then* $E \vdash Q$.

Proof This proof is similar to the corresponding proof for the type system of Cardelli, Ghelli, and Gordon [20]; it is an easy induction on the derivation of $P \equiv Q$. In the case of scope extrusion, we use a weakening lemma, which is easy to prove by induction on type derivations. □

The subject-reduction lemma says that typing is preserved by computation.

Lemma 5.1.3 (Subject reduction) *If* $E \vdash P$ *and* $P \rightarrow Q$ *then* $E \vdash Q$.

Proof The proof is by induction on the derivation of $P \rightarrow Q$.

- In the case of (Red I/O), we have

$$\overline{a}\langle M\rangle.Q \mid a(x).P \rightarrow Q \mid P\{M/x\}$$

We assume $E \vdash \overline{a}\langle M\rangle.Q \mid a(x).P$. This judgment must have been derived using the rule (Parallel composition), so $E \vdash \overline{a}\langle M\rangle.Q$ and $E \vdash a(x).P$. The judgment $E \vdash a(x).P$ must have been derived by (Input) from $E \vdash a : T$ and $\forall T' \in \mathit{conveys}(T), E, x : T' \vdash P$ for some T. The judgment $E \vdash \overline{a}\langle M\rangle.Q$ must have been derived by (Output) from $E \vdash a : T$ (for the same T as in the (Input) derivation, since each term has at most one type), $E \vdash M : T'$, $T' \in \mathit{conveys}(T)$, and $E \vdash Q$. By the substitution lemma (Lemma 5.1.1), we obtain $E \vdash P\{M/x\}$. By (Parallel composition), $E \vdash Q \mid P\{M/x\}$.

- In the case of (Red Destr 1), we have $g(M_1, \ldots, M_n) = M'$ and

$$\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow P\{M'/x\}$$

We assume $E \vdash \mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$. This judgment must have been derived by (Destructor application) from $\forall i \in \{1, \ldots, n\}, E \vdash M_i : T_i$, and $\forall T \in O_g(T_1, \ldots, T_n), E, x : T \vdash P$ for some $T_1, \ldots, T_n$. There exists an equation $g(N_1, \ldots, N_n) = N'$ in $\mathrm{def}(g)$ and a substitution σ such that $\forall i \in \{1, \ldots, n\}, M_i = \sigma N_i$ and $M' = \sigma N'$. For each variable x_j that occurs in $N_1, \ldots, N_n$, we have a subterm σx_j of $M_1, \ldots, M_n$, and a type T_{x_j} must have been given to this subterm when typing $M_1, \ldots, M_n$, so we have $E \vdash \sigma x_j : T_{x_j}$ for each variable x_j that occurs in $N_1, \ldots, N_n$. (All occurrences of each variable x_j have the same type, since each term has at most one type.) Since $E \vdash \sigma N_i : T_i$, we have $E' \vdash N_i : T_i$ where E' is the environment that associates each variable x_j with the type T_{x_j}. Since $g(N_1, \ldots, N_n) = N'$ is in $\mathrm{def}(g)$, by (P3), there exists $T \in O_g(T_1, \ldots, T_n)$, such that $E' \vdash N' : T$. By the substitution lemma (Lemma 5.1.1), $E \vdash M' : T$. Since $E, x : T \vdash P$, the substitution lemma yields $E \vdash P\{M'/x\}$.

- The cases (Red Let) and (Red Cond 1) follow by (Red Destr 1). (Recall that local definitions and conditionals can be encoded as destructor applications.)

The remaining cases are easy. □

In the study of programming languages, it is common to complement subject-reduction properties with progress properties. A typical progress property says that well-typed programs do not get stuck as a result of dynamic type errors—for example, attempting to multiply a boolean and an integer. Dynamic type errors are meaningful whenever the language's execution model includes dynamic type information, such as different tags on booleans and integers. Without such tags, on the other hand, the representations of booleans and integers may well be multiplied, though the result of such an operation will typically be implementation-dependent.

In our context, by analogy, one might consider stating a progress property that would guarantee that no "dynamic secrecy-type error" causes a process to get stuck. A "dynamic secrecy-type error" might be using public data as non-public data, or vice versa. Like ordinary dynamic type errors, "dynamic secrecy-type errors" are meaningful if the execution model includes dynamic type information, in this case tags that indicate secrecy types. The operational semantics of our process calculus does not however include such tags. Indeed, it is deliberately independent of any secrecy information. Furthermore, our typings do not imply any progress property: as the following typability lemma says, every process is well-typed (at least in a fairly trivial way that makes its free names and free variables public).

Lemma 5.1.4 (Typability) *Let P be an untyped process. If $fn(P) \subseteq \{a_1, \dots, a_n\}$, $fv(P) \subseteq \{x_1, \dots, x_m\}$, $T'_i \in T_{Public}$ for all $i \in \{1, \dots, n\}$, and $T_i \in T_{Public}$ for all $i \in \{1, \dots, m\}$, then*

$$a_1 : T'_1, \dots, a_n : T'_n, x_1 : T_1, \dots, x_m : T_m \vdash P$$

Proof We first prove by induction that all terms are of a type in T_{Public}; that is:

$$a_1 : T'_1, \dots, a_n : T'_n, x_1 : T_1, \dots, x_m : T_m \vdash M : T$$

with $T \in T_{\text{Public}}$ if $fn(M) \subseteq \{a_1, \dots, a_n\}$, $fv(M) \subseteq \{x_1, \dots, x_m\}$, $T'_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, n\}$, and $T_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, m\}$.

- For names and variables, this follows by Rule (Atom).
- For composite terms $f(M_1, \dots, M_k)$, this follows by Rule (Constructor application) and induction hypothesis, since if $T''_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, k\}$, then $O_f(T''_1, \dots, T''_k) \in T_{\text{Public}}$ by (P1).

Now we prove the claim, by induction on the structure of P.

- For output, notice that if $T \in T_{\text{Public}}$, then $T_{\text{Public}} \subseteq conveys(T)$ by (P0).
- For input, if $T \in T_{\text{Public}}$, then $T_{\text{Public}} \supseteq conveys(T)$ by (P0).
- In the case of restriction, we let the type of the new name be in T_{Public}.
- For destructor application, notice that if $T''_i \in T_{\text{Public}}$ for all $i \in \{1, \dots, k\}$, then $T \in T_{\text{Public}}$ for all $T \in O_g(T''_1, \dots, T''_k)$, by (P2).

□

This typability lemma is important because it means that any process that represents an adversary is well-typed. It is a formal counterpart to the informal idea that the type system cannot constrain the adversary.

5.2 Secrecy

The secrecy theorem says that if a closed process P is well-typed in an environment E, and a name s is not of a type in T_{Public} according to E, then P preserves the secrecy of s from S, where S is the set of names that are of a type in T_{Public} according to E. In other words, P preserves the secrecy of names whose type is not in T_{Public} against adversaries that can output, input, and compute on names of types in T_{Public}.

Theorem 5.2.1 (Secrecy) *Let P be a closed process. Suppose that $E \vdash P$, $E \vdash s : T'$, and $T' \notin T_{Public}$. Let $S = \{a \mid E \vdash a : T \text{ and } T \in T_{Public}\}$. Then P preserves the secrecy of s from S.*

This secrecy theorem is a consequence of the subject-reduction lemma and the typability lemma.

Proof Suppose that $S = \{a_1, \dots, a_l\}$, let T_i be the type of a_i, so $(a_i : T_i) \in E$ with $T_i \in T_{\text{Public}}$.

In order to derive a contradiction, we assume that P does not preserve the secrecy of s from S. Then there exists a process Q with $fv(Q) = \emptyset$ and $fn(Q) \subseteq S$, such that $P \mid Q \rightarrow^* R$ and $R \equiv \overline{c}\langle s \rangle.Q' \mid R'$, where $c \in S$. By Lemma 5.1.4, $a_1 : T_1, \dots, a_l : T_l \vdash Q$, so $E \vdash Q$. Therefore, $E \vdash P \mid Q$. By Lemma 5.1.3, $E \vdash R$, and by Lemma 5.1.2, $E \vdash \overline{c}\langle s \rangle.Q' \mid R'$. Since $c \in S$, we have $E \vdash c : T$ and $T \in T_{\text{Public}}$ for some T. The judgment $E \vdash \overline{c}\langle s \rangle.Q'$ must be derived by (Output) from $E \vdash c : T$ and $E \vdash s : T'$ with $T' \in \mathit{conveys}(T)$. Furthermore, $T' \in T_{\text{Public}}$ by (P0), contradicting the hypotheses of the theorem. So P preserves the secrecy of s from S. □

We restate a special case of the theorem, as it may be particularly clear.

Corollary 5.2.2 *Suppose that $a : T, s : T' \vdash P$ with $T \in T_{Public}$ and $T' \notin T_{Public}$. Then P preserves the secrecy of s from a. That is, for all closed processes Q such that $fn(Q) \subseteq \{a\}$, $P \mid Q$ does not output s on a.*

Suppose that the secrecy theorem implies that a process P preserves the secrecy of two names s and s', treating each of these names separately. The two applications of the secrecy theorem may in general rely on two different ways of showing that P is well-typed, with two different typing environments E and E'. We must have that $E \vdash s : T$ and $E' \vdash s' : T'$ for some types $T, T' \notin T_{\text{Public}}$. However, we may also have that $E \vdash s' : T_1$ and $E' \vdash s : T_1'$ for some types $T_1, T_1' \in T_{\text{Public}}$. Ideally, we may like to have a single environment E such that $E \vdash P$, $E \vdash s : T$, and $E \vdash s' : T'$ with $T, T' \notin T_{\text{Public}}$. Thus, E would make secret as much as possible, providing a "most secret typing" for P. In general, most secret typings are not always possible. For example, the instance of Section 6.1 does not guarantee the existence of most secret typings. (The proof is very similar to that in [5, Section 5.3].) In contrast, in the instance of Section 7, the types generated by the verifier yield most secret typings.

6 Some Instances of the Type System

As an important example, we show how the general type system applies to symmetric and asymmetric encryption. Specifically, we show two instances of the general type system, one infinitary and the other finitary, for processes that use symmetric and asymmetric encryption, built using the constructors *ntuple*, *sencrypt*, *pencrypt*, and *pk*, and the corresponding destructors ith_n, *sdecrypt*, and *pdecrypt*, introduced in Section 2. These instances are similar in scope and power to previous special-purpose type systems [1, 5], but they treat additional constructs and could easily treat even more.

In both instances, we include types for public and secret data, Public and Secret (as well as types with more structure, such as certain types for tuples). It would be straightforward to extend the instances with additional levels of secrecy, for example introducing an extra type

$T ::=$	types
Public	public data
Secret	secret data
$T_1 \times \ldots \times T_n$	tuple
$\mathrm{C}[T]$	secret channel
$\mathrm{K}^{\mathrm{Secret}}[T]$	secret shared key
$\mathrm{DK}^{\mathrm{Secret}}[T]$	decryption key whose corresponding encryption key is secret
$\mathrm{EK}^{\mathrm{Secret}}[T]$	secret encryption key
$\mathrm{DK}^{\mathrm{Public}}[T]$	decryption key whose corresponding encryption key is public
$\mathrm{EK}^{\mathrm{Public}}[T]$	public encryption key

Figure 4: Grammar of types in an instance of the type system

TopSecret. Our results carry over to such extensions, and they can imply, for example, that even when data of type Secret is allowed to become public (by letting Secret $\in T_{\mathrm{Public}}$), data of type TopSecret need not be. Such results are perhaps reminiscent of classic work on multilevel security (e.g., [27]). However, unlike in that classic work, we need not require that the types form a lattice. We also have different concerns (for example, protecting against network attacks rather than against Trojan horses), and obtain different security guarantees (since our definition of secrecy does not preclude all information flows).

6.1 An Infinitary Instance

For the first instance, the grammar of types is given in Figure 4. Informally, types have the following meanings:

- Public is the type of public data.
- Secret is the type of secret data.
- $T_1 \times \ldots \times T_n$ is the type of tuples, whose components are of types $T_1, \ldots, T_n$.
- $\mathrm{C}[T]$ is the type of a channel that can convey data of type T and that cannot be known by the adversary. (Channels that can be known by the adversary are of type Public.) Channel types are ordinary data types, so channel names can be encrypted and can be sent in messages.
- $\mathrm{K}^{\mathrm{Secret}}[T]$ is the type of symmetric keys that can be used to encrypt data of type T and that cannot be known by the adversary. (Symmetric keys that can be known by the adversary are of type Public.)
- $\mathrm{EK}^{\mathrm{Secret}}[T]$ is the type of secret asymmetric encryption keys that can be used to encrypt cleartexts of type T.
- $\mathrm{DK}^{\mathrm{Secret}}[T]$ is the type of asymmetric decryption keys for cleartexts of type T and such that the corresponding encryption keys are secret. These decryption keys are also secret.
- $\mathrm{EK}^{\mathrm{Public}}[T]$ is the type of public asymmetric encryption keys that can be used to encrypt cleartexts of type T. The adversary can use these keys to encrypt its messages, so public messages can also be encrypted under these keys.

$$T_{\mathrm{Public}} = \{T \mid T \leq \mathrm{Public}\}$$
$$= \{\mathrm{Public}, \mathrm{EK}^{\mathrm{Public}}[T]\} \cup \{T_1 \times \ldots \times T_n \mid \forall i \in \{1, \ldots, n\}, T_i \in T_{\mathrm{Public}}\}.$$

If $T \leq \mathrm{Public}$, then $\mathit{conveys}(T) = T_{\mathrm{Public}}$;
$\mathit{conveys}(\mathrm{C}[T]) = \{T' \mid T' \leq T\}$.

$O_{ntuple}(T_1, \ldots, T_n) = T_1 \times \ldots \times T_n$.
If $T_1 \leq \mathrm{Public}$ and $T_2 \leq \mathrm{Public}$, then $O_{sencrypt}(T_1, T_2) = \mathrm{Public}$;
if $T' \leq T$, then $O_{sencrypt}(T', \mathrm{K}^{\mathrm{Secret}}[T]) = \mathrm{Public}$.
If $T_1 \leq \mathrm{Public}$ and $T_2 \leq \mathrm{Public}$, then $O_{pencrypt}(T_1, T_2) = \mathrm{Public}$;
if $T' \leq T$, then $O_{pencrypt}(T', \mathrm{EK}^{L}[T]) = \mathrm{Public}$.
If $T_1 \leq \mathrm{Public}$, then $O_{pk}(T_1) = \mathrm{Public}$;
$O_{pk}(\mathrm{DK}^{L}[T]) = \mathrm{EK}^{L}[T]$.

$O_{ith_n}(T_1 \times \ldots \times T_n) = \{T_i\}$.
If $T \leq \mathrm{Public}$, then $O_{sdecrypt}(\mathrm{Public}, T) = T_{\mathrm{Public}}$;
$O_{sdecrypt}(\mathrm{Public}, \mathrm{K}^{\mathrm{Secret}}[T]) = \{T' \mid T' \leq T\}$.
If $T \leq \mathrm{Public}$, then $O_{pdecrypt}(\mathrm{Public}, T) = T_{\mathrm{Public}}$;
$O_{pdecrypt}(\mathrm{Public}, \mathrm{DK}^{\mathrm{Secret}}[T]) = \{T' \mid T' \leq T\}$;
$O_{pdecrypt}(\mathrm{Public}, \mathrm{DK}^{\mathrm{Public}}[T]) = \{T' \mid T' \leq T\} \cup T_{\mathrm{Public}}$.

Other cases: $\mathit{conveys}(T) = \emptyset$, $O_f(T_1, \ldots, T_n)$ is undefined, $O_g(T_1, \ldots, T_n) = \emptyset$.

Figure 5: Definition of T_{Public} and type operators in an instance of the type system

- $\mathrm{DK}^{\mathrm{Public}}[T]$ is the type of asymmetric decryption keys for cleartexts of type T and such that the corresponding encryption keys are public. These decryption keys are however secret. When decrypting a message with such a key, the result can be of type T (in normal use of the key) or of type Public (when the adversary has used the corresponding encryption key to encrypt one of its messages).

We define T_{Public} and the type operators of the system in Figure 5. For this purpose, we let the subtyping relation $\leq$ be reflexive and transitive, with

$$\begin{aligned}
&\mathrm{C}[T] \leq \mathrm{Secret},\\
&\mathrm{K}^{\mathrm{Secret}}[T] \leq \mathrm{Secret},\\
&\mathrm{DK}^{\mathrm{Secret}}[T] \leq \mathrm{Secret},\\
&\mathrm{EK}^{\mathrm{Secret}}[T] \leq \mathrm{Secret},\\
&\mathrm{DK}^{\mathrm{Public}}[T] \leq \mathrm{Secret},\\
&\mathrm{EK}^{\mathrm{Public}}[T] \leq \mathrm{Public},\\
&\mathrm{Public} \times \ldots \times \mathrm{Public} \leq \mathrm{Public},\\
&\text{if } \exists i \in \{1, \ldots, n\}, T_i = \mathrm{Secret} \text{ then } T_1 \times \ldots \times T_n \leq \mathrm{Secret},\\
&\text{if } T_1 \leq T_1', \ldots, T_n \leq T_n' \text{ then } T_1 \times \ldots \times T_n \leq T_1' \times \ldots \times T_n'.
\end{aligned}$$

Importantly, the definitions allow encryption under a public key of type $\mathrm{EK}^{\mathrm{Public}}[T]$ to accept data both of type Public and of type T. For the corresponding decryption, we handle both cases: $O_{pdecrypt}(\mathrm{Public}, \mathrm{DK}^{\mathrm{Public}}[T])$ includes both subtypes of T and subtypes of Public. (A similar idea appears in the special-purpose type system of [5].) As explained above, we do not need a "subsumption" rule.

Proposition 6.1.1 *These definitions satisfy the constraints of the general type system (P0, P1, P2, P3).*

Proof (P0), (P1), and (P2) are obvious. We prove (P3).

- $ith_n(ntuple(M_1, \ldots, M_n)) = M_i$. Suppose that $E \vdash ntuple(M_1, \ldots, M_n) : T$. This judgment must have been derived by (Constructor application). Therefore, $T = T_1 \times \ldots \times T_n$ and $E \vdash M_i : T_i$, with $T_i \in O_{ith_n}(T)$.

- $sdecrypt(sencrypt(M, N), N) = M$. Suppose that $E \vdash sencrypt(M, N) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by (Constructor application). Therefore, $E \vdash M : T$ and $O_{sencrypt}(T, T_2) = T_1 = \mathrm{Public}$ for some T. By definition of $O_{sencrypt}$, we have two cases.

  In case $T \leq \mathrm{Public}$ and $T_2 \leq \mathrm{Public}$, we obtain $E \vdash M : T$ and $T \in T_{\mathrm{Public}} = O_{sdecrypt}(\mathrm{Public}, T_2)$.

  Otherwise, $T_2 = \mathrm{K}^{\mathrm{Secret}}[T']$ with $T \leq T'$, so $E \vdash M : T$ and $T \in O_{sdecrypt}(\mathrm{Public}, T_2)$.

- $pdecrypt(pencrypt(M, pk(N)), N) = M$. Suppose that $E \vdash pencrypt(M, pk(N)) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by applying (Constructor application) twice, from $E \vdash M : T$ with $O_{pencrypt}(T, O_{pk}(T_2)) = T_1 = \mathrm{Public}$ for some T. By definition of O_{pk}, we have three cases.

  In case $T_2 \leq \mathrm{Public}$, we have $O_{pk}(T_2) = \mathrm{Public}$. Moreover, since $O_{pencrypt}(T, \mathrm{Public}) = \mathrm{Public}$, we also have $T \in T_{\mathrm{Public}}$. Thus, $E \vdash M : T$ and $T \in T_{\mathrm{Public}} = O_{pdecrypt}(\mathrm{Public}, T_2)$.

  In case $T_2 = \mathrm{DK}^{\mathrm{Secret}}[T']$, we have $O_{pk}(T_2) = \mathrm{EK}^{\mathrm{Secret}}[T']$. Moreover, since $O_{pencrypt}(T, \mathrm{EK}^{\mathrm{Secret}}[T']) = \mathrm{Public}$, we also have $T \leq T'$. Thus, $E \vdash M : T$ and $T \in O_{pdecrypt}(\mathrm{Public}, T_2)$.

Otherwise, $T_2 = \mathrm{DK}^{\mathrm{Public}}[T']$, and we have $O_{pk}(T_2) = \mathrm{EK}^{\mathrm{Public}}[T']$. Moreover, since $O_{pencrypt}(T, \mathrm{EK}^{\mathrm{Public}}[T']) = \mathrm{Public}$, we also have $T \leq T'$ or $T \in \mathcal{T}_{\mathrm{Public}}$. We obtain $E \vdash M : T$ and $T \in O_{pdecrypt}(\mathrm{Public}, T_2)$.

□

As an immediate corollary, Theorem 5.2.1 applies, so we can prove secrecy by typing. For example, the type system can be used to establish that s remains secret in the process P of the example protocol of Section 2.2. For this proof, we define $E \triangleq s : \mathrm{Secret}, e : \mathrm{Public}$, and derive $E \vdash P$. In the (Restriction) rule, we choose the types

$$T_{sK_A} \triangleq \mathrm{DK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$$

for sK_A and

$$T_{sK_B} \triangleq \mathrm{DK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]]$$

for sK_B. Then $pk(sK_A)$ has the type

$$\begin{aligned} T_{pK_A} &\triangleq O_{pk}(T_{sK_A}) \\ &= \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]] \end{aligned}$$

and $pk(sK_B)$ has the type

$$\begin{aligned} T_{pK_B} &\triangleq O_{pk}(T_{sK_B}) \\ &= \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]] \end{aligned}$$

The remainder of the process is typed in the environment:

$$\begin{aligned} E' \triangleq E, &sK_A : \mathrm{DK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]], \\ &sK_B : \mathrm{DK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]], \\ &pK_A : \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]], \\ &pK_B : \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]] \end{aligned}$$

We have that $T_{pK_A} \in \mathit{conveys}(\mathrm{Public})$ and $T_{pK_B} \in \mathit{conveys}(\mathrm{Public})$ (since these types are subtypes of Public). Then we only have to show that $E' \vdash A$ and $E' \vdash B$. In the typing of A, we choose k of type Secret. Then

$$E', k : \mathrm{Secret} \vdash \mathit{pencrypt}((k, pK_A), pK_B) : \mathrm{Public}$$

follows by (Constructor application), so the output $\overline{e}\langle \mathit{pencrypt}((k, pK_A), pK_B)\rangle$ is well-typed by (Output). In the input $e(z)$, by (Input), z can be of any subtype of Public, then by (Destructor application), we have to prove $E', k : \mathrm{Secret}, x : T_x, y : T_y \vdash \mathit{if}\ x = k\ \mathit{then}\ \overline{e}\langle \mathit{sencrypt}(s, y)\rangle$, where either $T_x \leq \mathrm{Secret}$ and $T_y \leq \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$ or $T_x \leq \mathrm{Public}$ and $T_y \leq \mathrm{Public}$.

- In the first case, the conditional is well-typed, since the output is well-typed.
- In the second case, the conditional is well-typed, since x and k cannot have the same type.

For typing B, by (Input), the type of z is a subtype of Public. By (Destructor application), we have to show that

$$E', x : T_x, y : T_y \vdash (\nu K_{AB}) \begin{pmatrix} \overline{e}\langle \mathit{pencrypt}((x, K_{AB}), y)\rangle. \\ e(z').\mathit{let}\ s' = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0 \end{pmatrix}$$

where either $T_x \leq \mathrm{Secret}$ and $T_y \leq \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$, or $T_x \leq \mathrm{Public}$ and $T_y \leq \mathrm{Public}$.

- In the first case, we choose K_{AB} of type $\mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$. We have $T_x \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}] \leq \mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$, and $T_y = \mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$ (the only subtype of $\mathrm{EK}^{\mathrm{Public}}[\mathrm{Secret} \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$ is itself), so $O_{pencrypt}(T_x \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}], T_y) = \mathrm{Public}$.

- In the second case, we choose K_{AB} of type Public. We have $T_x \times \mathrm{Public} \leq \mathrm{Public}$ and $T_y \leq \mathrm{Public}$, therefore $O_{pencrypt}(T_x \times \mathrm{Public}, T_y) = \mathrm{Public}$.

In both cases, it follows that the encryption is of type Public by (Constructor application), and that the output is well-typed. In both cases, also, the input $e(z').let\ s' = sdecrypt(z', K_{AB})\ in\ 0$ is clearly well-typed. Thus, we obtain $E \vdash P$. Finally, by Theorem 5.2.1, we conclude that P preserves the secrecy of s from $\{e\}$.

As for the process P' of Section 2.2, we cannot show that it preserves the secrecy of s_A or s_B from $\{e\}$ using this instance of the type system. This difficulty stems from two different uses of the key sK_A, which appear because A plays both roles in the protocol. (Indeed, if s_A or s_B were of type Secret, then y would be of type $\mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$ in A', so sK_A would have a type of the form $\mathrm{DK}^{\mathrm{Public}}[T \times \mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]]$, and y could be of type $\mathrm{K}^{\mathrm{Secret}}[\mathrm{Secret}]$ in B'' in conflict with the use of y as public encryption key.) In Section 6.2, we present a variant that avoids this difficulty. We postpone the formal analysis of the process P' itself to Section 7.

6.2 A Finitary Instance

Typechecking may be difficult, or at least non-trivial, in infinitary instances such as the one of Section 6.1, in which the type rules contain universal quantifications over infinite sets of types. In this section, we present a weaker instance that deals with the same function symbols but uses only a finite number of types. For this finitary instance, automatic typechecking and type inference are easy by exhaustive exploration of all typings.

The set of types of this instance is:

$$Types = \{\mathrm{Public}, \mathrm{Secret}, \mathrm{EK}^{\mathrm{Public}}, \text{Public-Secret-}\mathrm{EK}^{\mathrm{Public}}\}$$

These types have the following meanings:

- Public is the type of public data and Secret is the type of secret data, as in the previous instance.

- $\mathrm{EK}^{\mathrm{Public}}$ is the type of public asymmetric encryption keys such that the corresponding decryption keys are secret.

- Public-Secret-$\mathrm{EK}^{\mathrm{Public}}$ is the type of triples whose first component is of type Public, second component is of type Secret, and third component is of type $\mathrm{EK}^{\mathrm{Public}}$.

We define T_{Public} and the type operators of the system in Figure 6. The resulting instance of the type system has similarities with the special-purpose type system of [1]. However, that type system does not handle public-key encryption; more importantly, it establishes a different notion of secrecy (a form of non-interference), and accordingly its typing of tests is more restrictive in order to prevent the so-called "implicit" information flows.

Proposition 6.2.1 *These definitions satisfy the constraints of the general type system (P0, P1, P2, P3).*

Proof (P0), (P1), and (P2) are obvious. We prove (P3).

- $ith_n(ntuple(M_1, \ldots, M_n)) = M_i$. Suppose that $E \vdash ntuple(M_1, \ldots, M_n) : T$. This judgment must have been derived by (Constructor application), so we have four cases, one for each case in the definition of O_{ntuple}.

$T_{\text{Public}} = \{\text{EK}^{\text{Public}}, \text{Public}\}$.

If $T \in T_{\text{Public}}$, then $\mathit{conveys}(T) = T_{\text{Public}}$;
$\mathit{conveys}(\text{Secret}) = \{\text{Public-Secret-EK}^{\text{Public}}\}$.

If $T \in T_{\text{Public}}$, then $O_{3tuple}(T, \text{Secret}, \text{EK}^{\text{Public}}) = \text{Public-Secret-EK}^{\text{Public}}$;
$O_{ntuple}(\text{Secret}, \ldots, \text{Secret}) = \text{Secret}$;
$O_{ntuple}(\text{EK}^{\text{Public}}, \ldots, \text{EK}^{\text{Public}}) = \text{EK}^{\text{Public}}$;
if $T_1, \ldots, T_n \in T_{\text{Public}}$ and there exists $i \in \{1, \ldots, n\}$ such that $T_i \neq \text{EK}^{\text{Public}}$, then $O_{ntuple}(T_1, \ldots, T_n) = \text{Public}$.
$O_{sencrypt}(\text{Public-Secret-EK}^{\text{Public}}, \text{Secret}) = \text{Public}$;
if $T_1, T_2 \in T_{\text{Public}}$, then $O_{sencrypt}(T_1, T_2) = \text{Public}$.
$O_{pencrypt}(\text{Public-Secret-EK}^{\text{Public}}, \text{EK}^{\text{Public}}) = \text{Public}$;
if $T_1, T_2 \in T_{\text{Public}}$, then $O_{pencrypt}(T_1, T_2) = \text{Public}$.
$O_{pk}(\text{Secret}) = \text{EK}^{\text{Public}}$;
if $T_1 \in T_{\text{Public}}$, then $O_{pk}(T_1) = \text{Public}$.

$O_{ith_n}(\text{EK}^{\text{Public}}) = \{\text{EK}^{\text{Public}}\}$;
$O_{ith_n}(\text{Public}) = T_{\text{Public}}$;
$O_{ith_n}(\text{Secret}) = \{\text{Secret}\}$;
$O_{1th_3}(\text{Public-Secret-EK}^{\text{Public}}) = T_{\text{Public}}$;
$O_{2th_3}(\text{Public-Secret-EK}^{\text{Public}}) = \{\text{Secret}\}$;
$O_{3th_3}(\text{Public-Secret-EK}^{\text{Public}}) = \{\text{EK}^{\text{Public}}\}$.
If $T \in T_{\text{Public}}$, then $O_{sdecrypt}(\text{Public}, T) = T_{\text{Public}}$;
$O_{sdecrypt}(\text{Public}, \text{Secret}) = \{\text{Public-Secret-EK}^{\text{Public}}\}$.
If $T \in T_{\text{Public}}$, then $O_{pdecrypt}(\text{Public}, T) = T_{\text{Public}}$;
$O_{pdecrypt}(\text{Public}, \text{Secret}) = \{\text{Public-Secret-EK}^{\text{Public}}, \text{EK}^{\text{Public}}, \text{Public}\}$.

Other cases: $\mathit{conveys}(T) = \emptyset$, $O_f(T_1, \ldots, T_n)$ is undefined, $O_g(T_1, \ldots, T_n) = \emptyset$.

Figure 6: Definition of T_{Public} and type operators in another instance of the type system

1. $n = 3$, $T = \text{Public-Secret-EK}^{\text{Public}}$, $E \vdash M_1 : T'$ with $T' \in T_{\text{Public}}$, $E \vdash M_2 : \text{Secret}$, $E \vdash M_3 : \text{EK}^{\text{Public}}$, and $T' \in T_{\text{Public}} = O_{1th_3}(T)$, $\text{Secret} \in O_{2th_3}(T)$, $\text{EK}^{\text{Public}} \in O_{3th_3}(T)$.
2. $T = \text{Secret}$, $E \vdash M_i : \text{Secret}$, and $\text{Secret} \in O_{ith_n}(\text{Secret})$.
3. $T = \text{EK}^{\text{Public}}$, $E \vdash M_i : \text{EK}^{\text{Public}}$, and $\text{EK}^{\text{Public}} \in O_{ith_n}(\text{EK}^{\text{Public}})$.
4. $T = \text{Public}$, $E \vdash M_i : T_i$ with $T_i \in T_{\text{Public}}$, and $T_i \in T_{\text{Public}} = O_{ith_n}(\text{Public})$.

- $\mathit{sdecrypt}(\mathit{sencrypt}(M, N), N) = M$. Suppose that $E \vdash \mathit{sencrypt}(M, N) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by (Constructor application). Therefore, $E \vdash M : T$ and $O_{\mathit{sencrypt}}(T, T_2) = T_1 = \text{Public}$ for some T. By definition of O_{sencrypt}, we have two cases.

  In case $T, T_2 \in T_{\text{Public}}$, we obtain $E \vdash M : T$ and $T \in T_{\text{Public}} = O_{\mathit{sdecrypt}}(\text{Public}, T_2)$.

  Otherwise, $T_2 = \text{Secret}$ and $T = \text{Public-Secret-EK}^{\text{Public}}$, so $E \vdash M : T$ and $T \in O_{\mathit{sdecrypt}}(\text{Public}, T_2)$.

- $\mathit{pdecrypt}(\mathit{pencrypt}(M, \mathit{pk}(N)), N) = M$. Suppose that $E \vdash \mathit{pencrypt}(M, \mathit{pk}(N)) : T_1$ and $E \vdash N : T_2$. The former judgment must have been derived by applying (Constructor application) twice, from $E \vdash M : T$ with $O_{\mathit{pencrypt}}(T, O_{\mathit{pk}}(T_2)) = T_1 = \text{Public}$ for some T. By definition of O_{pk}, we have two cases.

  In case $T_2 \in T_{\text{Public}}$, we have $O_{\mathit{pk}}(T_2) = \text{Public}$. Moreover, since $O_{\mathit{pencrypt}}(T, \text{Public}) = \text{Public}$, we also have $T \in T_{\text{Public}}$. Thus, $E \vdash M : T$ and $T \in T_{\text{Public}} = O_{\mathit{pdecrypt}}(\text{Public}, T_2)$.

  Otherwise, $T_2 = \text{Secret}$, and we have $O_{\mathit{pk}}(T_2) = \text{EK}^{\text{Public}}$. Moreover, since $O_{\mathit{pencrypt}}(T, \text{EK}^{\text{Public}}) = \text{Public}$, we also have $T = \text{Public-Secret-EK}^{\text{Public}}$ or $T \in T_{\text{Public}}$. We obtain $E \vdash M : T$ and $T \in O_{\mathit{pdecrypt}}(\text{Public}, T_2)$.

□

The process P of Section 2.2 clearly does not typecheck in this type system, since the type system supports encryption of only public data and triples, and the protocol uses encryption of pairs containing secrets. This point illustrates that this instance is more restrictive than the instance of the previous section. We can however adapt the protocol to obtain a similar protocol that does typecheck. More precisely, we modify the encryptions so that their cleartexts are always of type $\text{Public-Secret-EK}^{\text{Public}}$. Thus the protocol becomes:

$$\begin{array}{ll}
\text{Message 1.} & A \to B : \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), pK_B) \\
\text{Message 2.} & B \to A : \mathit{pencrypt}((a'_{\text{Public}}, (k, K_{AB}), a'_{\text{EK}^{\text{Public}}}), pK_A) \\
\text{Message 3.} & A \to B : \mathit{sencrypt}((a''_{\text{Public}}, s, a''_{\text{EK}^{\text{Public}}}), K_{AB})
\end{array}$$

where a_{Public} and similar fields indicate arbitrary padding of the appropriate types. This protocol can be represented by the following process:

$$\begin{array}{rcl}
P & \triangleq & (\nu sK_A)(\nu sK_B) \mathit{let}\ pK_A = pk(sK_A)\ \mathit{in} \\
& & \quad \mathit{let}\ pK_B = pk(sK_B)\ \mathit{in}\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(A \mid B) \\
A & \triangleq & (\nu k)(\nu a_{\text{Public}})\overline{e}\langle \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), pK_B)\rangle. \\
& & \quad e(z).\mathit{let}\ (x'_1, (x, y), x'_3) = \mathit{pdecrypt}(z, sK_A)\ \mathit{in} \\
& & \quad \mathit{if}\ x = k\ \mathit{then}\ (\nu a''_{\text{Public}})(\nu a''_{\text{EK}^{\text{Public}}}) \\
& & \quad \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s, a''_{\text{EK}^{\text{Public}}}), y)\rangle \\
B & \triangleq & e(z).\mathit{let}\ (x_1, x, y) = \mathit{pdecrypt}(z, sK_B)\ \mathit{in} \\
& & \quad (\nu K_{AB})(\nu a'_{\text{Public}})(\nu a'_{\text{EK}^{\text{Public}}}) \\
& & \quad \overline{e}\langle \mathit{pencrypt}((a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}}), y)\rangle. \\
& & \quad e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0
\end{array}$$

This process is typable in this instance of the type system: letting $E \triangleq s : \text{Secret}, e : \text{Public}$, we can show that $E \vdash P$. In the (Restriction) rule, we choose the type Secret for sK_A and sK_B. Then $pk(sK_A)$ and $pk(sK_B)$ have the type $O_{pk}(\text{Secret}) = \text{EK}^{\text{Public}}$. The remainder of the process is typed in the environment:

$$E' \triangleq E, sK_A : \text{Secret}, sK_B : \text{Secret}, pK_A : \text{EK}^{\text{Public}}, pK_B : \text{EK}^{\text{Public}}$$

We check that $\text{EK}^{\text{Public}} \in \mathit{conveys}(\text{Public})$ (since this type is in T_{Public}), so the outputs $\overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle$ are well-typed. Then we only have to show that $E' \vdash A$ and $E' \vdash B$. In the typing of A, we choose k of type Secret, a_{Public} of type Public. Then

$$E', k : \text{Secret}, a_{\text{Public}} : \text{Public} \vdash \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), pK_B) : \text{Public}$$

follows by (Constructor application), so the output $\overline{e}\langle \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), pK_B)\rangle$ is well-typed by (Output). In the input $e(z)$, by (Input), z can be of type Public or $\text{EK}^{\text{Public}}$, then by (Destructor application), $(x'_1, (x, y), x'_3)$ can be of type Public-Secret-$\text{EK}^{\text{Public}}$, Public, or $\text{EK}^{\text{Public}}$, so (x, y) can be of type Secret, Public, or $\text{EK}^{\text{Public}}$, hence we have to prove $E', k : \text{Secret}, x : T_x, y : T_y, a''_{\text{Public}} : \text{Public}, a''_{\text{EK}^{\text{Public}}} : \text{EK}^{\text{Public}} \vdash \mathit{if}\ x = k\ \mathit{then}\ \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s, a''_{\text{EK}^{\text{Public}}}), y)\rangle$, where either $T_x = T_y = \text{Secret}$ or $T_x, T_y \in T_{\text{Public}}$.

- In the first case, the conditional is well-typed, since the output is well-typed.
- In the second case, the conditional is well-typed, since x and k cannot have the same type.

For typing B, by (Input), the type of z is in T_{Public}. By (Destructor application), we have to show that

$$\begin{aligned} E', x : T_x, y : T_y \vdash\ & (\nu K_{AB})(\nu a'_{\text{Public}})(\nu a'_{\text{EK}^{\text{Public}}}) \\ & \overline{e}\langle \mathit{pencrypt}((a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}}), y)\rangle. \\ & e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0 \end{aligned}$$

where either $T_x = \text{Secret}$ and $T_y = \text{EK}^{\text{Public}}$, or $T_x, T_y \in T_{\text{Public}}$.

- In the first case, we choose K_{AB} of type Secret, a'_{Public} of type Public, and $a'_{\text{EK}^{\text{Public}}}$ of type $\text{EK}^{\text{Public}}$. Then $(a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}})$ is of type Public-Secret-$\text{EK}^{\text{Public}}$ and $O_{\mathit{pencrypt}}(\text{Public-Secret-EK}^{\text{Public}}, \text{EK}^{\text{Public}}) = \text{Public}$.
- In the second case, we choose K_{AB}, a'_{Public}, and $a'_{\text{EK}^{\text{Public}}}$ of type Public. Then $(a'_{\text{Public}}, (x, K_{AB}), a'_{\text{EK}^{\text{Public}}})$ is of type Public and $O_{\mathit{pencrypt}}(\text{Public}, T_y) = \text{Public}$.

In both cases, it follows that the encryption is of type Public by (Constructor application), and that the output is well-typed. The input $e(z').\mathit{let}\ (x''_1, s', x''_3) = \mathit{sdecrypt}(z', K_{AB})\ \mathit{in}\ 0$ is clearly well-typed in both cases. Thus, we obtain $E \vdash P$. Finally, by Theorem 5.2.1, we conclude that P preserves the secrecy of s from $\{e\}$.

We can adapt the process P' of Section 2.2 in a similar way, with the redefinitions:

$$\begin{aligned} P' \triangleq\ & (\nu sK_A)(\nu sK_B)\mathit{let}\ pK_A = pk(sK_A)\ \mathit{in} \\ & \quad \mathit{let}\ pK_B = pk(sK_B)\ \mathit{in}\ \overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(!A' \mid !B' \mid !B'') \\ A' \triangleq\ & e(x_{pK_B}).(\nu k)(\nu a_{\text{Public}})\overline{e}\langle \mathit{pencrypt}((a_{\text{Public}}, k, pK_A), x_{pK_B})\rangle. \\ & \quad e(z).\mathit{let}\ (x'_1, (x, y), x'_3) = \mathit{pdecrypt}(z, sK_A)\ \mathit{in} \\ & \quad \mathit{if}\ x = k\ \mathit{then} \\ & \quad (\mathit{if}\ x_{pK_B} = pK_B\ \mathit{then}\ (\nu a''_{\text{Public}})(\nu a''_{\text{EK}^{\text{Public}}}) \\ & \quad\ \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s_B, a''_{\text{EK}^{\text{Public}}}), y)\rangle \\ & \quad \mid \mathit{if}\ x_{pK_B} = pK_A\ \mathit{then}\ (\nu a''_{\text{Public}})(\nu a''_{\text{EK}^{\text{Public}}}) \\ & \quad\ \overline{e}\langle \mathit{sencrypt}((a''_{\text{Public}}, s_A, a''_{\text{EK}^{\text{Public}}}), y)\rangle) \end{aligned}$$

$$
\begin{aligned}
B' \;\triangleq\;& e(z).let\ (x_1, x, y) = pdecrypt(z, sK_B)\ in \\
& (\nu K_{AB})(\nu a'_{\mathrm{Public}})(\nu a'_{\mathrm{EK}^{\mathrm{Public}}}) \\
& \overline{e}\langle pencrypt((a'_{\mathrm{Public}}, (x, K_{AB}), a'_{\mathrm{EK}^{\mathrm{Public}}}), y)\rangle. \\
& e(z').let\ (x''_1, s', x''_3) = sdecrypt(z', K_{AB})\ in\ 0 \\
B'' \;\triangleq\;& e(z).let\ (x_1, x, y) = pdecrypt(z, sK_A)\ in \\
& (\nu K_{AB})(\nu a'_{\mathrm{Public}})(\nu a'_{\mathrm{EK}^{\mathrm{Public}}}) \\
& \overline{e}\langle pencrypt((a'_{\mathrm{Public}}, (x, K_{AB}), a'_{\mathrm{EK}^{\mathrm{Public}}}), y)\rangle. \\
& e(z').let\ (x''_1, s', x''_3) = sdecrypt(z', K_{AB})\ in\ 0
\end{aligned}
$$

We can show that this variant is well-typed in this instance of the type system: $e : \mathrm{Public}, s_A : \mathrm{Secret}, s_B : \mathrm{Secret} \vdash P'$. Thus, we obtain that this process (but not the original process P' of Section 2.2) preserves the secrecy of s_A and s_B from $\{e\}$.

As in these examples, this finitary instance of the type system requires a rather strong discipline in the format of data encrypted under secret keys or sent on secret channels. While this discipline may not be hard to follow in writing new processes, it typically requires rewriting other processes before they can be typechecked. Even when the rewriting may appear simple, it may strengthen the processes in question. For example (as suggested above and explained fully in Section 7) the original process P' of Section 2.2 does not satisfy the secrecy properties that hold for its well-typed variant. What might be perceived as a disappointment if one is interested in the properties of the original process is a positive outcome if one aims to obtain security guarantees.

We return to the analysis of the original processes P and P' of Section 2.2 in Section 7.

7 The Protocol Checker

In this section we give a precise definition of a protocol checker based on untyped logic programs, then study its properties, in particular proving its equivalence to the type system. This equivalence is considerably less routine and predictable than properties such as subject reduction (Lemma 5.1.3).

As explained in the introduction, the checker takes as input a process and translates it into an abstract representation by logic-programming rules. This representation and its manipulation, but not the translation of processes, come from previous work [13]. Interested readers may consult that work for further explanations of the material in the early part of Section 7.1.

In our definition and study of the checker, we emphasize its use for proving secrecy properties, in particular that names remain secret in the sense defined in Section 3. However, the checker has also been used for establishing other security properties. In particular, it has been quite effective in proofs of authenticity properties, expressed as correspondences between events [14]. Recently, it has also been used in establishing certain process equivalences that capture strong secrecy properties [15].

7.1 Definition of the Protocol Checker

Given a closed process P_0 and a set of names S, the protocol checker builds a set of rules, in the form of Horn clauses.

The rules use two predicates: attacker and message. The fact attacker(p) means that the attacker may have p, and the fact message(p, p') means that the message p' may appear on channel p.

$F ::=$	facts
attacker(p)	attacker knowledge
message(p, p')	channel messages

Here p and p' range over patterns (or "terms", but we prefer the word "patterns" in order to avoid confusion), which are generated by the following grammar:

$p ::=$	patterns
$\quad x, y, z$	variable
$\quad a[p_1, \ldots, p_n]$	name
$\quad f(p_1, \ldots, p_n)$	constructor application

For each name a in P_0 we have a corresponding pattern construct $a[p_1, \ldots, p_n]$. We treat a as a function symbol, and write $a[p_1, \ldots, p_n]$ rather than $a(p_1, \ldots, p_n)$ only for clarity. If a is a free name, then the arity of this function is 0. If a is bound by a restriction $(\nu a)P$ in P_0, then this arity is the number of input statements above the restriction $(\nu a)P$ in the abstract syntax tree of P_0. Without loss of generality, we assume that each restriction $(\nu a)P$ in P_0 has a different name a, and that this name is different from any free name of P_0. Thus, in the checker, a new name behaves as a function of the inputs that take place (lexically) before its creation. For instance, when we represent a process of the form $(\nu b)a(x).(\nu c)Q$, we use the pattern $a[]$ for the name a, $b[]$ for b, and $c[x]$ for c. Basically, we map a and b to constants, and c to a function of the input x.

We use the same patterns even when we treat processes with more replications, such as $!(\nu b)a(x).!(\nu c)Q$. Despite the replications, we use a single pattern for b, and one that depends only on x for c. Thus, we distinguish names only when they are created after receiving different inputs. In contrast, a restriction in a process always generates fresh names; hence the rules will not exactly reflect the operational semantics of processes, but this approximation is useful for automation and harmless in most examples. As we show below, this approximation is also compatible with soundness and completeness theorems that prove the equivalence between the type system and the logic-programming system.

The rules comprise rules for the attacker and rules for the protocol. Next we define these two kinds.

7.1.1 Rules for the Attacker

Initially, the attacker has all the names in a set S, hence the rules attacker$(a[])$ for each $a \in S$. Moreover, the abilities of the attacker are represented by the following rules:

$$\begin{array}{l} \text{For each constructor } f \text{ of arity } n, \\ \quad \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n)) \end{array} \tag{Rf}$$

$$\begin{array}{l} \text{For each destructor } g, \\ \quad \text{for each equation } g(M_1, \ldots, M_n) = M \text{ in def}(g), \\ \quad \text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M) \end{array} \tag{Rg}$$

$$\text{message}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y) \tag{Rl}$$

$$\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y) \tag{Rs}$$

The rules (Rf) and (Rg) mean that the attacker can apply all operations to all terms it has, (Rf) for constructors, (Rg) for destructors. The set of these rules is finite if the set of constructors and each of the sets def(g) is finite; handling this set is easiest in this finite case. In (Rg), notice that the terms $M_1, \ldots, M_n, M$ do not contain destructors, that equations in def(g) do not have free names, and that terms without free names are also patterns, so the rules have the required format. Rule (Rl) means that the attacker can listen on all the channels it has, and (Rs) that it can send all the messages it has on all the channels it has.

7.1.2 Rules for the Protocol

When a function ρ associates a pattern with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$.

The translation $[\![P]\!]\rho h$ of a process P is a set of rules, where the environment ρ is a function that associates a pattern with each name and variable, and h is a sequence of facts of the form $\mathrm{message}(p, p')$. The empty sequence is denoted by $\emptyset$; the concatenation of a fact F to the sequence h is denoted by $h \wedge F$.

- $[\![0]\!]\rho h = \emptyset$
- $[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h$
- $[\![!P]\!]\rho h = [\![P]\!]\rho h$
- $[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[p'_1, \ldots, p'_n]])h$ if $h = \mathrm{message}(p_1, p'_1) \wedge \ldots \wedge \mathrm{message}(p_n, p'_n)$
- $[\![M(x).P]\!]\rho h = [\![P]\!](\rho[x \mapsto x])(h \wedge \mathrm{message}(\rho(M), x))$
- $[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \mathrm{message}(\rho(M), \rho(N))\}$
- $[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho h =$
  $\cup\{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma h) \mid g(p'_1, \ldots, p'_n) = p'$ is in $\mathrm{def}(g)$ and (σ, σ') is a most general pair of substitutions such that $\sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho h$

Thus, the translation of a process is, very roughly, a set of rules that enable us to prove that it sends certain messages. The sequence h keeps track of messages received by the process, since these may trigger other messages.

- The translation of 0 is the empty set, because this process does nothing.
- The translation of a parallel composition $P \mid Q$ is the union of the translations of P and Q, because $P \mid Q$ sends the messages of P and Q plus any messages that result from the interaction of P and Q.
- Replication is ignored, because the target logic is classical, so all logical rules are applicable arbitrarily many times.
- For restriction, we replace the restricted name a in question with a pattern $a[\ldots]$ that depends on the messages received, as recorded in the sequence h.
- The sequence h is extended in the translation of an input, with the input in question.
- On the other hand, the translation of an output adds a clause; this clause represents that reception of the messages in h can trigger the output in question.
- Finally, the translation of a destructor application takes the union of the clauses for the case where the destructor succeeds (with an appropriate substitution) and those for the case where the destructor fails; thus the translation avoids having to determine whether the destructor will succeed or fail.

7.1.3 Summary and Secrecy Results

Let $\rho = \{a \mapsto a[] \mid a \in \mathit{fn}(P_0)\}$. We define the rule base corresponding to the closed process P_0 as:

$$B_{P_0,S} = [\![P_0]\!]\rho\emptyset \cup \{\mathrm{attacker}(a[]) \mid a \in S\} \cup \{(\mathrm{Rf}), (\mathrm{Rg}), (\mathrm{Rl}), (\mathrm{Rs})\}$$

As an example, Figure 7 gives the rule base for the process P of the end of Section 2.1. In this rule base, all occurrences of $\mathrm{message}(c[], M)$ where $c \in S$ are replaced by $\mathrm{attacker}(M)$.

$$
\begin{aligned}
&\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(\mathit{pencrypt}(x, y))\\
&\text{attacker}(x) \Rightarrow \text{attacker}(\mathit{pk}(x))\\
&\text{attacker}(\mathit{pencrypt}(m, \mathit{pk}(k))) \wedge \text{attacker}(k) \Rightarrow \text{attacker}(m)\\
&\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(\mathit{sencrypt}(x, y))\\
&\text{attacker}(\mathit{sencrypt}(m, k)) \wedge \text{attacker}(k) \Rightarrow \text{attacker}(m)\\
&\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y)\\
&\text{message}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y)\\
&\text{attacker}(e[])\\
&\text{attacker}(\mathit{pk}(sK_A[]))\\
&\text{attacker}(\mathit{pk}(sK_B[]))\\
&\text{attacker}(\mathit{pencrypt}((k[], \mathit{pk}(sK_A[])), \mathit{pk}(sK_B[])))\\
&\text{attacker}(\mathit{pencrypt}((k[], x), \mathit{pk}(sK_A[]))) \Rightarrow \text{attacker}(\mathit{sencrypt}(s[], x))\\
&\text{attacker}(\mathit{pencrypt}((x, y), \mathit{pk}(sK_B[])))\\
&\quad \Rightarrow \text{attacker}(\mathit{pencrypt}((x, K_{AB}[\mathit{pencrypt}((x, y), \mathit{pk}(sK_B[]))]), y))
\end{aligned}
$$

Figure 7: Rules for the process P of Section 2.2

These two facts are equivalent by the rules (Rl) and (Rs). The rules for tuples are omitted; these rules are built-in in the protocol checker [13].

We have the following secrecy result. Let $s \in \mathit{fn}(P_0)$. If attacker($s[]$) cannot be derived from $B_{P_0,S}$, then P_0 preserves the secrecy of s from S. This result is the basis for a method for proving secrecy properties. Of course, whether a fact can be derived from $B_{P_0,S}$ may be undecidable, but in practice there exist algorithms that terminate on numerous examples of protocols. In particular, we can use variants of resolution algorithms, such as resolution with free selection, as in [13]. It has been shown that this algorithm always terminates for a class of protocols called tagged protocols [16]. Intuitively, a tagged protocol is a protocol in which each application of a cryptographic constructor (in particular, each encryption and each signature) is distinguished from others by a constant tag. For instance, to encrypt m under k, we write $\mathit{sencrypt}((c, m), k)$ instead of $\mathit{sencrypt}(m, k)$, where c is a constant tag. Different encryptions in the protocol use different tags, and the receiver of a message always checks the tags. Experimentally, the algorithm also terminates on many non-tagged protocols. Comon and Cortier show that an algorithm using ordered binary resolution, ordered factorization, and splitting terminates on protocols which blindly copy at most one term in each message [22]. (A blind copy happens when a participant sends back part of a message it received without looking at what is contained inside this part.)

The secrecy result discussed above can be proved directly. Instead, below we establish it by showing that we can build a typing of P_0 in a suitable instance of our general type system; the result then follows from Theorem 5.2.1. We also establish a completeness theorem, as a converse: the checker yields the "best" instance of our general type system.

7.2 Correctness

We use the rule base $B_{P_0,S}$ to define an instance of our general type system, as follows.

- The grammar of types is:

$T ::=$	types
$\quad a[T_1, \ldots, T_n]$	name
$\quad f(T_1, \ldots, T_n)$	constructor application

The types are exactly closed patterns.

- $T_{\text{Public}} = \{T \mid \text{attacker}(T) \text{ is derivable from } B_{P_0,S}\}$ (that is, the protocol checker says that the attacker may have T).
- $\mathit{conveys}(T) = \{T' \mid \text{message}(T, T') \text{ is derivable from } B_{P_0,S}\}$ (that is, the protocol checker says that the channel T may convey T').
- $O_f(T_1, \ldots, T_n) = f(T_1, \ldots, T_n)$.
- $O_g(T_1, \ldots, T_n) = \{\sigma M \mid$ there exists an equation $g(M_1, \ldots, M_n) = M$ in def(g), σ maps variables to types, and for all $i \in \{1, \ldots, n\}$, $\sigma M_i = T_i\}$.

  (Notice that this definition is compatible with the definition of O_{id} and O_{equals} in the encoding of let and conditionals of Section 4.)

We have the following two results:

Proposition 7.2.1 *The checker's type system satisfies the constraints (P0, P1, P2, P3) of the general type system.*

Lemma 7.2.2 *Let P_0 be a closed process and $E = \{a : a[] \mid a \in fn(P_0)\}$. Then $E \vdash P_0$.*

The proofs of these results are in an appendix.

The secrecy theorem for the protocol checker follows from these results and the secrecy theorem for the general type system (Theorem 5.2.1):

Theorem 7.2.3 (Secrecy) *Let P_0 be a closed process and $s \in fn(P_0)$. If* attacker$(s[])$ *cannot be derived from $B_{P_0,S}$, then P_0 preserves the secrecy of s from S.*

Proof Let $E = \{a : a[] \mid a \in fn(P_0)\}$, and $E' = \{a : a[] \mid a \in fn(P_0) \cup S\}$. By Lemma 7.2.2, $E \vdash P_0$, so $E' \vdash P_0$. Since attacker$(s[])$ cannot be derived from $B_{P_0,S}$, we have $s[] \notin T_{\text{Public}}$. Let $S' = \{b \mid E' \vdash b : T \text{ and } T \in T_{\text{Public}}\}$. By Theorem 5.2.1 (and Proposition 7.2.1), P_0 preserves the secrecy of s from S'. We have $S \subseteq S'$. (If $b \in S$, then attacker$(b[]) \in B_{P_0,S}$, so $b[] \in T_{\text{Public}}$ and $E' \vdash b : b[]$, so $b \in S'$.) Therefore, a fortiori, P_0 preserves the secrecy of s from S. □

For example, attacker$(s[])$ is not derivable from $B_{P,\{e\}}$ where P is the process of Section 2.2, so we can show using this theorem that P preserves the secrecy of s from $\{e\}$. We can also show that the process P' preserves the secrecy s_B from $\{e\}$. However, attacker$(s_A[])$ is derivable from $B_{P',\{e\}}$, so we cannot prove that P' preserves the secrecy of s_A from $\{e\}$. More precisely, we can derive attacker$(s_A[])$ as follows. The clauses

$$\text{attacker}(pk(sK_A[])) \tag{1}$$

$$\text{attacker}(x_{pK_B}) \Rightarrow \text{attacker}(\mathit{pencrypt}((k[x_{pK_B}], pk(sK_A[])), x_{pK_B})) \tag{2}$$

$$\begin{aligned}&\text{attacker}(\mathit{pencrypt}((k[pk(sK_A[])], y), pk(sK_A[]))) \wedge \text{attacker}(pk(sK_A[]))\\ &\quad \Rightarrow \text{attacker}(\mathit{sencrypt}(s_A[], y))\end{aligned} \tag{3}$$

$$\text{attacker}(\mathit{sencrypt}(x, y)) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(x) \tag{4}$$

are in $B_{P',\{e\}}$: (1) comes from the output $\overline{e}\langle pK_A\rangle$, (2) comes from the output of message 1 by A $\overline{e}\langle \mathit{pencrypt}((k, pK_A), x_{pK_B})\rangle$, (3) comes from the output of message 3

by A $\overline{e}\langle sencrypt(s_A, y)\rangle$, and (4) means that the adversary can decrypt when it has the key; it is (Rg) for the destructor *sdecrypt*. By (1), attacker$(pk(sK_A[]))$ is true; by (2), we derive attacker$(pencrypt((k[pk(sK_A[])], pk(sK_A[])), pk(sK_A[])))$; by (3), we derive attacker$(sencrypt(s_A[], pk(sK_A[])))$; and by (4), we finally obtain attacker$(s_A[])$. This derivation corresponds to an attack against the protocol:

Message 1. $A \to C(A) : pencrypt((k, pK_A), pK_A)$
Message 2. $C \to A : pencrypt((k, pK_A), pK_A)$
Message 3. $A \to C(A) : sencrypt(s, pK_A)$

First A sends message 1 to itself playing the role of B. (This corresponds to applying the clause (2).) The attacker C intercepts this message and sends it back to A as message 2. A then replies with message 3 $sencrypt(s, pK_A)$. (This corresponds to applying the clause (3).) The attacker can decrypt this reply. (This corresponds to applying the clause (4).) This attack depends on A mistaking its own public key for a session key. Such "type confusions" are not always possible in concrete implementations (for example, because public keys and session keys may have different lengths). When they are, they can be prevented by tagging data with type tags. The "type confusions" can also be prevented through discipline: for example, in Section 6.2, the constraint that encryptions must take plaintexts of type Public-Secret-EK$^{\text{Public}}$ prevents the attack on a variant of P'. In this and many similar cases, type systems can support the prudent design of protocols.

We may note that this protocol is also subject to another attack, which does not compromise the secrecy of s_A and s_B and which resembles Lowe's attack against the Needham-Schroeder public-key protocol [43]:

Message 1. $A \to C : pencrypt((k, pK_A), pK_C)$
Message 1'. $C(A) \to B : pencrypt((k, pK_A), pK_B)$
Message 2. $B \to C(A) : pencrypt((k, K_{AB}), pK_A)$
Message 2'. $C \to A : pencrypt((k, K_{AB}), pK_A)$
Message 3. $A \to C : sencrypt(s, K_{AB})$
Message 3'. $C(A) \to B : sencrypt(s, K_{AB})$

In this attack, A executes a run with the adversary C, and C uses this run to execute a run of the protocol with B as if it were A. C decrypts the first message received from A, encrypts it with B's public key, and sends it to B. B then replies with the second message, which C simply forwards to A. A replies with the last message, which C forwards to B to complete the run. A then believes that k, K_{AB}, and s are secrets shared with C, while B believes that they are secrets shared with A. C can obtain k (but not s and K_{AB}). We can exhibit this attack by adding one more message $B \to A : sencrypt(s', k)$. The fact attacker$(s'[])$ is then derivable from the clauses that represent the protocol and the adversary, as expected since the resulting protocol does not preserve the secrecy of s'. This attack can be prevented by adding the public key pK_B of B in the second message.

With this addition and the addition of tags (discussed above), we obtain the following exchange:

Message 1. $A \to B : pencrypt((c_1, k, pK_A), pK_B)$
Message 2. $B \to A : pencrypt((c_2, k, K_{AB}, pK_B), pK_A)$
Message 3. $A \to B : sencrypt(s, K_{AB})$
Message 4. $B \to A : sencrypt(s', k)$

where c_1 and c_2 are tags for messages 1 and 2, respectively. We have studied a process that represents this exchange. Using the checker, we have proved that this process preserves the secrecy of s and s', as desired. (We omit details of this analysis for the sake of brevity.)

Despite what the previous examples might suggest, a derivation of the fact attacker($s[]$) does not always correspond to an actual attack that compromises the secrecy of the corresponding name s. For instance, the process $P_0 = (\nu c)(\overline{c}\langle s\rangle \mid c(x).\overline{d}\langle c\rangle)$ preserves the secrecy of s from $\{d\}$, but the checker cannot establish it because attacker($s[]$) is derivable from the clauses (Rl), message($c[], s[]$), and message($c[], x$) $\Rightarrow$ attacker($c[]$) that are in $B_{P_0,\{d\}}$. (Note that message($d[], c[]$) is equivalent to attacker($c[]$) since d is a public channel.) These clauses do not take into account that the output $\overline{c}\langle s\rangle$ must have been executed before the adversary gets the channel c. This incompleteness is not specific to the checker. In particular, our relative completeness result (below) implies that no instance of our general type system can prove that P_0 preserves the secrecy of s from $\{d\}$. Furthermore, in practice, the checker rarely signals false attacks when applied to processes that correspond to actual protocols.

7.3 Completeness

The protocol checker is incomplete in the sense that it fails to prove some true properties. However, as the next theorem states, the protocol checker is relatively complete: it is as complete as the type system of Section 4.

Theorem 7.3.1 (Completeness) *Let P_0 be a closed process, s a name, and S a set of names. Suppose that an instance of the general type system proves (by Theorem 5.2.1) that P_0 preserves the secrecy of s from S. Then* attacker($s[]$) *cannot be derived from $B_{P_0,S}$, so the protocol checker also proves that P_0 preserves the secrecy of s from S.*

This completeness result shows the power of the protocol checker. This power is not only theoretical: it has been demonstrated in practice on several examples [13], from simple protocols like variants of the Needham-Schroeder protocols [49] to Skeme [41], a certified email protocol [4, 8], and JFK [6, 10].

The completeness result does not however mean that the protocol checker constitutes the only useful instance of the general type system. In particular, simpler instances are easier to use in manual reasoning. Presenting those instances by type rules (rather than logic programs) is often quite convenient. Moreover, the checker does not always terminate, in particular when it tries to establish properties of an infinite family of types; in other instances of the type system, we may merge those types (obtaining some finite proofs at the cost of completeness). Similarly, the (rare) case where a set def(g) is large or infinite is more problematic for the checker than for the general type system. Finally, the general type system may be combined with other type-based analyses for proving protocol properties other than secrecy (e.g., as in [32], which deals with authenticity properties).

The proof of the theorem requires establishing a correspondence between types T of an instance of the general type system and closed patterns T_{c} (which are the types of the checker according to Section 7.2): we define a partial function ϕ that maps T_{c} to T. Then we prove that all rules of $B_{P_0,S}$ are satisfied, in the following sense:

Definition 7.3.2 The closed fact attacker(T_{c}) is said to be satisfied if $\phi(T_{\mathrm{c}})$ is defined and $\phi(T_{\mathrm{c}}) \in T_{\mathrm{Public}}$. The closed fact message($T_{\mathrm{c}}, T'_{\mathrm{c}}$) is satisfied if $\phi(T'_{\mathrm{c}}) \in \mathit{conveys}(\phi(T_{\mathrm{c}}))$. The sequence of closed facts $F_1 \wedge \ldots \wedge F_n$ is satisfied if for all $i \in \{1, \ldots, n\}$, F_i is satisfied. The rule $F_1 \wedge \ldots \wedge F_n \Rightarrow F$ is satisfied if, for every closed substitution σ such that $\sigma(F_1 \wedge \ldots \wedge F_n)$ is satisfied, σF is also satisfied.

Therefore, all facts derived from $B_{P_0,S}$ are satisfied. Moreover, if s is proved secret by the instance of the general type system, then attacker($s[]$) is not satisfied. (If attacker($s[]$) were satisfied, we would also have that $\phi(s[]) \in T_{\mathrm{Public}}$, so the instance of the general type system would not be able to prove the secrecy of s.) Hence, attacker($s[]$) cannot be derived from $B_{P_0,S}$. The result follows.

The rest of this section gives a more detailed explanation of the proof. We consider a closed process P_0, a name s, and a set of names S. We also consider an instance of the general type system, and assume that this instance proves (by Theorem 5.2.1) that P_0 preserves the secrecy of s from S. That is, we assume that, in this instance, there exists an environment E_0 such that $E_0 \vdash P_0$, $E_0 \vdash s : T$ with $T \notin T_{\text{Public}}$, and $S = \{a \mid E_0 \vdash a : T \text{ and } T \in T_{\text{Public}}\}$. Without loss of generality, we may assume that E_0 contains only names. We fix a proof of $E_0 \vdash P_0$ for the rest of this argument.

Now we consider the protocol checker. All values concerning this system have index c. The set of types is:

$T_{\text{c}} ::=$	types
$a[T_{\text{c}1}, \ldots, T_{\text{c}n}]$	name
$f(T_{\text{c}1}, \ldots, T_{\text{c}n})$	constructor application

Intuitively, a well-chosen environment for a subprocess P of P_0 is an environment that can be used to type P in a "standard" proof that P_0 is well-typed, using the type system associated with the protocol checker in Section 7.2. A "standard" proof is one in which types introduced by the rule (Restriction) for $(\nu a)Q$ are of the form $a[T_{\text{c}1}, \ldots, T_{\text{c}n}]$, where $T_{\text{c}1}, \ldots, T_{\text{c}n}$ are the types of the variables bound by inputs above $(\nu a)Q$ in P_0's syntax tree.

A $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for P is similar, except that the parameters $(T_{\text{c}1}, \ldots, T_{\text{c}n})$ indicate which types should be chosen for the variables bound by inputs. Note that a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for P does not always exist, for example when the number of parameters $(T_{\text{c}1}, \ldots, T_{\text{c}n})$ does not correspond to the number of variables bound by inputs above P in P_0.

Definition 7.3.3 Let $T_{\text{c}1}, \ldots, T_{\text{c}n}$ be closed patterns. A $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for an occurrence of a subprocess of P_0 is defined as follows:

- A ()-well-chosen environment for P_0 is $\rho_0 = \{a \mapsto a[] \mid (a : T) \in E_0\}$.
- If E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for $\overline{M}\langle N\rangle.P$, then E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for P.
- If E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for $M(x).P$, then $E_{\text{c}}[x \mapsto T_{\text{c}n+1}]$ is a $(T_{\text{c}1}, \ldots, T_{\text{c}n}, T_{\text{c}n+1})$-well-chosen environment for P, for all $T_{\text{c}n+1}$.
- If E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for $P \mid Q$, then E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for P and Q.
- If E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for $!P$, then E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for P.
- If E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for $(\nu a)P$, then $E_{\text{c}}[a \mapsto a[T_{\text{c}1}, \ldots, T_{\text{c}n}]]$ is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for P.
- Finally, if E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q, then E_{c} is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for Q, and if in addition there exist an equation $g(M'_1, \ldots, M'_n) = M'$ in $\text{def}(g)$ and a substitution σ such that for all $i \in \{1, \ldots, n\}$, $\sigma M'_i = E_{\text{c}}(M_i)$, then $E_{\text{c}}[x \mapsto \sigma M']$ is a $(T_{\text{c}1}, \ldots, T_{\text{c}n})$-well-chosen environment for P. (In writing $E_{\text{c}}(M_i)$, we view E_{c} as a function on atoms and extend it to terms as a substitution.)

A pair (ρ, h) is a well-chosen pair for P if $h = \text{message}(c_1, p_1) \wedge \ldots \wedge \text{message}(c_n, p_n)$ and, for every closed substitution σ, $\sigma\rho$ is a $(\sigma p_1, \ldots, \sigma p_n)$-well-chosen environment for P.

A $(T_{c1}, \dots, T_{cn})$-well-chosen environment for P depends not only on the process P, but on its occurrence in P_0. However, notice that if $P = (\nu a)P'$ and we fix the bound name a, the occurrence of the process P is unique, since different restrictions in P_0 must create different names. We will have that, if $[\![P]\!]\rho h$ is called during the evaluation of $[\![P_0]\!]\rho_0\emptyset$ for $\rho_0 = \{a \mapsto a[] \mid (a:T) \in E_0\}$, then (ρ, h) is a well-chosen pair for P.

The function ϕ is defined so that if a type T_c appears in a standard proof that P_0 is well-typed using the type system associated with the protocol checker in Section 7.2, then $\phi(T_c)$ appears in the corresponding place in the proof of $E_0 \vdash P_0$ in the instance of the general type system under consideration.

Definition 7.3.4 The partial function $\phi : T_c \to T$ from types of the protocol checker to types of the instance of the general type system is defined by induction on the term T_c:

- $\phi(f(T_{c1}, \dots, T_{cn})) = O_f(\phi(T_{c1}), \dots, \phi(T_{cn}))$. (Therefore, $\phi(f(T_{c1}, \dots, T_{cn}))$ is undefined if $O_f(\phi(T_{c1}), \dots, \phi(T_{cn}))$ is undefined.)
- If $E_0 \vdash a : T$, then $\phi(a[]) = T$.
- When a is bound by a restriction in P_0, we define $\phi(a[T_{c1}, \dots, T_{cn}])$ as follows. Let P be the process such that $(\nu a)P$ is a subprocess of P_0. Let E_c be a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $(\nu a)P$. Let $E = \phi \circ E_c$. Then $\phi(a[T_{c1}, \dots, T_{cn}]) = T'$ where T' is such that $E, a : T' \vdash P$ is a judgment used to prove $E_0 \vdash P_0$. There is at most one such judgment, so T' is unique.

  If a $(T_{c1}, \dots, T_{cn})$-well-chosen environment for $(\nu a)P$ does not exist, or if no suitable judgment $E, a{:}T' \vdash P$ appears in the proof of $E_0 \vdash P_0$, then $\phi(a[T_{c1}, \dots, T_{cn}])$ is undefined.

This definition is recursive, and we can check that it is well-founded using the following ordering. Names are ordered by $a < b$ if a is bound above b in P_0, or a is free and b is bound in P_0. The ordering on terms is then the lexicographic ordering of pairs containing as first component the multiset of names that appear in the term and as second component the size of the term. In the first case of the definition of ϕ, the first component is constant or decreases and the second one decreases. In the third case, the first component decreases: when defining $\phi(a[T_{c1}, \dots, T_{cn}])$, in the recursive calls used to compute $\phi \circ E_c$, the name a at the top of the term has disappeared, and the only names that have appeared with the computation of the well-chosen environment are free names or names bound above a (therefore names smaller than a).

In an appendix, we establish the following three lemmas:

Lemma 7.3.5 *Let $a \in S$. The fact* attacker$(a[])$ *is satisfied.*

Lemma 7.3.6 *The rules for the attacker are satisfied.*

Lemma 7.3.7 *Let P be an occurrence of a subprocess of P_0, and (ρ, h) be a well-chosen pair for P. If, for every closed substitution σ such that σh is satisfied, $\phi \circ \sigma\rho \vdash P$ has been proved to obtain $E_0 \vdash P_0$, then the rules in $[\![P]\!]\rho h$ are satisfied. In particular, the rules in $[\![P_0]\!]\rho_0\emptyset$ are satisfied, where $\rho_0 = \{a \mapsto a[] \mid (a:T) \in E_0\}$.*

Using these lemmas, we obtain the theorem as indicated above:

Proof of Theorem 7.3.1 All the rules in $B_{P_0,S}$ are satisfied, by Lemmas 7.3.5, 7.3.6, and 7.3.7. By induction on derivations, we easily see that all facts derived from $B_{P_0,S}$ are satisfied. Moreover, $E_0 \vdash s : T$, with $T \notin T_{\text{Public}}$. By definition of ϕ, $\phi(s[]) = T \notin T_{\text{Public}}$. Therefore, attacker$(s[])$ is not satisfied, so attacker$(s[])$ cannot be derived from $B_{P_0,S}$, that is, the checker claims that P_0 preserves the secrecy of s from S. □

8 Treatment of General Equational Theories

As Section 2.1 indicates, the classification of functions into constructors and destructors has limitations; for example, XOR does not fit in either class, so it is hard to treat. A convenient way to overcome these limitations is to allow more general equational theories, as in the applied pi calculus [7]. This section briefly describes one treatment of those equational theories.

In this treatment, we assume that terms are subject to an equational theory $\mathcal{T}$, defined by a set of equations $M = N$ in which the terms M and N do not contain free names. The equational theory is the smallest congruence relation that includes this set of equations and that is preserved by substitution of terms for variables. We write $\mathcal{T} \vdash M = N$ when M equals N in the equational theory.

We can extend the semantics of our calculus to handle equational theories. For this purpose, we can either require that the definitions of destructors be invariant under the equational theory, or allow destructors that can non-deterministically yield several values. In either case, we add the structural congruence $P\{M/x\} \equiv P\{N/x\}$ when $\mathcal{T} \vdash M = N$, and make sure that an *else* branch of a destructor is selected only when no equation makes it possible to apply the destructor. Similarly, for a conditional, the *else* branch should be selected only when the corresponding terms are not equal modulo $\mathcal{T}$.

It is fairly straightforward to extend the generic type system to equational theories. It suffices to add the condition that if two terms are equal then they have the same types:

(P4) If $E \vdash M : T$ and $\mathcal{T} \vdash M = N$, then $E \vdash N : T$.

On the other hand, defining useful instances of the generic type system (in the style of Section 6.1) can sometimes be difficult. For instance, it is not clear what types should be used for XOR or for Diffie-Hellman key-agreement operations, though we have ideas on the latter.

In extending the protocol checker of Section 7, we can use essentially the same Horn clauses to represent a protocol, but these Horn clauses have to be considered modulo an equational theory, and that raises difficult issues. We have to perform unifications modulo an equational theory, or to use other techniques for reasoning on Horn clauses modulo equations, such as paramodulation [12]. (Correspondingly, in our proofs, the types that correspond to the checker would be quotients of closed patterns by an equational theory.)

For simplicity, the current implementation of the checker includes only a simple treatment of equations. To each constructor f is attached a finite set of equations $f(M_1, \dots, M_n) = M$ which is required to satisfy certain closure conditions. It is then easy to generate appropriate Horn clauses for representing a protocol. Obviously, this approach limits which equational theories can be handled. For instance, this approach permits the equation $f(x, g(y)) = f(y, g(x))$, which can be used to model Diffie-Hellman operations [7], but unification modulo an equational theory could yield a more detailed model [36, 45].

9 Conclusion

This paper makes two main contributions:

1. a type system for expressing and proving secrecy properties of security protocols with a generic treatment of many cryptographic operations;

2. a tight relation between two useful but superficially quite different approaches to protocol analysis, respectively embodied in the type system and in a logic-programming tool.

The first contribution can be seen as the continuation of a line of work on static analyses for security, discussed in the introduction. So far, those static analyses have been developed successfully but often in ad hoc ways. We believe that type systems such as ours not only are useful in examples but also shed light on the constraints and the design space for static analyses.

In the last few years, there has been a vigorous proliferation of frameworks and techniques for reasoning about security protocols. Their relations are seldom explicit or obvious. Moreover, little is known about how to combine techniques. The second contribution is part of a broader effort to understand those relations. It focuses on techniques based on types and on logic programs because of their effectiveness and their popularity, illustrated by the many references given in the introduction. Previous work (in particular [30]) suggests connections between (untyped) process calculi and logic-programming notations for protocols; we go further by relating proof methods in those two worlds. Such connections are perhaps the start of a healthy consolidation.

Acknowledgments

This work was partly done while Martín Abadi was at Bell Labs Research, Lucent Technologies, and at InterTrust's Strategic Technologies and Architectural Research Laboratory, and while Bruno Blanchet was at INRIA Rocquencourt and at Max-Planck-Institut für Informatik. Martín Abadi's research was partly supported by faculty research funds granted by the University of California, Santa Cruz, and by the National Science Foundation under Grants CCR-0204162 and CCR-0208800. We would like to thank the anonymous referees, the Area Editor, and Xavier Allamigeon for their helpful comments on this paper.

Appendix: Additional Proofs

This appendix contains a few proofs omitted in the main body of the paper.

Proofs of Proposition 7.2.1 and Lemma 7.2.2

If a finite function E maps atoms to types, we write E also for the environment that binds each atom u in $dom(E)$ with $u : E(u)$. The bindings can be in any order. In addition, the function E is extended to all terms as a substitution.

Lemma .0.8 *In the type system of Section 7.2, if E binds all names and variables in M to types (that is, closed patterns), then*

$$E \vdash M : E(M)$$

Proof The proof is by induction on the term M.

- For an atom u, we have $E \vdash u : E(u)$ by (Atom), hence the result.
- For a composite term $f(M_1, \ldots, M_n)$, we have $E \vdash M_i : E(M_i)$ by induction hypothesis. Therefore, by (Constructor application), we obtain $E \vdash f(M_1, \ldots, M_n){:}E(f(M_1, \ldots, M_n))$ since $O_f(E(M_1), \ldots, E(M_n)) = f(E(M_1), \ldots, E(M_n)) = E(f(M_1, \ldots, M_n))$.

□

Proof of Proposition 7.2.1 The proof relies on the rules that represent the attacker in the checker.

(P0) The rule $\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y)$ is in $B_{P_0,S}$. If $T \in T_{\text{Public}}$ and $T' \in T_{\text{Public}}$, then $\text{attacker}(T)$ and $\text{attacker}(T')$ can be derived from $B_{P_0,S}$. So $\text{message}(T, T')$ can also be derived from $B_{P_0,S}$ and $T' \in \mathit{conveys}(T)$. Therefore, $T \in T_{\text{Public}}$ implies $T_{\text{Public}} \subseteq \mathit{conveys}(T)$.

Conversely, the rule $\text{attacker}(x) \wedge \text{message}(x, y) \Rightarrow \text{attacker}(y)$ is also in $B_{P_0,S}$. If $T \in T_{\text{Public}}$ and $T' \in \mathit{conveys}(T)$ then $T' \in T_{\text{Public}}$. Therefore, $T \in T_{\text{Public}}$ implies $T_{\text{Public}} \supseteq \mathit{conveys}(T)$.

(P1) The rule $\mathrm{attacker}(x_1) \wedge \ldots \wedge \mathrm{attacker}(x_n) \Rightarrow \mathrm{attacker}(f(x_1, \ldots, x_n))$ is in $B_{P_0,S}$. Therefore, if $T_1 \in T_{\mathrm{Public}}, \ldots, T_n \in T_{\mathrm{Public}}$, then $O_f(T_1, \ldots, T_n) \in T_{\mathrm{Public}}$.

(P2) Assume that $T \in O_g(T_1, \ldots, T_n)$. Then there exists an equation $g(M_1, \ldots, M_n) = M$ in $\mathrm{def}(g)$ and a substitution σ such that $T_i = \sigma M_i$ for all i and $T = \sigma M$. The rule $\mathrm{attacker}(M_1) \wedge \ldots \wedge \mathrm{attacker}(M_n) \Rightarrow \mathrm{attacker}(M)$ is in $B_{P_0,S}$. If $\mathrm{attacker}(T_1) \wedge \ldots \wedge \mathrm{attacker}(T_n)$ can be derived from $B_{P_0,S}$, then $\mathrm{attacker}(T)$ can also be derived from $B_{P_0,S}$; therefore, if $T_i \in T_{\mathrm{Public}}$ for all $i \in \{1, \ldots, n\}$, then $T \in T_{\mathrm{Public}}$.

(P3) If $g(M_1, \ldots, M_n) = M$ is in $\mathrm{def}(g)$, and $E \vdash M_i : T_i$ for all i, then $T_i = E(M_i)$ by Lemma .0.8 and the uniqueness of the type of a term. So, taking $T = E(M)$, we have $T \in O_g(T_1, \ldots, T_n)$, by definition of O_g, and $E \vdash M : T$, by Lemma .0.8.

□

Proof of Lemma 7.2.2 We prove by induction on the process P that, if

1. ρ binds all free names and variables of P to patterns,
2. $B_{P_0,S} \supseteq [\![P]\!]\rho h$,
3. σ is a closed substitution, mapping all variables of h and of the image of ρ to patterns,
4. for all p and p', if $\mathrm{message}(p, p') \in h$ then $\sigma p' \in \mathit{conveys}(\sigma p)$,

then $\sigma\rho \vdash P$.

- Case 0: $\sigma\rho \vdash 0$ is always true (since $\sigma\rho$ is well-formed).
- Case $P \mid Q$: Assume that $[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h \subseteq B_{P_0,S}$. Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho \vdash P$ and $\sigma\rho \vdash Q$, so $\sigma\rho \vdash P \mid Q$ by (Parallel composition).
- Case $!P$: Assume that $[\![!P]\!]\rho h = [\![P]\!]\rho h \subseteq B_{P_0,S}$. Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho \vdash P$, so $\sigma\rho \vdash !P$ by (Replication).
- Case $(\nu a)P$: Let $h = \mathrm{message}(c_1, p_1) \wedge \ldots \wedge \mathrm{message}(c_n, p_n)$. Assume that

  $$[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[p_1, \ldots, p_n]])h \subseteq B_{P_0,S}$$

  Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho, a : \sigma(a[p_1, \ldots, p_n]) \vdash P$. Therefore, $\sigma\rho \vdash (\nu a)P$ by (Restriction).
- Case $M(x).P$: Assume that

  $$[\![M(x).P]\!]\rho h = [\![P]\!](\rho[x \mapsto x])(h \wedge \mathrm{message}(\rho(M), x)) \subseteq B_{P_0,S}$$

  Assume that σ satisfies (3) and (4). By Lemma .0.8, $\sigma\rho \vdash M : \sigma\rho(M)$. Let $h' = h \wedge \mathrm{message}(\rho(M), x)$. Let $T \in \mathit{conveys}(\sigma\rho(M))$. Let $\sigma' = \sigma[x \mapsto T]$. Then $\sigma' x \in \mathit{conveys}(\sigma'\rho(M))$, then $\mathrm{message}(p, p') \in h'$ implies $\sigma' p' \in \mathit{conveys}(\sigma' p)$. By induction hypothesis, $\sigma'\rho, x : \sigma' x \vdash P$. So for all $T \in \mathit{conveys}(\sigma\rho(M))$, $\sigma\rho, x : T \vdash P$. By (Input), $\sigma\rho \vdash M(x).P$.
- Case $\overline{M}\langle N\rangle.P$: Assume that

  $$[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \mathrm{message}(\rho(M), \rho(N))\} \subseteq B_{P_0,S}$$

  Assume that σ satisfies (3) and (4). By induction hypothesis, $\sigma\rho \vdash P$. By Lemma .0.8, $\sigma\rho \vdash M : \sigma\rho(M)$ and $\sigma\rho \vdash N : \sigma\rho(N)$. The rule $R = h \Rightarrow \mathrm{message}(\rho(M), \rho(N))$ is in $B_{P_0,S}$. By condition (4), for each $\mathrm{message}(p, p')$ in h, $\sigma p' \in \mathit{conveys}(\sigma p)$, so $\mathrm{message}(\sigma p, \sigma p')$ is derivable from $B_{P_0,S}$. Using the rule R, the fact $\mathrm{message}(\sigma\rho(M), \sigma\rho(N))$ is also derivable from $B_{P_0,S}$. Therefore, we have $\sigma\rho(N) \in \mathit{conveys}(\sigma\rho(M))$. By (Output), $\sigma\rho \vdash \overline{M}\langle N\rangle.P$.

- Case *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q: Assume that

$$\begin{aligned} &[\![let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q]\!]\rho h = \\ &\quad \cup \{[\![P]\!]((\sigma_1\rho)[x \mapsto \sigma'_1 p'])(\sigma_1 h) \\ &\qquad \mid g(p'_1, \ldots, p'_n) = p' \text{ is in } \mathrm{def}(g)\} \cup [\![Q]\!]\rho h \\ &\quad \subseteq B_{P_0,S} \end{aligned}$$

  where (σ_1, σ'_1) is a most general pair of substitutions such that $\sigma_1\rho(M_1) = \sigma'_1 p'_1, \ldots, \sigma_1\rho(M_n) = \sigma'_1 p'_n$. Assume that σ satisfies (3) and (4). By Lemma .0.8, $\sigma\rho \vdash M_i : \sigma\rho(M_i)$ for all $i \in \{1, \ldots, n\}$.

  If $T \in O_g(\sigma\rho(M_1), \ldots, \sigma\rho(M_n))$, then there exist an equation $g(p'_1, \ldots, p'_n) = p'$ in $\mathrm{def}(g)$ and a substitution σ' such that, for all i, $\sigma\rho(M_i) = \sigma' p'_i$ and $T = \sigma' p'$. Then there exists σ'' such that $\sigma = \sigma''\sigma_1$ and $\sigma' = \sigma''\sigma'_1$. Moreover, we have

$$[\![P]\!](\sigma_1\rho[x \mapsto \sigma'_1 p'])(\sigma_1 h) \subseteq B_{P_0,S}$$

  For all $\mathrm{message}(p_1, p_2) \in \sigma''\sigma_1 h = \sigma h$, we have $p_2 \in \mathit{conveys}(p_1)$. By induction hypothesis on P, we have $\sigma''\sigma_1\rho, x : \sigma''\sigma'_1 p' \vdash P$, that is, $\sigma\rho, x : \sigma' p' \vdash P$.

  Therefore, if $T \in O_g(\sigma\rho(M_1), \ldots, \sigma\rho(M_n))$, then $\sigma\rho, x : T \vdash P$. Finally, by induction hypothesis on Q, $\sigma\rho \vdash Q$. By (Destructor application), $\sigma\rho \vdash$ *let* $x = g(M_1, \ldots, M_n)$ *in* P *else* Q.

In particular, $B_{P_0,S} \supseteq [\![P_0]\!]\rho\emptyset$, where $\rho = \{a \mapsto a[] \mid a \in \mathit{fn}(P_0)\}$. Then, with $E = \sigma\rho = \{a : a[] \mid a \in \mathit{fn}(P_0)\}$, we obtain $E \vdash P_0$. □

Proof of Lemmas 7.3.5, 7.3.6, and 7.3.7

Proof of Lemma 7.3.5 Since $a \in S$, $(a : T) \in E_0$, with $T \in T_{\text{Public}}$. By definition of ϕ, $\phi(a[]) = T \in T_{\text{Public}}$. Therefore, attacker($a[]$) is satisfied. □

Lemma .0.9 *Let E_{c} be a partial function from atoms to closed patterns, defined for all names and variables of M. The function E_{c} is extended to a substitution.*

1. *If $\phi \circ E_{\mathrm{c}} \vdash M : T$ then $T = \phi(E_{\mathrm{c}}(M))$ (in particular, $\phi(E_{\mathrm{c}}(M))$ is defined).*
2. *If $\phi(E_{\mathrm{c}}(M))$ is defined, then $\phi \circ E_{\mathrm{c}} \vdash M : \phi(E_{\mathrm{c}}(M))$.*
   (If $\phi(E_{\mathrm{c}}(M))$ is defined, then ϕ is defined on $E_{\mathrm{c}}(u)$ for all $u \in \mathit{fn}(M) \cup \mathit{fv}(M)$.)

Proof The proof of (1) is by induction on M.

- Case M is an atom u. Since $\phi \circ E_{\mathrm{c}} \vdash u{:}T$ must have been derived by (Atom), $T = \phi(E_{\mathrm{c}}(u))$.
- Case M is a composite term $f(M_1, \ldots, M_n)$. Since $\phi \circ E_{\mathrm{c}} \vdash M : T$ can be obtained only by (Constructor), for each $i \in \{1, \ldots, n\}$, $\phi \circ E_{\mathrm{c}} \vdash M_i : T_i$ and $T = O_f(T_1, \ldots, T_n)$. Therefore, by induction hypothesis, $T_i = \phi(E_{\mathrm{c}}(M_i))$ and, by definition of ϕ, $T = O_f(\phi(E_{\mathrm{c}}(M_1)), \ldots, \phi(E_{\mathrm{c}}(M_n))) = \phi(f(E_{\mathrm{c}}(M_1), \ldots, E_{\mathrm{c}}(M_n))) = \phi(E_{\mathrm{c}}(M))$.

The proof of (2) is also by induction on M.

- Case M is an atom u. By (Atom), $\phi \circ E_{\mathrm{c}} \vdash u : \phi(E_{\mathrm{c}}(u))$.
- Case M is a composite term $f(M_1, \ldots, M_n)$. Since $\phi(E_{\mathrm{c}}(M)) = \phi(f(E_{\mathrm{c}}(M_1), \ldots, E_{\mathrm{c}}(M_n))) = O_f(\phi(E_{\mathrm{c}}(M_1)), \ldots, \phi(E_{\mathrm{c}}(M_n)))$ is defined, $\forall i \in \{1, \ldots, n\}$, $\phi(E_{\mathrm{c}}(M_i))$ is defined. By induction hypothesis, we have $\phi \circ E_{\mathrm{c}} \vdash M_i : \phi(E_{\mathrm{c}}(M_i))$. Moreover, $O_f(\phi(E_{\mathrm{c}}(M_1)), \ldots, \phi(E_{\mathrm{c}}(M_n)))$ is defined, therefore, by (Constructor), $\phi \circ E_{\mathrm{c}} \vdash M : \phi(E_{\mathrm{c}}(M))$.

□

Proof of Lemma 7.3.6 Let us prove first that $\mathrm{attacker}(x) \wedge \mathrm{message}(x,y) \Rightarrow \mathrm{attacker}(y)$ is satisfied. Let σ be any closed substitution. If $\mathrm{attacker}(\sigma x)$ and $\mathrm{message}(\sigma x, \sigma y)$ are satisfied, then $\phi(\sigma x) \in T_{\mathrm{Public}}$, so by (P0), $\mathit{conveys}(\phi(\sigma x)) = T_{\mathrm{Public}}$ and $\phi(\sigma y) \in \mathit{conveys}(\phi(\sigma x)) = T_{\mathrm{Public}}$. Then $\mathrm{attacker}(\sigma y)$ is satisfied. Therefore, the rule $\mathrm{attacker}(x) \wedge \mathrm{message}(x,y) \Rightarrow \mathrm{attacker}(y)$ is satisfied.

Similarly, $\mathrm{attacker}(x) \wedge \mathrm{attacker}(y) \Rightarrow \mathrm{message}(x,y)$ is satisfied.

Let f be a constructor. Let us prove that $\mathrm{attacker}(x_1) \wedge \ldots \wedge \mathrm{attacker}(x_n) \Rightarrow \mathrm{attacker}(f(x_1,\ldots,x_n))$ is satisfied. Let σ be any closed substitution. Assume that $\mathrm{attacker}(\sigma x_1), \ldots, \mathrm{attacker}(\sigma x_n)$ are satisfied. Then for all $i \in \{1,\ldots,n\}$, $\phi(\sigma x_i) \in T_{\mathrm{Public}}$, therefore $\phi(f(\sigma x_1,\ldots,\sigma x_n)) = O_f(\phi(\sigma x_1),\ldots,\phi(\sigma x_n)) \in T_{\mathrm{Public}}$ by (P1). Then $\mathrm{attacker}(f(\sigma x_1,\ldots,\sigma x_n))$ is satisfied. Therefore, the rule $\mathrm{attacker}(x_1) \wedge \ldots \wedge \mathrm{attacker}(x_n) \Rightarrow \mathrm{attacker}(f(x_1,\ldots,x_n))$ is satisfied.

Assume that there is an equation $g(M_1,\ldots,M_n) = M$ in $\mathrm{def}(g)$, and let us prove that $\mathrm{attacker}(M_1) \wedge \ldots \wedge \mathrm{attacker}(M_n) \Rightarrow \mathrm{attacker}(M)$ is satisfied. For every closed substitution σ, if $\mathrm{attacker}(\sigma M_1), \ldots, \mathrm{attacker}(\sigma M_n)$ are satisfied, then for all $i \in \{1,\ldots,n\}$, $\phi(\sigma M_i) \in T_{\mathrm{Public}}$, so $O_g(\phi(\sigma M_1),\ldots,\phi(\sigma M_n)) \subseteq T_{\mathrm{Public}}$ by (P2). Moreover, for all $i \in \{1,\ldots,n\}$, $\phi \circ \sigma \vdash M_i : \phi(\sigma M_i)$ by Lemma .0.9(2), therefore $\phi \circ \sigma \vdash M : T$ and $T \in O_g(\phi(\sigma M_1),\ldots,\phi(\sigma M_n))$ for some T by (P3). By Lemma .0.9(1), $T = \phi(\sigma M)$, so $\phi(\sigma M) \in O_g(\phi(\sigma M_1),\ldots,\phi(\sigma M_n))$. Hence $\phi(\sigma M) \in T_{\mathrm{Public}}$, so $\mathrm{attacker}(\sigma M)$ is satisfied. Therefore, the rule $\mathrm{attacker}(M_1) \wedge \ldots \wedge \mathrm{attacker}(M_n) \Rightarrow \mathrm{attacker}(M)$ is satisfied. □

Proof of Lemma 7.3.7 By induction on P.

- Case 0: $[\![0]\!]\rho h = \emptyset$, so the result is obvious.

- Case $P \mid Q$: Let (ρ, h) be a well-chosen pair for $P \mid Q$. Let σ be such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash P \mid Q$ has been proved to obtain $E_0 \vdash P_0$, this must have been derived by (Parallel composition), therefore $E \vdash P$ and $E \vdash Q$ have been proved to obtain $E_0 \vdash P_0$. Since this is true for any σ such that σh is satisfied, and (ρ, h) is also a well-chosen pair for P and Q, by induction hypothesis, the rules in $[\![P]\!]\rho h$ and in $[\![Q]\!]\rho h$ are satisfied. Therefore, the rules in $[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h$ are satisfied.

- Case $!P$: Let (ρ, h) be a well-chosen pair for $!P$. Let σ such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash !P$ has been proved to obtain $E_0 \vdash P_0$, this must have been derived by (Replication), then $E \vdash P$ has been proved to obtain $E_0 \vdash P_0$. Since this is true for any σ such that σh is satisfied, and (ρ, h) is also a well-chosen pair for P, by induction hypothesis, the rules in $[\![P]\!]\rho h$ are satisfied. Therefore, the rules in $[\![!P]\!]\rho h = [\![P]\!]\rho h$ are satisfied.

- Case $(\nu a)P$: Let (ρ, h) be a well-chosen pair for $(\nu a)P$. Let σ be such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash (\nu a)P$ has been proved to obtain $E_0 \vdash P_0$, this must have been derived by (Restriction), then there exists T such that $E, a : T \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By definition of ϕ, $T = \phi(a[\sigma p_1, \ldots, \sigma p_n])$, where $h = \mathrm{message}(c_1, p_1) \wedge \ldots \wedge \mathrm{message}(c_n, p_n)$, since $\sigma\rho$ is a $(\sigma p_1, \ldots, \sigma p_n)$-well-chosen environment for $(\nu a)P$. We have that $(\rho[a \mapsto a[p_1, \ldots, p_n]], h)$ is a well-chosen pair for P, and for any σ such that σh is satisfied, $\phi \circ \sigma\rho, a : \phi(a[\sigma p_1, \ldots, \sigma p_n]) \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By induction hypothesis, the rules in $[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[p_1, \ldots, p_n]])h$ are satisfied.

- Case $M(x).P$: Let (ρ, h) be a well-chosen pair for $M(x).P$. We assume that for all σ such that σh is satisfied, and $E = \phi \circ \sigma\rho$, $E \vdash M(x).P$ has been proved to obtain $E_0 \vdash P_0$. Then this must have been derived by (Input), therefore $E \vdash M : T$ and $\forall T' \in \mathit{conveys}(T), E, x : T' \vdash P$. By Lemma .0.9(1), $T = \phi(\sigma\rho(M))$.

Let $h' = h \wedge \text{message}(\rho(M), x)$. If σ' is such that $\sigma' h'$ is satisfied, then $\text{message}(\sigma'\rho(M), \sigma' x)$ is satisfied, then $\phi(\sigma' x) \in \mathit{conveys}(\phi(\sigma'\rho(M))) = \mathit{conveys}(T)$. Moreover, $\sigma' h$ is satisfied, so we can apply the reasoning above to σ' instead of σ, therefore $E, x{:}\phi(\sigma' x) \vdash P$ for $E = \phi \circ \sigma'\rho$. Let $\rho' = \rho[x \mapsto x]$. Then (ρ', h') is a well-chosen pair for P, and $\phi \circ \sigma'\rho' \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By induction hypothesis, the rules in $[\![P]\!]\rho' h'$ are satisfied. Therefore, the rules in $[\![M(x).P]\!]\rho h$ are satisfied.

- Case $\overline{M}\langle N\rangle.P$: Let (ρ, h) be a well-chosen pair for $\overline{M}\langle N\rangle.P$. Let σ be such that σh is satisfied, and $E = \phi \circ \sigma\rho$. If $E \vdash \overline{M}\langle N\rangle.P$ has been proved to obtain $E_0 \vdash P_0$, then this must have been derived by (Output), therefore $E \vdash M : T$, $E \vdash N : T'$, $T' \in \mathit{conveys}(T)$, and $E \vdash P$. By Lemma .0.9(1), $T = \phi(\sigma\rho(M))$ and $T' = \phi(\sigma\rho(N))$, therefore $\phi(\sigma\rho(N)) \in \mathit{conveys}(\phi(\sigma\rho(M)))$.

  Let $R = h \Rightarrow \text{message}(\rho(M), \rho(N))$, and let σ' be any closed substitution. If $\sigma' h$ is satisfied, the argument of the paragraph above can be applied to σ'. Then $\phi(\sigma'\rho(N)) \in \mathit{conveys}(\phi(\sigma'\rho(M)))$, so $\text{message}(\sigma'\rho(M), \sigma'\rho(N))$ is satisfied. Therefore, R is satisfied.

  We have that (ρ, h) is a well-chosen pair for P, and for all σ such that σh is satisfied, $E \vdash P$ has been proved to obtain $E_0 \vdash P_0$. By induction hypothesis on P, the rules in $[\![P]\!]\rho h$ are satisfied.

  Hence the rules in $[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{R\}$ are satisfied.

- Case *let* $x = g(M_1, \dots, M_n)$ *in* P *else* Q: Let (ρ, h) be a well-chosen pair for *let* $x = g(M_1, \dots, M_n)$ *in* P *else* Q. We assume that for all σ such that σh is satisfied, and $E = \phi \circ \sigma\rho$, $E \vdash$ *let* $x = g(M_1, \dots, M_n)$ *in* P *else* Q has been proved to obtain $E_0 \vdash P_0$. This must have been derived by (Destructor application), then $\forall i \in \{1, \dots, n\}$, $E \vdash M_i{:}T_i$, $\forall T \in O_g(T_1, \dots, T_n), E, x : T \vdash P$, and $E \vdash Q$. By Lemma .0.9(1), $T_i = \phi(\sigma\rho(M_i))$.

  Assume that there is an equation $g(p'_1, \dots, p'_n) = p'$ in $\text{def}(g)$. Let $\rho' = \sigma_1\rho[x \mapsto \sigma'_1 p']$ and $h' = \sigma_1 h$ where (σ_1, σ'_1) is the most general pair of substitutions such that $\sigma_1\rho(M_1) = \sigma'_1 p'_1$, $\dots, \sigma_1\rho(M_n) = \sigma'_1 p'_n$. Let σ'' be such that $\sigma'' h'$ is satisfied. Then $\sigma = \sigma''\sigma_1$ is such that σh is satisfied, so the argument of the paragraph above can be applied to σ. Moreover $\sigma\rho(M_i) = \sigma''\sigma'_1 p'_i$. We have $\phi \circ \sigma''\sigma'_1 \vdash p'_i : \phi(\sigma''\sigma'_1 p'_i)$ (by Lemma .0.9(2)). Therefore, by (P3), $\phi \circ \sigma''\sigma'_1 \vdash p' : \phi(\sigma''\sigma'_1 p')$ with $\phi(\sigma''\sigma'_1 p') \in O_g(\phi(\sigma''\sigma'_1 p'_1), \dots, \phi(\sigma''\sigma'_1 p'_n))$. That is, $\phi(\sigma''\sigma'_1 p') \in O_g(T_1, \dots, T_n)$. Therefore $E, x : \phi(\sigma''\sigma'_1 p') \vdash P$. That is, $\phi \circ \sigma''\rho' \vdash P$. This is true for any σ'' such that $\sigma'' h'$ is satisfied, and (ρ', h') is a well-chosen pair for P, therefore by induction hypothesis, the rules in $[\![P]\!]\rho' h'$ are satisfied.

  Moreover, (ρ, h) is also a well-chosen pair for Q, then by induction hypothesis, the rules in $[\![Q]\!]\rho h$ are satisfied. Therefore, the rules in $[\![$*let* $x = g(M_1, \dots, M_n)$ *in* P *else* $Q]\!]\rho h$ are satisfied.

In particular, for $[\![P_0]\!]\rho_0\emptyset$, $(\rho_0, \emptyset)$ is a well-chosen pair for P_0, and $E_0 = \{a : \phi(a[]) \mid (a : T) \in E_0\} = \phi \circ \sigma\rho_0$, for any σ. Therefore, $\phi \circ \sigma\rho_0 \vdash P_0$ has been proved to obtain $E_0 \vdash P_0$. Then the rules in $[\![P_0]\!]\rho_0\emptyset$ are satisfied. □

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, Sept. 1999.

[2] M. Abadi. Security protocols and their properties. In F. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, Amsterdam, The Netherlands, 2000. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

[3] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, New-York, NY, Jan. 1999. ACM Press.

[4] M. Abadi and B. Blanchet. Computer-assisted verification of a protocol for certified email. In R. Cousot, editor, *Static Analysis, 10th International Symposium (SAS'03)*, volume 2694 of *Lecture Notes in Computer Science*, pages 316–335, Berlin, Germany, June 2003. Springer-Verlag.

[5] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, Apr. 2003.

[6] M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the pi calculus. In D. Schmidt, editor, *Programming Languages and Systems: 13th European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lecture Notes in Computer Science*, pages 340–354, Berlin, Germany, Mar. 2004. Springer-Verlag.

[7] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, New-York, NY, Jan. 2001. ACM Press.

[8] M. Abadi, N. Glew, B. Horne, and B. Pinkas. Certified email with a light on-line trusted third party: Design and implementation. In *11th International World Wide Web Conference*, pages 387–395, New-York, NY, May 2002. ACM Press.

[9] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, Jan. 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[10] W. Aiello, S. Bellovin, M. Blaze, R. Canetti, J. Ionnidis, A. Keromytis, and O. Reingold. Efficient, DoS-resistant, secure key exchange for internet protocols. In R. Sandhu, editor, *ACM Conference on Computer and Communications Security (CCS'02)*, pages 48–58, New-York, NY, Nov. 2002. ACM.

[11] R. M. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In C. Palamidessi, editor, *CONCUR 2000: Concurrency Theory (11th International Conference)*, volume 1877 of *Lecture Notes in Computer Science*, pages 380–394, Berlin, Germany, Aug. 2000. Springer-Verlag.

[12] L. Bachmair and H. Ganzinger. Equational reasoning in saturation-based theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I, chapter 11, pages 353–397. Kluwer, Dordrecht, The Netherlands, 1998.

[13] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Los Alamitos, CA, June 2001. IEEE Computer Society.

[14] B. Blanchet. From secrecy to authenticity in security protocols. In M. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359, Berlin, Germany, Sept. 2002. Springer-Verlag.

[15] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Los Alamitos, CA, May 2004. IEEE Computer Society.

[16] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. In A. Gordon, editor, *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *Lecture Notes in Computer Science*, pages 136–152, Berlin, Germany, Apr. 2003. Springer-Verlag.

[17] C. Bodei. *Security Issues in Process Calculi.* PhD thesis, Università di Pisa, Jan. 2000.

[18] C. Bodei, P. Degano, F. Nielson, and H. Nielson. Control flow analysis for the π-calculus. In *CONCUR'98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98, Berlin, Germany, Sept. 1998. Springer Verlag.

[19] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton,FL, 1997.

[20] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In C. Palamidessi, editor, *CONCUR 2000: Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379, Berlin, Germany, Aug. 2000. Springer-Verlag.

[21] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW'99)*, pages 55–69, Los Alamitos, CA, June 1999. IEEE Computer Society.

[22] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In R. Nieuwenhuis, editor, *14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 148–164, Berlin, Germany, June 2003. Springer-Verlag.

[23] K. J. Compton and S. Dexter. Proof techniques for cryptographic protocols. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 25–39, Berlin, Germany, July 1999. Springer-Verlag.

[24] M. Dam. Proving trust in systems of second-order processes. In *Proceedings of the 31th Hawaii International Conference on System Sciences*, volume VII, pages 255–264, 1998.

[25] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. A new algorithm for the automatic verification of authentication protocols: From specifications to flaws and attack scenarios. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, New Jersey, Sept. 1997.

[26] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols*, 25 June 1998.

[27] D. E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, Mass., 1982.

[28] A. Durante, R. Focardi, and R. Gorrieri. CVS: A compiler for the analysis of cryptographic protocols. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW'99)*, pages 203–212, Los Alamitos, CA, June 1999. IEEE Computer Society.

[29] N. Durgin, J. Mitchell, and D. Pavlovic. A compositional logic for protocol correctness. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 241–255, Los Alamitos, CA, June 2001. IEEE Computer Society.

[30] N. A. Durgin and J. C. Mitchell. Analysis of security protocols. In M. Broy and R. Steinbruggen, editors, *Calculational System Design*, pages 369–395, Amsterdam, The Netherlands, 1999. IOS Press.

[31] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, Sept. 1997.

[32] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, Los Alamitos, CA, June 2001. IEEE Computer Society.

[33] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop (CSFW-15)*, pages 77–91, Los Alamitos, CA, June 2002. IEEE Computer Society.

[34] J. Goubault-Larrecq. Protocoles cryptographiques: la logique à la rescousse! In *Atelier SEcurité des Communications sur Internet (SECI'02)*, Sept. 2002.

[35] J. Goubault-Larrecq. Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve ? In *Actes 15èmes journées francophones sur les langages applicatifs (JFLA'04)*, Rocquencourt, France, Jan. 2004. INRIA.

[36] J. Goubault-Larrecq, M. Roger, and K. N. Verma. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 2004. To appear.

[37] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, New-York, NY, 1998. ACM Press.

[38] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 415–427, Berlin, Germany, 2000. Springer-Verlag.

[39] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In G. Smolka, editor, *Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming (ESOP 2000), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199, Berlin, Germany, 2000. Springer-Verlag.

[40] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.

[41] H. Krawczyk. SKEME: A versatile secure key exchange mechanism for internet. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

[42] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 112–121, New-York, NY, 1998. ACM Press.

[43] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166, Berlin, Germany, 1996. Springer Verlag.

[44] C. Meadows. Panel on languages for formal specification of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, page 96, Los Alamitos, CA, 1997. IEEE Computer Society.

[45] C. Meadows and P. Narendran. A unification algorithm for the group Diffie-Hellman protocol. In *Workshop on Issues in the Theory of Security (WITS'02)*, Jan. 2002.

[46] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Cambridge, United Kingdom, June 1999.

[47] J. H. Morris. Protection in programming languages. *Commun. ACM*, 16(1):15–21, Jan. 1973.

[48] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, New-York, NY, Jan. 1999. ACM Press.

[49] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

[50] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[51] P. Selinger. Models for an adversary-centric protocol logic. In J. Goubault-Larrecq, editor, *Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification*, volume 55(1) of *Electronic Notes in Theoretical Computer Science*, pages 73–88, Amsterdam, The Netherlands, July 2001. Elsevier.

[52] E. Sumii and B. C. Pierce. Logical relations and encryption (Extended abstract). In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 256–269, Los Alamitos, CA, June 2001. IEEE Computer Society.

[53] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.

[54] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–328, Berlin, Germany, July 1999. Springer-Verlag.

Automatic Verification of Correspondences for Security Protocols*

Bruno Blanchet
CNRS, École Normale Supérieure, INRIA†
Bruno.Blanchet@ens.fr

Abstract

We present a new technique for verifying correspondences in security protocols. In particular, correspondences can be used to formalize authentication. Our technique is fully automatic, it can handle an unbounded number of sessions of the protocol, and it is efficient in practice. It significantly extends a previous technique for the verification of secrecy. The protocol is represented in an extension of the pi calculus with fairly arbitrary cryptographic primitives. This protocol representation includes the specification of the correspondence to be verified, but no other annotation. This representation is then translated into an abstract representation by Horn clauses, which is used to prove the desired correspondence. Our technique has been proved correct and implemented. We have tested it on various protocols from the literature. The experimental results show that these protocols can be verified by our technique in less than 1 s.

1 Introduction

The verification of security protocols has already been the subject of numerous research works. It is particularly important since the design of protocols is error-prone, and errors cannot be detected by testing, since they appear only in the presence of a malicious adversary. An important trend in this area aims to verify protocols in the so-called Dolev-Yao model [39], with an unbounded number of sessions, while relying as little as possible on human intervention. While protocol insecurity is NP-complete for a bounded number of sessions [65], it is undecidable for an unbounded number of sessions [41]. Hence, automatic verification for an unbounded number of sessions cannot be achieved for all protocols. It is typically achieved using language-based techniques such as typing or abstract interpretation, which can handle infinite-state systems thanks to safe approximations. These techniques are not complete (a correct protocol can fail to typecheck, or false attacks can be found by abstract interpretation tools), but they are sound (when they do not find attacks, the protocol is guaranteed to satisfy the considered property). This is important for the certification of protocols.

Our goal in this paper is to extend previous work in this line of research by providing a fully automatic technique for verifying correspondences in security protocols, without bounding the number of sessions of the protocol. Correspondences are properties of the form: if the protocol executes some event, then it must have executed some other events before[1]. We consider a rich language of correspondences, in which the events that must have been executed can be described by a logical formula containing conjunctions and disjunctions. Furthermore, we consider both non-injective correspondences (if the protocol executes some event, then it must have executed

*This paper is an updated and extended version of [13] and [14].

†This research has been done within the INRIA ABSTRACTION project-team (common with the CNRS and the ÉNS).

[1]In the CSP terminology, our events correspond to CSP signal events.

some other events at least once) and injective correspondences (if the protocol executes some event n times, then it must have executed some other events at least n times). Correspondences, initially named correspondence assertions [71], and the similar notion of agreement [54] were first introduced to model authentication. Intuitively, a protocol authenticates A to B if, when B thinks he talks to A, then he actually talks to A. When B thinks he has run the protocol with A, he executes an event $e(A, B)$. When A thinks she runs the protocol with B, she executes another event $e'(A, B)$. Authentication is satisfied when, if B executes his event $e(A, B)$, then A has executed her event $e'(A, B)$. Several variants along this scheme appear in the literature and, as we show below, our technique can handle most of them. Our correspondences can also encode secrecy, as follows. A protocol preserves the secrecy of some value M when the adversary cannot obtain M. We associate an "event" attacker(M) to the fact that the adversary obtains M, and represent the secrecy of M as "attacker(M) cannot be executed", that is, "if attacker(M) has been executed, then false." More complex properties can also be specified by our correspondences, for example that all messages of the protocol have been sent in order; this feature was used in [3].

Our technique is based on a substantial extension of a previous verification technique for secrecy [1, 13, 69]. More precisely, the protocol is represented in the process calculus introduced in [1], which is an extension of the pi calculus with fairly arbitrary cryptographic primitives. This process calculus is extended with events, used in the statement of correspondences. These events are the only required annotation of the protocol; no annotation is needed to help the tool proving correspondences. The protocol is then automatically translated into a set of Horn clauses. This translation requires significant extensions with respect to the translation for secrecy given in [1], and can be seen as an implementation of a type system, as in [1]. Some of these extensions improve the precision of the analysis, in particular to avoid merging different nonces. Other extensions define the translation of events. Finally, this set of Horn clauses is passed to a resolution-based solver, similar to that of [13, 20, 69]. Some minor extensions of this solver are required to prove correspondences. This solver does not always terminate, but we show in Section 8.1 that it terminates for a large class of well-designed protocols, named *tagged protocols*. Our experiments also demonstrate that, in practice, it terminates on many examples of protocols.

The main advantages of our method can be summarized as follows. It is fully automatic; the user only has to code the protocol and the correspondences to prove. It puts no bounds on the number of sessions of the protocol or the size of terms that the adversary can manipulate. It can handle fairly general cryptographic primitives, including shared-key encryption, public-key encryption, signatures, one-way hash functions, and Diffie-Hellman key agreements. It relies on a precise semantic foundation. One limitation of the technique is that, in rare cases, the solving algorithm does not terminate. The technique is also not complete: the translation into Horn clauses introduces an abstraction, which forgets the number of repetitions of each action [17]. This abstraction is key to the treatment of an unbounded number of sessions. Due to this abstraction, the tool provides sufficient conditions for proving correspondences, but can fail on correct protocols. Basically, it fails to prove protocols that first need to keep some value secret and later reveal it (see Section 5.2.2). In practice, the tool is still very precise and, in our experiments, it always succeeded in proving protocols that were correct.

Our technique is implemented in the protocol verifier ProVerif, available at `http://www.proverif.ens.fr/`.

Comparison with Other Papers on ProVerif As mentioned above, this paper extends previous work on the verification of secrecy [1] in order to prove correspondences. Secrecy (defined as the impossibility for the adversary to compute the secret) and correspondences are trace properties. Other papers deal with the proof of certain classes of observational equivalences, *i.e.*, that the adversary cannot distinguish certain processes: [15, 16] deal with the proof

of strong secrecy, *i.e.*, that the adversary cannot see when the value of a secret changes; [18] deals with the proof of equivalences between processes that differ only by the terms that they contain. Moreover, [18] also explains how to handle cryptographic primitives defined by equational theories (instead of rewrite rules) and how to deal with guessing attacks against weak secrets.

As shown in [20], the resolution algorithm terminates for tagged protocols. The present paper extends this result in Section 8.1, by providing a characterization of tagged protocols at the level of processes instead of at the level of Horn clauses.

ProVerif can also reconstruct an attack using a derivation from the Horn clauses, when the proof of a secrecy property fails [6]. Although the present paper does not detail this point, this work has also been extended to the reconstruction of attacks against non-injective correspondences.

Finally, [2], [3], and [19] present three case studies done at least partly using ProVerif: [2] studies a certified email protocol, [3] studies the Just Fast Keying protocol, and [19] studies the Plutus secure file system. These case studies rely partly on the results presented in this paper.

Related Work We mainly focus on the works that automatically verify correspondences and authentication for security protocols, without bounding the number of sessions.

The NRL protocol analyzer [42, 57], based on narrowing in rewriting systems, can verify correspondences defined in a rich language of logical formulae [68]. It is sound and complete, but does not always terminate. Our Horn clause representation is more abstract than the representation of NRL, which should enable us to terminate more often and be more efficient, while remaining precise enough to prove most desired properties.

Gordon and Jeffrey designed a system named Cryptic for verifying authentication by typing in security protocols [45–47]. They handle shared-key and public-key cryptography. Our system allows more general cryptographic primitives (including hash functions and Diffie-Hellman key agreements). Moreover, in our system, no annotation is needed, whereas, in Cryptic, explicit type casts and checks have to be manually added. However, Cryptic has the advantage that type checking always terminates, whereas, in some rare cases, our analyzer does not.

Bugliesi et al. [25] define another type system for proving authentication in security protocols. The main advantage of their system is that it is compositional: it allows one to prove independently the correctness of the code of each role of the protocol. However, the form of messages is restricted to certain tagged terms. This approach is compared with Cryptic in [24].

Backes et al. [10] prove secrecy and authentication for security protocols, using an abstract-interpretation-based analysis. This analysis builds a causal graph, which captures the causality among program events; the security properties are proved by traversing this graph. This analysis can handle an unbounded number of sessions of the protocol; it always terminates, at the cost of additional abstractions, which may cause false attacks. It handles shared-key and public-key cryptography, but not Diffie-Hellman key agreements. It assumes that the messages are typed, so that names can be distinguished from other terms.

Bodei et al. [21] show message authentication via a control flow analysis on a process calculus named Lysa. Like [10], they handle shared-key and public-key cryptography, and their analysis always terminates, at the cost of additional abstractions. The notion of authentication they prove is different from ours: they show message authentication rather than entity authentication.

Debbabi et al. [36] also verify authentication thanks to a representation of protocols by inference rules, very similar to our Horn clauses. However, they verify a weaker notion of authentication (corresponding to aliveness: if B terminates the protocol, then A must have been alive at some point before), and handle only shared-key encryption.

A few other methods require little human effort, while supporting an unbounded number of runs: the verifier of [51], based on rank functions, can prove the correctness of or find attacks against protocols with atomic symmetric or asymmetric keys. Theorem proving [63] often

requires manual intervention of the user. An exception to this is [32], but it deals only with secrecy. The theorem prover TAPS [30] often succeeds without or with little human intervention.

Model checking [53, 59] in general implies a limit on the number of sessions of the protocol. This problem has been tackled by [22, 23, 64]. They recycle nonces, to use only a finite number of them in an infinite number of runs. The technique was first used for sequential runs, then generalized to parallel runs in [23], but with the additional restriction that the agents must be "factorisable". (Basically, a single run of the agent has to be split into several runs such that each run contains only one fresh value.)

Strand spaces [44] are a formalism for reasoning about security protocols. They have been used for elegant manual proofs of authentication [49]. The automatic tool Athena [66] combines model checking and theorem proving, and uses strand spaces to reduce the state space. Scyther [33] uses an extension of Athena's method with trace patterns to analyze simultaneously a group of traces. These tools still sometimes limit the number of sessions to guarantee termination.

Amadio and Prasad [7] note that authentication can be translated into secrecy, by using a judge process. The translation is limited in that only one message can be registered by the judge, so the verified authentication property is not exactly the same as ours.

Outline Section 2 introduces our process calculus. Section 3 defines the correspondences that we verify, including secrecy and various notions of authentication. Section 4 outlines the main ideas behind our technique for verifying correspondences. Section 5 explains the construction of Horn clauses and shows its correctness, Section 6 describes our solving algorithm and shows its correctness, and Section 7 applies these results to the proof of correspondences. Section 8 discusses the termination of our algorithm: it shows termination for tagged protocols and how to obtain termination more often in the general case. Section 9 presents some extensions to our framework. Section 10 gives our experimental results on a selection of security protocols of the literature, and Section 11 concludes. The proofs of our results are grouped in the appendices.

2 The Process Calculus

In this section, we present the process calculus that we use to represent security protocols: we give its syntax, semantics, and illustrate it on an example protocol.

2.1 Syntax and Informal Semantics

Figure 1 gives the syntax of terms (data) and processes (programs) of our calculus. The identifiers a, b, c, k, and similar ones range over names, and x, y, and z range over variables. The syntax also assumes a set of symbols for constructors and destructors; we often use f for a constructor and g for a destructor.

Constructors are used to build terms. Therefore, the terms are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$; the terms are untyped. On the other hand, destructors do not appear in terms, but only manipulate terms in processes. They are partial functions on terms that processes can apply. The process $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$ tries to evaluate $g(M_1, \ldots, M_n)$; if this succeeds, then x is bound to the result and P is executed, else Q is executed. More precisely, the semantics of a destructor g of arity n is given by a set $\mathrm{def}(g)$ of rewrite rules of the form $g(M_1, \ldots, M_n) \to M$ where $M_1, \ldots, M_n, M$ are terms without names, and the variables of M also occur in $M_1, \ldots, M_n$. We extend these rules by $g(M'_1, \ldots, M'_n) \to M'$ if and only if there exist a substitution σ and a rewrite rule $g(M_1, \ldots, M_n) \to M$ in $\mathrm{def}(g)$ such that $M'_i = \sigma M_i$ for all $i \in \{1, \ldots, n\}$, and $M' = \sigma M$. We assume that the set $\mathrm{def}(g)$ is finite. (It usually contains one or two rules in examples.) We define destructors by rewrite rules instead of the equalities used in [1]. This definition allows

$M, N ::=$	terms
x, y, z	variable
a, b, c, k	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\overline{M}\langle N \rangle.P$	output
$M(x).P$	input
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
$\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$	destructor application
$\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$	conditional
$\mathtt{event}(M).P$	event

Figure 1: Syntax of the process calculus

destructors to yield several different results non-deterministically. (Non-deterministic rewrite rules are used in our modeling of Diffie-Hellman key agreements; see Section 9.1). Using constructors and destructors, we can represent data structures and cryptographic operations as summarized in Figure 2. (We present only probabilistic public-key encryption because, in the computational model, a secure public-key encryption algorithm must be probabilistic. We have chosen to present deterministic signatures; we could easily model probabilistic signatures by adding a third argument r containing the random coins, as for encryption. The coins should be chosen using a restriction (νa) which creates a fresh name a, representing a fresh random number.)

Constructors and destructors can be public or private. The public ones can be used by the adversary, which is the case when not stated otherwise. The private ones can be used only by honest participants. They are useful in practice to model tables of keys stored in a server, for instance. A public constructor *host* computes a host name from a long-term secret key, and a private destructor *getkey* returns the key from the host name, and simulates a lookup in a table of pairs (host name, key). Using a public constructor *host* allows the adversary to create and register any number of host names and keys. However, since *getkey* is private, the adversary cannot compute a key from the host name, which would break all protocols: host names are public while keys of honest participants are secret.

The process calculus provides additional instructions for executing events, which will be used for specifying correspondences. The process $\mathtt{event}(M).P$ executes the event $\mathtt{event}(M)$, then executes P.

The other constructs in the syntax of Figure 1 are standard; most of them come from the pi calculus. The input process $M(x).P$ inputs a message on channel M, and executes P with x bound to the input message. The output process $\overline{M}\langle N \rangle.P$ outputs the message N on the channel M and then executes P. We allow communication on channels that can be arbitrary terms. (We could adapt our work to the case in which channels are only names.) Our calculus is monadic (in that the messages are terms rather than tuples of terms), but a polyadic calculus can be simulated since tuples are terms. It is also synchronous (in that a process P is executed after the output of a message). The nil process 0 does nothing. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents an unbounded number of copies of P in parallel. The restriction $(\nu a)P$ creates a new name a and then executes P. The conditional $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ executes P if M and N reduce to the same term at runtime; otherwise,

Tuples:
Constructor: tuple $ntuple(x_1, \dots, x_n)$
Destructors: projections $ith_n(ntuple(x_1, \dots, x_n)) \rightarrow x_i$
Shared-key encryption:
Constructor: encryption of x under the key y, $sencrypt(x, y)$
Destructor: decryption $sdecrypt(sencrypt(x, y), y) \rightarrow x$
Probabilistic shared-key encryption:
Constructor: encryption of x under the key y with random coins r, $sencrypt_p(x, y, r)$
Destructor: decryption $sdecrypt_p(sencrypt_p(x, y, r), y) \rightarrow x$
Probabilistic public-key encryption:
Constructors: encryption of x under the key y with random coins r, $pencrypt_p(x, y, r)$
public key generation from a secret key y, $pk(y)$
Destructor: decryption $pdecrypt_p(pencrypt_p(x, pk(y), r), y) \rightarrow x$
Signatures:
Constructors: signature of x with the secret key y, $sign(x, y)$
public key generation from a secret key y, $pk(y)$
Destructors: signature verification $checksignature(sign(x, y), pk(y)) \rightarrow x$
message without signature $getmessage(sign(x, y)) \rightarrow x$
Non-message-revealing signatures:
Constructors: signature of x with the secret key y, $nmrsign(x, y)$
public key generation from a secret key y, $pk(y)$
constant *true*
Destructor: verification $nmrchecksign(nmrsign(x, y), pk(y), x) \rightarrow true$
One-way hash functions:
Constructor: hash function $h(x)$
Table of host names and keys
Constructor: host name from key $host(x)$
Private destructor: key from host name $getkey(host(x)) \rightarrow x$

Figure 2: Constructors and destructors

it executes Q. We define *let* $x = M$ *in* P as syntactic sugar for $P\{M/x\}$. As usual, we may omit an *else* clause when it consists of 0.

The name a is bound in the process $(\nu a)P$. The variable x is bound in P in the processes $M(x).P$ and *let* $x = g(M_1, \dots, M_n)$ *in* P *else* Q. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively. A process is closed if it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables. We write $\{M_1/x_1, \dots, M_n/x_n\}$ for the substitution that replaces $x_1, \dots, x_n$ with $M_1, \dots, M_n$, respectively.

2.2 Operational Semantics

A semantic configuration is a pair $E, \mathcal{P}$ where the environment E is a finite set of names and $\mathcal{P}$ is a finite multiset of closed processes. The environment E must contain at least all free names of processes in $\mathcal{P}$. The configuration $\{a_1, \dots, a_n\}, \{P_1, \dots, P_n\}$ corresponds intuitively to the process $(\nu a_1) \dots (\nu a_n)(P_1 \mid \dots \mid P_n)$. The semantics of the calculus is defined by a reduction relation $\rightarrow$ on semantic configurations, shown in Figure 3. The rule (Red Res) is the only one that uses renaming. This is important so that the parameters of events are not renamed after the execution of the event, to be able to compare them with the parameters of events executed later. This semantics is superficially different from those of [1, 14], which were defined using a structural congruence relation and a reduction relation on processes. The new semantics (in particular the renaming point mentioned above) provides simplifications in the definitions of

$$
\begin{array}{lr}
E, \mathcal{P} \cup \{\, 0 \,\} \rightarrow E, \mathcal{P} & \text{(Red Nil)} \\
E, \mathcal{P} \cup \{\, !P \,\} \rightarrow E, \mathcal{P} \cup \{\, P, !P \,\} & \text{(Red Repl)} \\
E, \mathcal{P} \cup \{\, P \mid Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P, Q \,\} & \text{(Red Par)} \\
E, \mathcal{P} \cup \{\, (\nu a)P \,\} \rightarrow E \cup \{a'\}, \mathcal{P} \cup \{\, P\{a'/a\} \,\} & \text{(Red Res)} \\
\quad \text{where } a' \notin E. & \\
E, \mathcal{P} \cup \{\, \overline{N}\langle M \rangle.Q, N(x).P \,\} \rightarrow E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\} & \text{(Red I/O)} \\
E, \mathcal{P} \cup \{\, \mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P\{M'/x\} \,\} & \\
\quad \text{if } g(M_1, \ldots, M_n) \rightarrow M' & \text{(Red Destr 1)} \\
E, \mathcal{P} \cup \{\, \mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} & \text{(Red Destr 2)} \\
\quad \text{if there exists no } M' \text{ such that } g(M_1, \ldots, M_n) \rightarrow M' & \\
E, \mathcal{P} \cup \{\, \mathit{if}\ M = M\ \mathit{then}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\} & \text{(Red Cond 1)} \\
E, \mathcal{P} \cup \{\, \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} & \text{(Red Cond 2)} \\
\quad \text{if } M \neq N & \\
E, \mathcal{P} \cup \{\, \mathtt{event}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\} & \text{(Red Event)}
\end{array}
$$

Figure 3: Operational semantics

correspondences (Definitions 2, 3, 6, 7, and 9) and in the proofs that correspondences hold.

2.3 Example

As a running example, we consider a simplified version of the Needham-Schroeder public-key protocol [60], with the correction by Lowe [53], in which host names are replaced by public keys, which makes interaction with a server useless. (The version tested in the benchmarks is the full version. Obviously, our tool can verify much more complex protocols; we use this simple example for illustrative purposes.) The protocol contains the following messages:

$$
\begin{array}{lll}
\text{Message 1.} & A \rightarrow B: & \{a, pk_A\}_{pk_B} \\
\text{Message 2.} & B \rightarrow A: & \{a, b, pk_B\}_{pk_A} \\
\text{Message 3.} & A \rightarrow B: & \{b\}_{pk_B}
\end{array}
$$

A first sends to B a nonce (fresh name) a encrypted under the public key of B. B decrypts this message using his secret key sk_B and replies with the nonce a, a fresh nonce he chooses b, and its own public key pk_B, all encrypted under pk_A. When A receives this message, she decrypts it. When A sees the nonce a, she is convinced that B answered since only B can decrypt the first message and obtain a. Then A replies with the nonce b encrypted under pk_B. B decrypts this message. When B sees the nonce b, he is convinced that A replied, since only A could decrypt the second message and obtain b. The presence of pk_A in the first message and pk_B in the second message makes explicit that these messages are for sessions between A and B, and so avoids man-in-the-middle attacks, such as the well-known attack found by Lowe [53]. This

protocol can be represented in our calculus by the process P, explained below:

$$\begin{aligned}
&P_A(sk_A, pk_A, pk_B) = !c(x_pk_B).(\nu a)\texttt{event}(e_1(pk_A, x_pk_B, a)).\\
&\quad (\nu r_1)\overline{c}\langle pencrypt_p((a, pk_A), x_pk_B, r_1)\rangle.\\
&\quad c(m).let\ (= a, x_b, = x_pk_B) = pdecrypt_p(m, sk_A)\ in\\
&\quad \texttt{event}(e_3(pk_A, x_pk_B, a, x_b)).(\nu r_3)\overline{c}\langle pencrypt_p(x_b, x_pk_B, r_3)\rangle\\
&\quad if\ x_pk_B = pk_B\ then\\
&\quad \texttt{event}(e_A(pk_A, x_pk_B, a, x_b)).\overline{c}\langle sencrypt(sAa, a)\rangle.\overline{c}\langle sencrypt(sAb, x_b)\rangle\\
&P_B(sk_B, pk_B, pk_A) = !c(m').let\ (x_a, x_pk_A) = pdecrypt_p(m', sk_B)\ in\ (\nu b)\\
&\quad \texttt{event}(e_2(x_pk_A, pk_B, x_a, b)).(\nu r_2)\overline{c}\langle pencrypt_p((x_a, b, pk_B), x_pk_A, r_2)\rangle.\\
&\quad c(m'').let\ (= b) = pdecrypt_p(m'', sk_B)\ in\\
&\quad if\ x_pk_A = pk_A\ then\\
&\quad \texttt{event}(e_B(x_pk_A, pk_B, x_a, b)).\overline{c}\langle sencrypt(sBa, x_a)\rangle.\overline{c}\langle sencrypt(sBb, b)\rangle\\
&P = (\nu sk_A)(\nu sk_B)let\ pk_A = pk(sk_A)\ in\ let\ pk_B = pk(sk_B)\ in\\
&\quad \overline{c}\langle pk_A\rangle\overline{c}\langle pk_B\rangle.(P_A(sk_A, pk_A, pk_B) \mid P_B(sk_B, pk_B, pk_A))
\end{aligned}$$

The channel c is public: the adversary can send and listen on it. We use a single public channel and not two or more channels because the adversary could take a message from one channel and relay it on another channel, thus removing any difference between the channels. The process P begins with the creation of the secret and public keys of A and B. The public keys are output on channel c to model that the adversary has them in its initial knowledge. Then the protocol itself starts: P_A represents A, P_B represents B. Both principals can run an unbounded number of sessions, so P_A and P_B start with replications.

We consider that A and B are both willing to talk to any principal. So, to determine to whom A will talk, we consider that A first inputs a message containing the public key x_pk_B of its interlocutor. (This interlocutor is therefore chosen by the adversary.) Then A starts a protocol run by choosing a nonce a, and executing the event $e_1(pk_A, x_pk_B, a)$. Intuitively, this event records that A sent Message 1 of the protocol, for a run with the participant of public key x_pk_B, using the nonce a. Event e_1 is placed before the actual output of Message 1; this is necessary for the desired correspondences to hold: if event e_1 followed the output of Message 1, one would not be able to prove that event e_1 must have been executed, even though Message 1 must have been sent, because Message 1 could be sent without executing event e_1. The situation is similar for events e_2 and e_3 below. Then A sends the first message of the protocol $pencrypt_p((a, pk_A), x_pk_B, r_1)$, where r_1 are fresh coins, used to model that public-key encryption is probabilistic. A waits for the second message and decrypts it using her secret key sk_A. If decryption succeeds, A checks that the message has the right form using the pattern-matching construct $let\ (= a, x_b, = x_pk_B) = pdecrypt_p(m, sk_A)\ in\ \ldots$ This construct is syntactic sugar for $let\ y = pdecrypt_p(m, sk_A)\ in\ let\ x_1 = 1th_3(y)\ in\ let\ x_b = 2th_3(y)\ in\ let\ x_3 = 3th_3(y)\ in\ if\ x_1 = a\ then\ if\ x_3 = x_pk_B\ then\ \ldots$ Then A executes the event $e_3(pk_A, x_pk_B, a, x_b)$, to record that she has received Message 2 and sent Message 3 of the protocol, in a session with the participant of public key x_pk_B, and nonces a and x_b. Finally, she sends the last message of the protocol $pencrypt_p(x_b, x_pk_B, r_3)$. After sending this message, A executes some actions needed only for specifying properties of the protocol. When $x_pk_B = pk_B$, that is, when the session is between A and B, A executes the event $e_A(pk_A, x_pk_B, a, x_b)$, to record that A ended a session of the protocol, with the participant of public key x_pk_B and nonces a and x_b. A also outputs the secret name sAa encrypted under the nonce a and the secret name sAb encrypted under the nonce x_b. These outputs are helpful in order to formalize the secrecy of the nonces. Our tool can prove the secrecy of free names, but not the secrecy of bound names (such as a) or of variables (such as x_b). In order to overcome this limitation, we publish the encryption of a free

name sAa under a; then sAa is secret if and only if the nonce a chosen by A is secret. Similarly, sAb is secret if and only if the nonce x_b received by A is secret.

The process P_B proceeds similarly: it executes the protocol, with the additional event $e_2(x_pk_A, pk_B, x_a, b)$ to record that Message 1 has been received and Message 2 has been sent by B, in a session with the participant of public key x_pk_A and nonces x_a and b. After finishing the protocol itself, when $x_pk_A = pk_A$, that is, when the session is between A and B, P_B executes the event $e_B(x_pk_A, pk_B, x_a, b)$, to record that B finished the protocol, and outputs sBa encrypted under x_a and sBb encrypted under b, to model the secrecy of x_a and b respectively.

The events will be used in order to formalize authentication. For example, we formalize that, if A ends a session of the protocol, then B has started a session of the protocol with the same nonces by requiring that, if $e_A(x_1, x_2, x_3, x_4)$ has been executed, then $e_2(x_1, x_2, x_3, x_4)$ has been executed.[2]

3 Definition of Correspondences

In this section, we formally define the correspondences that we verify. We prove correspondences of the form "if an event e has been executed, then events $e_{11}, \dots, e_{1l_1}$ have been executed, or ..., or $e_{m1}, \dots, e_{ml_m}$ have been executed". These events may include arguments, which allows one to relate the values of variables at the various events. Furthermore, we can replace the event e with the fact that the adversary knows some term (which allows us to prove secrecy properties), or that a certain message has been sent on a certain channel. We can prove that each execution of e corresponds to a distinct execution of some events e_{jk} (injective correspondences, defined in Section 3.2), and we can prove that the events e_{jk} have been executed in a certain order (general correspondences, defined in Section 3.3).

We assume that the protocol is executed in the presence of an adversary that can listen to all messages, compute, and send all messages it has, following the so-called Dolev-Yao model [39]. Thus, an adversary can be represented by any process that has a set of public names *Init* in its initial knowledge and that does not contain events. (Although the initial knowledge of the adversary contains only names in *Init*, one can give any terms to the adversary by sending them on a channel in *Init*.)

Definition 1 Let *Init* be a finite set of names. The closed process Q is an *Init-adversary* if and only if $fn(Q) \subseteq Init$ and Q does not contain events.

3.1 Non-injective Correspondences

Next, we define when a trace satisfies an atom α, generated by the following grammar:

$\alpha ::=$	atom
$\quad$ attacker(M)	$\quad$ attacker knowledge
$\quad$ message(M, M')	$\quad$ message on a channel
$\quad$ event(M)	$\quad$ event

Intuitively, a trace satisfies attacker(M) when the attacker has M, or equivalently, when M has been sent on a public channel in *Init*. It satisfies message(M, M') when the message M' has been sent on channel M. Finally, it satisfies event(M) when the event `event`(M) has been executed.

[2] For this purpose, the event e_A must not be executed when A thinks she talks to the adversary. Indeed, in this case, it is correct that no event has been executed by the interlocutor of A, since the adversary never executes events.

Definition 2 We say that a trace $\mathcal{T} = E_0, \mathcal{P}_0 \to^* E', \mathcal{P}'$ satisfies attacker(M) if and only if $\mathcal{T}$ contains a reduction $E, \mathcal{P} \cup \{ \overline{c}\langle M\rangle.Q, c(x).P \} \to E, \mathcal{P} \cup \{ Q, P\{M/x\} \}$ for some E, $\mathcal{P}$, x, P, Q, and $c \in \mathit{Init}$.

We say that a trace $\mathcal{T} = E_0, \mathcal{P}_0 \to^* E', \mathcal{P}'$ satisfies message(M, M') if and only if $\mathcal{T}$ contains a reduction $E, \mathcal{P} \cup \{ \overline{M}\langle M'\rangle.Q, M(x).P \} \to E, \mathcal{P} \cup \{ Q, P\{M'/x\} \}$ for some E, $\mathcal{P}$, x, P, Q.

We say that a trace $\mathcal{T} = E_0, \mathcal{P}_0 \to^* E', \mathcal{P}'$ satisfies event(M) if and only if $\mathcal{T}$ contains a reduction $E, \mathcal{P} \cup \{ \mathtt{event}(M).P \} \to E, \mathcal{P} \cup \{ P \}$ for some E, $\mathcal{P}$, P.

The correspondence $\alpha \Rightarrow \bigvee_{j=1}^{m} \left(\alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \text{event}(M_{jk})\right)$, formally defined below, means intuitively that, if an instance of α is satisfied, then for some $j \in \{1, \dots, m\}$, the considered instance of α is an instance of α_j and a corresponding instance of each of the events $\mathtt{event}(M_{j1})$, $\dots$, $\mathtt{event}(M_{jl_j})$ has been executed.[3]

Definition 3 The closed process P_0 *satisfies the correspondence*

$$\alpha \Rightarrow \bigvee_{j=1}^{m} \left(\alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \text{event}(M_{jk})\right)$$

against *Init*-adversaries if and only if, for any *Init*-adversary Q, for any E_0 containing $\mathit{fn}(P_0) \cup \mathit{Init} \cup \mathit{fn}(\alpha) \cup \bigcup_j \mathit{fn}(\alpha_j) \cup \bigcup_{j,k} \mathit{fn}(M_{jk})$, for any substitution σ, for any trace $\mathcal{T} = E_0, \{P_0, Q\} \to^* E', \mathcal{P}'$, if $\mathcal{T}$ satisfies $\sigma\alpha$, then there exist σ' and $j \in \{1, \dots, m\}$ such that $\sigma'\alpha_j = \sigma\alpha$ and, for all $k \in \{1, \dots, l_j\}$, $\mathcal{T}$ satisfies event$(\sigma' M_{jk})$ as well.

This definition is very general; we detail some interesting particular cases below. When $m = 0$, the disjunction $\bigvee_{j=1}^{m} \dots$ is denoted by false. When $\alpha = \alpha_j$ for all j, we abbreviate the correspondence by $\alpha \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \text{event}(M_{jk})$. This correspondence means that, if an instance of α is satisfied, then for some $j \le m$, a corresponding instance of $\mathtt{event}(M_{j1})$, $\dots$, $\mathtt{event}(M_{jl_j})$ has been executed. The variables in α are universally quantified (because, in Definition 3, σ is universally quantified). The variables in M_{jk} that do not occur in α are existentially quantified (because σ' is existentially quantified).

Example 1 In the process of Section 2.3, the correspondence event$(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow$ event$(e_1(x_1, x_2, x_3)) \wedge$ event$(e_2(x_1, x_2, x_3, x_4)) \wedge$ event$(e_3(x_1, x_2, x_3, x_4))$ means that, if the event $e_B(x_1, x_2, x_3, x_4)$ has been executed, then the events $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$ have been executed, with the same value of the arguments x_1, x_2, x_3, x_4.

The correspondence

$$\begin{aligned}
&\text{event}(\mathit{R_received}(\mathit{msg}(x, z))) \Rightarrow \\
&\quad (\text{event}(\mathit{R_received}(\mathit{msg}(x, (z', \mathit{Auth})))) \rightsquigarrow \\
&\quad\ \text{event}(\mathit{S_has}(k, \mathit{msg}(x, (z', \mathit{Auth}))))\wedge \\
&\quad\ \text{event}(\mathit{TTP_send}(\mathit{sign}((\mathit{sencrypt}(\mathit{msg}(x, (z', \mathit{Auth})), k), x), \mathit{sk}_{\mathit{TTP}})))) \\
&\vee (\text{event}(\mathit{R_received}(\mathit{msg}(x, (z', \mathit{NoAuth})))) \rightsquigarrow \\
&\quad\ \text{event}(\mathit{S_has}(k, \mathit{msg}(x, (z', \mathit{NoAuth}))))\wedge \\
&\quad\ \text{event}(\mathit{TTP_send}(\mathit{sign}(\mathit{sencrypt}(\mathit{msg}(x, (z', \mathit{NoAuth})), k), \mathit{sk}_{\mathit{TTP}}))))
\end{aligned}$$

means that, if the event $\mathit{R_received}(\mathit{msg}(x, z))$ has been executed, then two cases can happen: either $z = (z', \mathit{Auth})$ or $z = (z', \mathit{NoAuth})$ for some z'. In both cases, the events $\mathit{TTP_send}(\mathit{certificate})$ and $\mathit{S_has}(k, \mathit{msg}(x, z))$ have been executed for some k, but with a different value of *certificate*: $\mathit{certificate} = \mathit{sign}((\mathit{S2TTP}, x), \mathit{sk}_{\mathit{TTP}})$ when $z = (z', \mathit{Auth})$, and

[3] The implementation in ProVerif uses a slightly different notation: α_j is omitted, but additionnally equality tests are allowed on the right-hand side of $\rightsquigarrow$, so that one can check that α is actually an instance of α_j.

certificate $=$ *sign*(*S2TTP*$, sk_{TTP}$) when $z = (z', NoAuth)$, with *S2TTP* $=$ *sencrypt*(*msg*(x,z), k). A similar correspondence was used in our study of a certified email protocol, in collaboration with Martín Abadi [2, Section 5, Proposition 4]. We refer to that paper for additional details.

The following definitions are particular cases of Definition 3.

Definition 4 The closed process P *preserves the secrecy of all instances of* M from *Init* if and only if it satisfies the correspondence $\text{attacker}(M) \rightsquigarrow \text{false}$ against *Init*-adversaries.

When M is a free name, this definition is equivalent to that of [1].

Example 2 The process P of Section 2.3 preserves the secrecy of sAa when the correspondence $\text{attacker}(sAa) \rightsquigarrow \text{false}$ is satisfied. In this case, intuitively, P preserves the secrecy of the nonce a that A chooses. The situation is similar for sAb, sBa, and sBb.

Definition 5 *Non-injective agreement* is a correspondence of the form $\text{event}(e(x_1,\ldots,x_n)) \rightsquigarrow \text{event}(e'(x_1,\ldots,x_n))$.

Intuitively, the correspondence $\text{event}(e(x_1,\ldots,x_n)) \rightsquigarrow \text{event}(e'(x_1,\ldots,x_n))$ means that, if an event $e(M_1,\ldots,M_n)$ is executed, then the event $e'(M_1,\ldots,M_n)$ has also been executed. This definition can be used to represent Lowe's notion of non-injective agreement [54].

Example 3 In the example of Section 2.3, the correspondence $\text{event}(e_A(x_1,x_2,x_3,x_4)) \rightsquigarrow \text{event}(e_2(x_1,x_2,x_3,x_4))$ means that, if A executes an event $e_A(x_1,x_2,x_3,x_4)$, then B has executed the event $e_2(x_1,x_2,x_3,x_4)$. So, if A terminates the protocol thinking she talks to B, then B is actually involved in the protocol. Moreover, the agreement on the parameter of the events, $pk_A = x_pk_A$, $x_pk_B = pk_B$, $a = x_a$, and $x_b = b$ implies that B actually thinks he talks to A, and that A and B agree on the values of the nonces.

The correspondence $\text{event}(e_B(x_1,x_2,x_3,x_4)) \rightsquigarrow \text{event}(e_3(x_1,x_2,x_3,x_4))$ is similar, after swapping the roles of A and B.

3.2 Injective Correspondences

Definition 6 We say that the event $\texttt{event}(M)$ is executed at step τ in a trace $\mathcal{T} = E_0, \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ if and only if the τ-th reduction of $\mathcal{T}$ is of the form $E, \mathcal{P} \cup \{\, \texttt{event}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\}$ for some E, $\mathcal{P}$, P.

Intuitively, an injective correspondence $\text{event}(M) \rightsquigarrow \text{inj event}(M')$ requires that each event $\texttt{event}(\sigma M)$ is enabled by distinct events $\texttt{event}(\sigma M')$, while a non-injective correspondence $\text{event}(M) \rightsquigarrow \text{event}(M')$ allows several events $\texttt{event}(\sigma M)$ to be enabled by the same event $\texttt{event}(\sigma M')$. We denote by [inj] an optional inj marker: it can be either inj or nothing. When [inj] = inj, an injective correspondence is required. When [inj] is nothing, the correspondence does not need to be injective.

Definition 7 The closed process P_0 *satisfies the correspondence*

$$\text{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(N_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} \text{event}(M_{jk}) \right)$$

against *Init*-adversaries if and only if, for any *Init*-adversary Q, for any E_0 containing $fn(P_0) \cup Init \cup fn(M) \cup \bigcup_j fn(N_j) \cup \bigcup_{j,k} fn(M_{jk})$, for any trace $\mathcal{T} = E_0, \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, there exist functions ϕ_{jk} from a subset of steps in $\mathcal{T}$ to steps in $\mathcal{T}$ such that

- For all τ, if the event $\texttt{event}(\sigma M)$ is executed at step τ in $\mathcal{T}$ for some σ, then there exist σ' and j such that $\sigma' N_j = \sigma M$ and, for all $k \in \{1, \ldots, l_j\}$, $\phi_{jk}(\tau)$ is defined and $\texttt{event}(\sigma' M_{jk})$ is executed at step $\phi_{jk}(\tau)$ in $\mathcal{T}$.
- If $[\text{inj}]_{jk} = \text{inj}$, then ϕ_{jk} is injective.

The functions ϕ_{jk} map execution steps of events $\texttt{event}(\sigma M)$ to the execution steps of the events $\texttt{event}(\sigma' M_{jk})$ that enable $\texttt{event}(\sigma M)$. When $[\text{inj}]_{jk} = \text{inj}$, the injectivity of ϕ_{jk} guarantees that distinct executions of $\texttt{event}(\sigma M)$ correspond to distinct executions of $\texttt{event}(\sigma' M_{jk})$. When $M = N_j$ for all j, we abbreviate the correspondence by $\text{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} \text{event}(M_{jk})$, as in the non-injective case.

Woo and Lam's correspondence assertions [71] are a particular case of this definition. Indeed, they consider properties of the form: if γ_1 or ... or γ_k have been executed, then μ_1 or ... or μ_m must have been executed, denoted by $\gamma_1 \mid \ldots \mid \gamma_k \hookrightarrow \mu_1 \mid \ldots \mid \mu_m$. Such a correspondence assertion is formalized in our setting by for all $i \in \{1, \ldots, k\}$, the process satisfies the correspondence $\text{event}(\gamma_i) \rightsquigarrow \bigvee_{j=1}^{m} \text{inj event}(\mu_j)$.

Remark 1 Correspondences $\alpha \Rightarrow \bigvee_{j=1}^{m} \left(\alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} \text{event}(M_{jk})\right)$ with $\alpha = \text{attacker}(M)$ and at least one inj marker would always be wrong: the adversary can always repeat the output of M on one of his channels any number of times. With $\alpha = \text{message}(M, M')$ and at least one inj marker, the correspondence may be true only when the adversary cannot execute the corresponding output. For simplicity, we focus on the case $\alpha = \text{event}(M)$ only.

Definition 8 *Injective agreement* is a correspondence of the form $\text{event}(e(x_1, \ldots, x_n)) \rightsquigarrow \text{inj event}(e'(x_1, \ldots, x_n))$.

Injective agreement requires that the number of executions of $\texttt{event}(e(M_1, \ldots, M_n))$ is smaller than the number of executions of $\texttt{event}(e'(M_1, \ldots, M_n))$: each execution of $\texttt{event}(e(M_1, \ldots, M_n))$ corresponds to a distinct execution of $\texttt{event}(e'(M_1, \ldots, M_n))$. This corresponds to Lowe's agreement specification [54].

Example 4 In the example of Section 2.3, the correspondence $\text{event}(e_A(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_2(x_1, x_2, x_3, x_4))$ means that each execution of $\text{event}(e_A(x_1, x_2, x_3, x_4))$ corresponds to a distinct execution of $\text{event}(e_2(x_1, x_2, x_3, x_4))$. So each completed session of A talking to B corresponds to a distinct session of B talking to A, and A and B agree on the values of the nonces.

The correspondence $\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_3(x_1, x_2, x_3, x_4))$ is similar, after swapping the roles of A and B.

3.3 General Correspondences

Correspondences also give information on the order in which events are executed. Indeed, if we have the correspondence

$$\text{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(N_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} \text{event}(M_{jk})\right)$$

then the events $\text{event}(M_{jk})$ for $k \leq l_j$ have been executed before $\text{event}(N_j)$. Formally, in the definition of injective correspondences, we can define ϕ_{jk} such that $\phi_{jk}(\tau) \leq \tau$ when ϕ_{jk} is defined. (The inequality $\tau' \leq \tau$ means that τ' occurs before τ in the trace.) Indeed, otherwise, by considering the prefix of the trace that stops just after τ, we would contradict the correspondence. In this section, we exploit this point to define more general properties involving the ordering of events.

Let us first consider some examples. Using the process of Section 2.3, we will denote by

$$\begin{aligned}&\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \\ &\quad(\text{inj event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_1(x_1, x_2, x_3))))\end{aligned} \tag{1}$$

the correspondence that means that each execution of the event $e_B(x_1, x_2, x_3, x_4)$ corresponds to distinct executions of the events $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$ in this order: each execution of $e_B(x_1, x_2, x_3, x_4)$ is preceded by a distinct execution of $e_3(x_1, x_2, x_3, x_4)$, which is itself preceded by a distinct execution of $e_2(x_1, x_2, x_3, x_4)$, which is itself preceded by a distinct execution of $e_1(x_1, x_2, x_3)$. This correspondence shows that, when B terminates the protocol talking with A, A and B have exchanged all messages of the protocol in the expected order. This correspondence is not equivalent to the conjunction of the correspondences $\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_3(x_1, x_2, x_3, x_4))$, $\text{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_2(x_1, x_2, x_3, x_4))$, and $\text{event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_1(x_1, x_2, x_3))$, because (1) may be true even when, in order to prove that e_2 is executed, we need to know that e_B has been executed, and not only that e_3 has been executed and, similarly, in order to prove that e_1 has been executed, we need to know that e_B has been executed, and not only that e_2 has been executed. Using general correspondences such as (1) is therefore strictly more expressive than using injective correspondences. A correspondence similar to (1) has been used in our study of the Just Fast Keying protocol, one of the proposed replacements for IKE in IPSec, in collaboration with Martín Abadi and Cédric Fournet [3, Appendix B.5].

As a more generic example, the correspondence $\text{event}(M) \Rightarrow \bigvee_{j=1}^{m} \big(\text{event}(M_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \big([\text{inj}]_{jk}\text{event}(M_{jk}) \rightsquigarrow \bigvee_{j'=1}^{m_{jk}} \bigwedge_{k'=1}^{l_{jkj'}} [\text{inj}]_{jkj'k'}\text{event}(M_{jkj'k'})\big)\big)$ means that, if an instance of $\mathtt{event}(M)$ has been executed, then there exists j such that this instance of $\mathtt{event}(M)$ is an instance of $\mathtt{event}(M_j)$ and for all k, a corresponding instance of $\mathtt{event}(M_{jk})$ has been executed before $\mathtt{event}(M_j)$, and there exists j'_k such that for all k' a corresponding instance of $\mathtt{event}(M_{jkj'_kk'})$ has been executed before $\mathtt{event}(M_{jk})$.

Let us now consider the general definition. We denote by $\overline{k}$ a sequence of indices k. The empty sequence is denoted ϵ. When $\overline{j} = j_1 \dots j_n$ and $\overline{k} = k_1 \dots k_n$ are sequences of the same length, we denote by $\overline{jk}$ the sequence obtained by taking alternatively one index in each sequence $\overline{j}$ and $\overline{k}$: $\overline{jk} = j_1k_1 \dots j_nk_n$. We sometimes use $\overline{jk}$ as an identifier that denotes a sequence obtained in this way; for instance, "for all $\overline{jk}$, $\phi_{\overline{jk}}$ is injective" abbreviates "for all $\overline{j}$ and $\overline{k}$ of the same length, $\phi_{\overline{jk}}$ is injective". We only consider sequences $\overline{jk}$ that occur in the correspondence. For instance, for the correspondence $\text{event}(M) \Rightarrow \bigvee_{j=1}^{m} \big(\text{event}(M_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \big([\text{inj}]_{jk}\text{event}(M_{jk}) \rightsquigarrow \bigvee_{j'=1}^{m_{jk}} \bigwedge_{k'=1}^{l_{jkj'}} [\text{inj}]_{jkj'k'}\text{event}(M_{jkj'k'})\big)\big)$, we consider the sequences $\overline{jk} = \epsilon$, $\overline{jk} = jk$, and $\overline{jk} = jkj'k'$ where $1 \leq j \leq m$, $1 \leq k \leq l_j$, $1 \leq j' \leq m_{jk}$, and $1 \leq k' \leq l_{jkj'}$.

Given a family of indices $J = (j_{\overline{k}})_{\overline{k}}$ indexed by sequences of indices $\overline{k}$, we define $\text{makejk}(\overline{k}, J)$ by $\text{makejk}(\epsilon, J) = \epsilon$ and $\text{makejk}(\overline{k}k, J) = \text{makejk}(\overline{k}, J)j_{\overline{k}}k$. Less formally, if $\overline{k} = k_1k_2k_3\dots$, we have $\text{makejk}(\overline{k}, J) = j_\epsilon k_1 j_{k_1} k_2 j_{k_1k_2} k_3 \dots$ Intuitively, the correspondence contains disjunctions over indices j and conjunctions over indices k, so we would like to express quantifications of the form $\exists j_\epsilon \forall k_1 \exists j_{k_1} \forall k_2 \exists j_{k_1k_2} \forall k_3 \dots$ on the sequence $j_\epsilon k_1 j_{k_1} k_2 j_{k_1k_2} k_3 \dots$. The notation $\text{makejk}(\overline{k}, J)$ allows us to replace such a quantification with the quantification $\exists J \forall \overline{k}$ on the sequence $\text{makejk}(\overline{k}, J)$.

Definition 9 The closed process P_0 *satisfies the correspondence*

$$\text{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(M_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} q_{jk}\right)$$

where

$$q_{\overline{jk}} = \text{event}(M_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

against *Init*-adversaries if and only if, for any *Init*-adversary Q, for any E_0 containing $fn(P_0) \cup \mathit{Init} \cup fn(M) \cup \bigcup_j fn(M_j) \cup \bigcup_{\overline{jk}} fn(M_{\overline{jk}})$, for any trace $\mathcal{T} = E_0, \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, there exists a function $\phi_{\overline{jk}}$ for each non-empty $\overline{jk}$, such that for all non-empty $\overline{jk}$, $\phi_{\overline{jk}}$ maps a subset of steps of $\mathcal{T}$ to steps of $\mathcal{T}$ and

- For all τ, if the event $\texttt{event}(\sigma M)$ is executed at step τ in $\mathcal{T}$ for some σ, then there exist σ' and $J = (j_{\overline{k}})_{\overline{k}}$ such that $\sigma' M_{j_\epsilon} = \sigma M$ and, for all non-empty $\overline{k}$, $\phi_{\text{makejk}(\overline{k},J)}(\tau)$ is defined and $\texttt{event}(\sigma' M_{\text{makejk}(\overline{k},J)})$ is executed at step $\phi_{\text{makejk}(\overline{k},J)}(\tau)$ in $\mathcal{T}$.

- For all non-empty $\overline{jk}$, if $[\text{inj}]_{\overline{jk}} = \text{inj}$, then $\phi_{\overline{jk}}$ is injective.

- For all non-empty $\overline{jk}$, for all j and k, if $\phi_{\overline{jk}jk}(\tau)$ is defined, then $\phi_{\overline{jk}}(\tau)$ is defined and $\phi_{\overline{jk}jk}(\tau) \leq \phi_{\overline{jk}}(\tau)$. For all j and k, if $\phi_{jk}(\tau)$ is defined, then $\phi_{jk}(\tau) \leq \tau$.

We abbreviate by $q_{\overline{jk}} = \text{event}(M_{\overline{jk}})$ the correspondence

$$q_{\overline{jk}} = \text{event}(M_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

when $m_{\overline{jk}} = 1$ and $l_{\overline{jk}1} = 0$, that is, the disjunction $\bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$ is true. Injective correspondences are then a particular case of general correspondences.

The function $\phi_{\overline{jk}}$ maps the execution steps of instances of $\texttt{event}(M)$ to the execution steps of the corresponding instances of $\texttt{event}(M_{\overline{jk}})$. The first item of Definition 9 guarantees that the required events have been executed. The second item means that, when the inj marker is present, the correspondence is injective. Finally, the third item guarantees that the events have been executed in the expected order.

Example 5 Let us consider again the correspondence (1). Using the notations of Definition 9, this correspondence is written $\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj } q_{11}$ (or $\text{event}(e_B(x_1, x_2, x_3, x_4)) \Rightarrow \text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj } q_{11}$), where $q_{11} = \text{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj } q_{1111}$, $q_{1111} = \text{event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj } q_{111111}$, and $q_{111111} = \text{event}(e_1(x_1, x_2, x_3))$. By Definition 9, this correspondence means that there exist functions ϕ_{11}, ϕ_{1111}, and ϕ_{111111} such that:

- For all τ, if the event $\texttt{event}(\sigma e_B(x_1, x_2, x_3, x_4))$ is executed at step τ for some σ, then $\phi_{11}(\tau)$, $\phi_{1111}(\tau)$, and $\phi_{111111}(\tau)$ are defined, and $\texttt{event}(\sigma e_3(x_1, x_2, x_3, x_4))$ is executed at step $\phi_{11}(\tau)$, $\texttt{event}(\sigma e_2(x_1, x_2, x_3, x_4))$ is executed at step $\phi_{1111}(\tau)$, and $\texttt{event}(\sigma e_1(x_1, x_2, x_3))$ is executed at step $\phi_{111111}(\tau)$. (Here, $\sigma' = \sigma$ since all variables of the correspondence occur in $\text{event}(e_B(x_1, x_2, x_3, x_4))$. Moreover, $j_{\overline{k}} = 1$ for all $\overline{k}$ and the non-empty sequences $\overline{k}$ are 1, 11, and 111, since all conjunctions and disjunctions have a single element. The sequences $\text{makejk}(\overline{k}, J)$ are then 11, 1111, and 111111.)

- The functions ϕ_{11}, ϕ_{1111}, and ϕ_{111111} are injective, so distinct executions of $e_B(x_1, x_2, x_3, x_4)$ correspond to distinct executions of $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$.

- When $\phi_{111111}(\tau)$ is defined, $\phi_{111111}(\tau) \leq \phi_{1111}(\tau) \leq \phi_{11}(\tau) \leq \tau$, so the events $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$ are executed in this order, before $e_B(x_1, x_2, x_3, x_4)$.

Similarly, general correspondences allow us to express that, if a protocol participant successfully terminates with honest interlocutors, then the expected messages of the protocol have been exchanged between the protocol participants, in the expected order. This notion is the formal counterpart of the notion of matching conversations initially introduced in the computational model by Bellare and Rogaway [11]. This notion of authentication is also used in [34].

We first focus on non-injective correspondences, and postpone the treatment of general correspondences to Section 7.2.

4 Automatic Verification: from Secrecy to Correspondences

Let us first summarize our analysis for secrecy. The clauses use two predicates: attacker and message, where $\text{attacker}(M)$ means that the attacker may have the message M and $\text{message}(M, M')$ means that the message M' may be sent on channel M. The clauses relate atoms that use these predicates as follows. A clause $\text{message}(M_1, M_1') \wedge \ldots \wedge \text{message}(M_n, M_n') \Rightarrow \text{message}(M, M')$ is generated when the process outputs M' on channel M after receiving M_1', $\ldots$, M_n' on channels M_1, $\ldots$, M_n respectively. A clause $\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)$ is generated when the attacker can compute M from M_1, $\ldots$, M_n. The clause $\text{message}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y)$ means that the attacker can listen on channel x when he has x, and the clause $\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{message}(x, y)$ means that the attacker can send any message y he has on any channel x he has. When $\text{attacker}(M)$ is derivable from the clauses the attacker *may* have M, that is, when $\text{attacker}(M)$ is not derivable from the clauses, we are sure that the attacker cannot have M, but the converse is not true, because the Horn clauses can be applied any number of times, which is not true in general for all actions of the process. Similarly, when $\text{message}(M, M')$ is derivable from the clauses, the message M' *may* be sent on channel M. Hence our analysis overapproximates the execution of actions.

Let us now consider that we want to prove a correspondence, for instance $\text{event}(e_1(x)) \rightsquigarrow \text{event}(e_2(x))$. In order to prove this correspondence, we can overapproximate the executions of event e_1: if we prove the correspondence with this overapproximation, it will also hold in the exact semantics. So we can easily extend our analysis for secrecy with an additional predicate event, such that $\text{event}(M)$ means that $\texttt{event}(M)$ may have been executed. We generate clauses $\text{message}(M_1, M_1') \wedge \ldots \wedge \text{message}(M_n, M_n') \Rightarrow \text{event}(M)$ when the process executes $\texttt{event}(M)$ after receiving M_1', $\ldots$, M_n' on channels M_1, $\ldots$, M_n respectively. However, such an overapproximation cannot be done for the event e_2: if we prove the correspondence after overapproximating the execution of e_2, we are not really sure that e_2 will be executed, so the correspondence may be wrong in the exact semantics. Therefore, we have to use a different method for treating e_2.

We use the following idea: we fix the exact set $\mathcal{E}$ of allowed events $e_2(M)$ and, in order to prove $\text{event}(e_1(x)) \rightsquigarrow \text{event}(e_2(x))$, we check that only events $e_1(M)$ for M such that $e_2(M) \in \mathcal{E}$ can be executed. If we prove this property for any value of $\mathcal{E}$, we have proved the desired correspondence. So we introduce a predicate m-event, such that $\text{m-event}(e_2(M))$ is true if and only if $e_2(M) \in \mathcal{E}$. We generate clauses $\text{message}(M_1, M_1') \wedge \ldots \wedge \text{message}(M_n, M_n') \wedge \text{m-event}(e_2(M_0)) \Rightarrow \text{message}(M, M')$ when the process outputs M' on channel M after executing the event $e_2(M_0)$ and receiving M_1', $\ldots$, M_n' on channels M_1, $\ldots$, M_n respectively. In other words, the output of M' on channel M can be executed only when $\text{m-event}(e_2(M_0))$ is true, that is, $e_2(M_0) \in \mathcal{E}$. (When the output of M' on channel M is under several events, the clause contains several m-event atoms in its hypothesis. We also have similar clauses with $\text{event}(e_1(M))$ instead of $\text{message}(M, M')$ when the event e_1 is executed after executing e_2 and receiving M_1', $\ldots$, M_n' on channels M_1, $\ldots$, M_n respectively.)

For instance, if the events $e_2(M_1)$ and $e_2(M_2)$ are executed in a certain trace of the protocol, we define $\mathcal{E} = \{e_2(M_1), e_2(M_2)\}$, so that $\text{m-event}(e_2(M_1))$ and $\text{m-event}(e_2(M_2))$ are true and all other m-event facts are false. Then we show that the only events e_1 that may be executed are $e_1(M_1)$ and $e_1(M_2)$. We prove a similar result for all values of $\mathcal{E}$, which proves the desired

correspondence.

In order to determine whether an atom is derivable from the clauses, we use a resolution-based algorithm. The resolution is performed for an unknown value of $\mathcal{E}$. So, basically, we keep m-event atoms without trying to evaluate them (which we cannot do since $\mathcal{E}$ is unknown). In the vocabulary of resolution, we never select m-event atoms. (We detail this point in Section 6.1.) Thus the obtained result holds for any value of $\mathcal{E}$, which allows us to prove correspondences. In order to prove the correspondence $\text{event}(e_1(x)) \rightsquigarrow \text{event}(e_2(x))$, we show that $\text{event}(e_1(M))$ is derivable only when $\text{m-event}(e_2(M))$ holds. We transform the initial set of clauses into a set of clauses that derives the same atoms. If, in the obtained set of clauses, all clauses that conclude $\text{event}(e_1(M))$ contain $\text{m-event}(e_2(M))$ in their hypotheses, then $\text{event}(e_1(M))$ is derivable only when $\text{m-event}(e_2(M))$ holds, so the desired correspondence holds.

We still have to solve one problem. For simplicity, we have considered that terms, which represent messages, are directly used in clauses. However, in order to represent nonces in our analysis for secrecy, we use a special encoding of names: a name a created by a restriction (νa) is represented by a function $a[M_1, \dots, M_n]$ of the messages $M_1, \dots, M_n$ received above the restriction, so that names created after receiving different messages are distinguished in the analysis (which is important for the precision of the analysis). However, this encoding still merges names created by the same restriction after receiving the same messages. For example, in the process $!c(x)(\nu a)$, the names created by (νa) are represented by $a[x]$, so several names created for the same value of x are merged. This merging is not acceptable for the verification of correspondences, because when we prove $\text{event}(e_1(x)) \rightsquigarrow \text{event}(e_2(x))$, we must make sure that x contains exactly the same names in $e_1(x)$ and in $e_2(x)$. In order to solve this problem, we label each replication with a *session identifier* i, which is an integer that takes a different value for each copy of the process generated by the replication. We add session identifiers as arguments to our encoding of names, which becomes $a[M_1, \dots, M_n, i_1, \dots, i_{n'}]$ where $i_1, \dots, i_{n'}$ are the session identifiers of the replications above the restriction (νa). For example, in the process $!c(x)(\nu a)$, the names created by (νa) are represented by $a[x, i]$. Each execution of the restriction is then associated with a distinct value of the session identifiers $i_1, \dots, i_{n'}$, so each name has a distinct encoding. We detail and formalize this encoding in Section 5.1.

5 From Processes to Horn Clauses

In this section, we first explain the instrumentation of processes with session identifiers. Next, we explain the translation of processes into Horn clauses.

5.1 Instrumented Processes

We consider a closed process P_0 representing the protocol we wish to check. We assume that the bound names of P_0 have been renamed so that they are pairwise distinct and distinct from names in $\mathit{Init} \cup \mathit{fn}(P_0)$ and in the correspondence to prove. We denote by Q a particular adversary; below, we prove the correspondence properties for any Q. Furthermore, we assume that, in the initial configuration $E_0, \{P_0, Q\}$, the names of E_0 not in $\mathit{Init} \cup \mathit{fn}(P_0)$ or in the correspondence to prove have been renamed to fresh names, and the bound names of Q have been renamed so that they are pairwise distinct and fresh. (These renamings do not change the satisfied correspondences, since $(\nu a)P$ and the renamed process $(\nu a')P\{a'/a\}$ reduce to the same configuration by (Red Res).) After encoding names, the terms are represented by *patterns* p (or "terms", but we prefer the word "patterns" in order to avoid confusion), which are generated by the following grammar:

$p ::=$	patterns
x, y, z, i	variable
$a[p_1, \dots, p_n, i_1, \dots, i_{n'}]$	name

$f(p_1, \ldots, p_n)$ constructor application

For each name a in P_0 we have a corresponding pattern construct $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$. We treat a as a function symbol, and write $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$ rather than $a(p_1, \ldots, p_n, i_1, \ldots, i_{n'})$ only to distinguish names from constructors. The symbol a in $a[\ldots]$ is called a *name function symbol*. If a is a free name, then its encoding is simply $a[\,]$. If a is bound by a restriction $(\nu a)P$ in P_0, then its encoding $a[\ldots]$ takes as argument session identifiers $i_1, \ldots, i_{n'}$, which can be constant session identifiers λ or variables i (taken in a set V_s disjoint from the set V_o of ordinary variables). There is one session identifier for each replication above the restriction (νa). The pattern $a[\ldots]$ may also take as argument patterns $p_1, \ldots, p_n$ containing the messages received by inputs above the restriction $(\nu a)P$ in the abstract syntax tree of P_0 and the result of destructor applications above the restriction $(\nu a)P$. (The precise definition is given below.)

In order to define formally the patterns associated with a name, we use a notion of instrumented processes. The syntax of instrumented processes is defined as follows:

- The replication $!P$ is labeled with a variable i in V_s: $!^i P$. The process $!^i P$ represents copies of P for a countable number of values of i. The variable i is a session identifier. It indicates which copy of P, that is, which session, is executed.
- The restriction $(\nu a)P$ is labeled with a restriction label ℓ: $(\nu a{:}\ell)P$, where ℓ is either $a[M_1, \ldots, M_n, i_1, \ldots, i_{n'}]$ for restrictions in honest processes or $b_0[a[i_1, \ldots, i_{n'}]]$ for restrictions in the adversary. The symbol b_0 is a special name function symbol, distinct from all other such symbols. Using a specific instrumentation for the adversary is helpful so that all names generated by the adversary are encoded by instances of $b_0[x]$. They are therefore easy to generate. This labeling of restrictions is similar to a Church-style typing: ℓ can be considered as the type of a. (This type is polymorphic since it can contain variables.)

The instrumented processes are then generated by the following grammar:

$P, Q ::=$ instrumented processes
  $!^i P$ replication
  $(\nu a : \ell)P$ restriction
  $\ldots$ (as in the standard calculus)

For instrumented processes, a semantic configuration $S, E, \mathcal{P}$ consists of a set S of session identifiers that have not yet been used by $\mathcal{P}$, an environment E that is a mapping from names to closed patterns of the form $a[\ldots]$, and a finite multiset of instrumented processes $\mathcal{P}$. The first semantic configuration uses any countable set of session identifiers S_0. The domain of E must always contain all free names of processes in $\mathcal{P}$, and the initial environment maps all names a to the pattern $a[\,]$. The semantic rules (Red Repl) and (Red Res) become:

$$S, E, \mathcal{P} \cup \{\, !^i P \,\} \rightarrow S \setminus \{\lambda\}, E, \mathcal{P} \cup \{\, P\{\lambda/i\}, !^i P \,\} \text{ where } \lambda \in S \qquad \text{(Red Repl)}$$

$$S, E, \mathcal{P} \cup \{\, (\nu a : \ell)P \,\} \rightarrow S, E[a' \mapsto E(\ell)\,], \mathcal{P} \cup \{\, P\{a'/a\} \,\} \text{ if } a' \notin \mathit{dom}(E) \qquad \text{(Red Res)}$$

where the mapping E is extended to all terms as a substitution by $E(f(M_1, \ldots, M_n)) = f(E(M_1), \ldots, E(M_n))$ and to restriction labels by $E(a[M_1, \ldots, M_n, i_1, \ldots, i_{n'}]) = a[E(M_1), \ldots, E(M_n), i_1, \ldots, i_{n'}]$ and $E(b_0[a[i_1, \ldots, i_{n'}]]) = b_0[a[i_1, \ldots, i_{n'}]]$, so that it maps terms and restriction labels to patterns. The rule (Red Repl) takes an unused constant session identifier λ in S, and creates a copy of P with session identifier λ. The rule (Red Res) creates a fresh name a', substitutes it for a in P, and adds to the environment E the mapping of a' to its encoding $E(\ell)$. Other semantic rules $E, \mathcal{P} \rightarrow E, \mathcal{P}'$ simply become $S, E, \mathcal{P} \rightarrow S, E, \mathcal{P}'$.

The instrumented process $P_0' = \mathrm{instr}(P_0)$ associated with the process P_0 is built from P_0 as follows:

- We label each replication $!P$ of P_0 with a distinct, fresh session identifier i, so that it becomes $!^i P$.
- We label each restriction (νa) of P_0 with $a[t, s]$, so that it becomes $(\nu a : a[t, s])$, where s is the sequence of session identifiers that label replications above (νa) in the abstract syntax tree of P'_0, in the order from top to bottom; t is the sequence of variables x that store received messages in inputs $M(x)$ above (νa) in P_0 and results of non-deterministic destructor applications *let* $x = g(\ldots)$ *in* P *else* Q above (νa) in P_0. (A destructor is said to be non-deterministic when it may return several different results for the same arguments. Adding the result of destructor applications to t is useful to improve precision, only for non-deterministic destructors. For deterministic destructors, the result of the destructor can be uniquely determined from the other elements of t, so the addition is useless. If we add the result of non-deterministic destructors to t, we can show that the relative completeness result of [1] still holds in the presence of non-deterministic destructors. This result shows that, for secrecy, the Horn clause approach is at least as precise as a large class of type systems.)

  Hence names are represented by functions $a[t, s]$ of the inputs and results of destructor applications in t and the session identifiers in s. In each trace of the process, at most one name corresponds to a given $a[t, s]$, since different copies of the restriction have different values of session identifiers in s. Therefore, different names are not merged by the verifier.

For the adversary, we use a slightly different instrumentation. We build the instrumented process $Q' = \text{instrAdv}(Q)$ as follows:

- We label each replication $!P$ of Q with a distinct, fresh session identifier i, so that it becomes $!^i P$.
- We label each restriction (νa) of Q with $b_0[a[s]]$, so that it becomes $(\nu a : b_0[a[s]])$, where s is the sequence of session identifiers that label replications above (νa) in Q'. (Including the session identifiers as arguments of nonces is necessary for soundness, as discussed in Section 4. Including the messages previously received as arguments of nonces is important for precision in the case of honest processes, in order to relate the nonces to these messages. It is however useless for the adversary: since we consider any *Init*-adversary Q, we have no definite information on the relation between nonces generated by the adversary and messages previously received by the adversary.)

Remark 2 By moving restrictions downwards in the syntax tree of the process (until the point at which the fresh name is used), one can add more arguments to the pattern that represents the fresh name, when the restriction is moved under an input, replication, or destructor application. Therefore, this transformation can make our analysis more precise. The tool can perform this transformation automatically.

Example 6 The instrumentation of the process of Section 2.3 yields:

$$\begin{aligned}
&P'_A(sk_A, pk_A, pk_B) = !^{i_A} c(x_pk_B).(\nu a : a[x_pk_B, i_A]) \ldots (\nu r_1 : r_1[x_pk_B, i_A]) \ldots \\
&\qquad c(m) \ldots (\nu r_3 : r_3[x_pk_B, m, i_A]]) \\
&P'_B(sk_B, pk_B, pk_A) = !^{i_B} c(m') \ldots (\nu b : b[m', i_B]) \ldots (\nu r_2 : r_2[m', i_B]) \ldots \\
&P' = (\nu sk_A : sk_A[\,])(\nu sk_B : sk_B[\,]) \ldots (P'_A(sk_A, pk_A, pk_B) \mid P'_B(sk_B, pk_B, pk_A))
\end{aligned}$$

The names created by the restriction (νa) will be represented by the pattern $a[x_pk_B, i_A]$, so we have a different pattern for each copy of the process, indexed by i_A, and the pattern also records the public key x_pk_B of the interlocutor of A. Similarly, the names created by the restriction (νb) will be represented by the pattern $b[m', i_B]$.

The semantics of instrumented processes allows exactly the same communications and events as the one of standard processes. More precisely, let $\mathcal{P}$ be a multiset of instrumented processes. We define $\mathrm{unInstr}(\mathcal{P})$ as the multiset of processes of $\mathcal{P}$ without the instrumentation. Thus we have:

Proposition 1 *If $E_0, \{P_0, Q\} \to^* E_1, \mathcal{P}_1$, then there exist E'_1 and $\mathcal{P}'_1$ such that for any S, countable set of session identifiers, there exists S' such that $S, \{a \mapsto a[\,] \mid a \in E_0\}, \{\mathrm{instr}(P_0), \mathrm{instrAdv}(Q)\} \to^* S', E'_1, \mathcal{P}'_1$, $dom(E'_1) = E_1$, $\mathrm{unInstr}(\mathcal{P}'_1) = \mathcal{P}_1$, and both traces execute the same events at the same steps and satisfy the same atoms.*

Conversely, if $S, \{a \mapsto a[\,] \mid a \in E_0\}, \{\mathrm{instr}(P_0), \mathrm{instrAdv}(Q)\} \to^* S', E'_1, \mathcal{P}'_1$, then $E_0, \{P_0, Q\} \to^* dom(E'_1), \mathrm{unInstr}(\mathcal{P}'_1)$, and both traces execute the same events at the same steps and satisfy the same atoms.

Proof This is an easy proof by induction on the length of the traces. The reduction rules applied in both traces are rules with the same name. □

We can define correspondences for instrumented processes. These correspondences and the clauses use *facts* defined by the following grammar:

$F ::=$	facts
$\quad \mathrm{attacker}(p)$	attacker knowledge
$\quad \mathrm{message}(p, p')$	message on a channel
$\quad \text{m-event}(p)$	must-event
$\quad \mathrm{event}(p)$	may-event

The fact $\mathrm{attacker}(p)$ means that the attacker may have p, and the fact $\mathrm{message}(p, p')$ means that the message p' may appear on channel p. The fact $\text{m-event}(p)$ means that $\mathtt{event}(M)$ must have been executed with M corresponding to p, and $\mathrm{event}(p)$ that $\mathtt{event}(M)$ may have been executed with M corresponding to p. We use the word "fact" to distinguish them from atoms $\mathrm{attacker}(M)$, $\mathrm{message}(M, M')$, and $\mathrm{event}(M)$. The correspondences do not use the fact $\text{m-event}(p)$, but the clauses use it.

The mapping E of a semantic configuration is extended to atoms by $E(\mathrm{attacker}(M)) = \mathrm{attacker}(E(M))$, $E(\mathrm{message}(M, M')) = \mathrm{message}(E(M), E(M'))$, and $E(\mathrm{event}(M)) = \mathrm{event}(E(M))$, so that it maps atoms to facts. We define that an instrumented trace $\mathcal{T}$ satisfies an atom α by naturally adapting Definition 2. When F is not $\text{m-event}(p)$, we say that an instrumented trace $\mathcal{T} = S_0, E_0, \mathcal{P}_0 \to^* S', E', \mathcal{P}'$ satisfies a fact F when there exists an atom α such that $\mathcal{T}$ satisfies α and $E'(\alpha) = F$. We also define that $\mathtt{event}(M)$ is executed at step τ in the instrumented trace $\mathcal{T}$ by naturally adapting Definition 6. We say that $\mathtt{event}(p)$ is executed at step τ in the instrumented trace $\mathcal{T} = S_0, E_0, \mathcal{P}_0 \to^* S', E', \mathcal{P}'$ when there exists a term M such that $\mathtt{event}(M)$ is executed at step τ in $\mathcal{T}$ and $E'(M) = p$.

Definition 10 Let P_0 be a closed process and $P'_0 = \mathrm{instr}(P_0)$. The instrumented process P'_0 satisfies the correspondence

$$F \Rightarrow \bigvee_{j=1}^{m} \left(F_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(p_{jk}) \right)$$

against *Init*-adversaries if and only if, for any *Init*-adversary Q, for any trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \to^* S', E', \mathcal{P}'$, with $Q' = \mathrm{instrAdv}(Q)$, $E_0(a) = a[\,]$ for all $a \in dom(E_0)$, and $fn(P'_0) \cup \mathit{Init} \subseteq dom(E_0)$, if $\mathcal{T}$ satisfies σF for some substitution σ, then there exist σ' and $j \in \{1, \ldots, m\}$ such that $\sigma' F_j = \sigma F$ and for all $k \in \{1, \ldots, l_j\}$, $\mathcal{T}$ satisfies $\mathrm{event}(\sigma' p_{jk})$.

A correspondence for instrumented processes implies a correspondence for standard processes, as shown by the following lemma, proved in Appendix A.

Lemma 1 *Let P_0 be a closed process and $P'_0 = \text{instr}(P_0)$. Let M_{jk} ($j \in \{1,\ldots,m\}$, $k \in \{1,\ldots,l_j\}$) be terms; let α and α_j ($j \in \{1,\ldots,m\}$) be atoms. Let p_{jk}, F, F_j be the patterns and facts obtained by replacing names a with patterns $a[\,]$ in the terms and atoms M_{jk}, α, α_j respectively. If P'_0 satisfies the correspondence*

$$F \Rightarrow \bigvee_{j=1}^{m} \left(F_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \text{event}(p_{jk}) \right)$$

against Init-adversaries then P_0 satisfies the correspondence

$$\alpha \Rightarrow \bigvee_{j=1}^{m} \left(\alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \text{event}(M_{jk}) \right)$$

against Init-adversaries.

For instrumented processes, we can specify properties referring to bound names of the process, which are represented by patterns. Such a specification is impossible in standard processes, because bound names can be renamed, so they cannot be referenced in terms in correspondences.

5.2 Generation of Horn Clauses

Given a closed process P_0 and a set of names *Init*, the protocol verifier first instruments P_0 to obtain $P'_0 = \text{instr}(P_0)$, then it builds a set of Horn clauses, representing the protocol in parallel with any *Init*-adversary. The clauses are of the form $F_1 \wedge \ldots \wedge F_n \Rightarrow F$, where $F_1, \ldots, F_n, F$ are facts. They comprise clauses for the attacker and clauses for the protocol, defined below. These clauses form the set $\mathcal{R}_{P'_0,Init}$. The predicate m-event is defined by a set of closed facts $\mathcal{F}_{\text{me}}$, such that m-event(p) is true if and only if m-event$(p) \in \mathcal{F}_{\text{me}}$. The facts in $\mathcal{F}_{\text{me}}$ do not belong to $\mathcal{R}_{P'_0,Init}$. The set $\mathcal{F}_{\text{me}}$ is the set of facts that corresponds to the set of allowed events $\mathcal{E}$, mentioned in Section 4.

5.2.1 Clauses for the Attacker

The clauses describing the attacker are almost the same as for the verification of secrecy in [1]. The only difference is that, here, the attacker is given an infinite set of fresh names $b_0[x]$, instead of only one fresh name $b_0[\,]$. Indeed, we cannot merge all fresh names created by the attacker, since we have to make sure that different terms are represented by different patterns for the verification of correspondences to be correctly implemented, as seen in Section 4. The abilities of the attacker are then represented by the following clauses:

For each $a \in Init$, attacker$(a[\,])$ (Init)

attacker$(b_0[x])$ (Rn)

For each public constructor f of arity n,
attacker$(x_1) \wedge \ldots \wedge$ attacker$(x_n) \Rightarrow$ attacker$(f(x_1, \ldots, x_n))$ (Rf)

For each public destructor g,
for each rewrite rule $g(M_1, \ldots, M_n) \rightarrow M$ in def(g), (Rg)
attacker$(M_1) \wedge \ldots \wedge$ attacker$(M_n) \Rightarrow$ attacker(M)

message$(x, y) \wedge$ attacker$(x) \Rightarrow$ attacker(y) (Rl)

attacker$(x) \wedge$ attacker$(y) \Rightarrow$ message(x, y) (Rs)

The clause (Init) represents the initial knowledge of the attacker. The clause (Rn) means that the attacker can generate an unbounded number of new names. The clauses (Rf) and (Rg)

mean that the attacker can apply all operations to all terms it has, (Rf) for constructors, (Rg) for destructors. For (Rg), notice that the rewrite rules in $\mathrm{def}(g)$ do not contain names and that terms without names are also patterns, so the clauses have the required format. Clause (Rl) means that the attacker can listen on all channels it has, and (Rs) that it can send all messages it has on all channels it has.

If $c \in \mathit{Init}$, we can replace all occurrences of $\mathrm{message}(c[\,], M)$ with $\mathrm{attacker}(M)$ in the clauses. Indeed, these facts are equivalent by the clauses (Rl) and (Rs).

5.2.2 Clauses for the Protocol

When a function ρ associates a pattern with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$.

The translation $[\![P]\!]\rho H$ of a process P is a set of clauses, where ρ is a function that associates a pattern with each name and variable, and H is a sequence of facts of the form $\mathrm{message}(p, p')$ or $\text{m-event}(p)$. The environment ρ maps each variable and name to its associated pattern representation. The sequence H keeps track of events that have been executed and of messages received by the process, since these may trigger other messages. The empty sequence is denoted by $\emptyset$; the concatenation of a fact F to the sequence H is denoted by $H \wedge F$. The pattern ρi is always a session identifier variable of V_s.

$$
\begin{aligned}
&[\![0]\!]\rho H = \emptyset\\
&[\![P \mid Q]\!]\rho H = [\![P]\!]\rho H \cup [\![Q]\!]\rho H\\
&[\![!^i P]\!]\rho H = [\![P]\!](\rho[i \mapsto i])H\\
&[\![(\nu a : a[M_1, \ldots, M_n, i_1, \ldots, i_{n'}])P]\!]\rho H =\\
&\quad [\![P]\!](\rho[a \mapsto a[\rho(M_1), \ldots, \rho(M_n), \rho(i_1), \ldots, \rho(i_{n'})]\,])H\\
&[\![M(x).P]\!]\rho H = [\![P]\!](\rho[x \mapsto x])(H \wedge \mathrm{message}(\rho(M), x))\\
&[\![\overline{M}\langle N\rangle.P]\!]\rho H = [\![P]\!]\rho H \cup \{H \Rightarrow \mathrm{message}(\rho(M), \rho(N))\}\\
&[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho H = \bigcup \{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma H)\\
&\quad \mid g(p'_1, \ldots, p'_n) \to p' \text{ is in } \mathrm{def}(g) \text{ and } (\sigma, \sigma') \text{ is a most general pair of}\\
&\quad \text{substitutions such that } \sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho H\\
&[\![\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q]\!]\rho H = [\![P]\!](\sigma\rho)(\sigma H) \cup [\![Q]\!]\rho H\\
&\quad \text{where } \sigma \text{ is the most general unifier of } \rho(M) \text{ and } \rho(N)\\
&[\![\mathtt{event}(M).P]\!]\rho H = [\![P]\!]\rho(H \wedge \text{m-event}(\rho(M))) \cup \{H \Rightarrow \mathrm{event}(\rho(M))\}
\end{aligned}
$$

The translation of a process is a set of Horn clauses that express that it may send certain messages or execute certain events. The clauses are similar to those of [1], except in the cases of replication, restriction, and the addition of events.

- The nil process does nothing, so its translation is empty.
- The clauses for the parallel composition of processes P and Q are the union of clauses for P and Q.
- The replication only inserts the new session identifier i in the environment ρ. It is otherwise ignored, because all Horn clauses are applicable arbitrarily many times.
- For the restriction, we replace the restricted name a in question with the pattern $a[\rho(M_1), \ldots, \rho(M_n), \rho(i_1), \ldots, \rho(i_{n'})]$. By definition of the instrumentation, this pattern contains the previous inputs, results of non-deterministic destructor applications, and session identifiers.

- The sequence H is extended in the translation of an input, with the input in question.
- The translation of an output adds a clause, meaning that the output is triggered when all conditions in H are true.
- The translation of a destructor application is the union of the clauses for the cases where the destructor succeeds (with an appropriate substitution) and where the destructor fails. For simplicity, we assume that the *else* branch of destructors may always be executed; this is sufficient in most cases, since the *else* branch is often empty or just sends an error message. We outline a more precise treatment in Section 9.2.
- The conditional *if* $M = N$ *then* P *else* Q is in fact equivalent to *let* $x = \mathit{equal}(M, N)$ *in* P *else* Q, where the destructor *equal* is defined by $\mathit{equal}(x, x) \to x$, so the translation of the conditional is a particular case of the destructor application. We give it explicitly since it is particularly simple.
- The translation of an event adds the hypothesis $\text{m-event}(\rho(M))$ to H, meaning that P can be executed only if the event has been executed first. Furthermore, it adds a clause, meaning that the event is triggered when all conditions in H are true.

Remark 3 Depending on the form of the correspondences we want to prove, we can sometimes simplify the clauses generated for events. Suppose that all arguments of events in the process and in correspondences are of the form $f(M_1, \ldots, M_n)$ for some function symbol f.

If, for a certain function symbol f, events $\text{event}(f(\ldots))$ occur only before $\rightsquigarrow$ in the desired correspondences, then it is easy to see in the following theorems that hypotheses of the form $\text{m-event}(f(\ldots))$ in clauses can be removed without changing the result, so the clauses generated by the event $\mathtt{event}(M)$ when M is of the form $f(\ldots)$ can be simplified into:

$$[\![\mathtt{event}(M).P]\!]\rho H = [\![P]\!]\rho H \cup \{H \Rightarrow \text{event}(\rho(M))\}$$

(Intuitively, since the events $\text{event}(f(\ldots))$ occur only before $\rightsquigarrow$ in the desired correspondences, we never prove that an event $\mathtt{event}(f(\ldots))$ has been executed, so the facts $\text{m-event}(f(\ldots))$ are useless.)

Similarly, if $\text{event}(f(\ldots))$ occurs only after $\rightsquigarrow$ in the desired correspondences, then clauses that conclude a fact of the form $\text{event}(f(\ldots))$ can be removed without changing the result, so the clauses generated by the event $\mathtt{event}(M)$ when M is of the form $f(\ldots)$ can be simplified into:

$$[\![\mathtt{event}(M).P]\!]\rho H = [\![P]\!]\rho(H \wedge \text{m-event}(\rho(M)))$$

(Intuitively, since the events $\text{event}(f(\ldots))$ occur only after $\rightsquigarrow$ in the desired correspondences, we never prove properties of the form "if $\mathtt{event}(f(\ldots))$ has been executed, then ...", so clauses that conclude $\text{event}(f(\ldots))$ are useless.)

This translation of the protocol into Horn clauses introduces approximations. The actions are considered as implicitly replicated, since the clauses can be applied any number of times. This approximation implies that the tool fails to prove protocols that first need to keep some value secret and later reveal it. For instance, consider the process $(\nu d)(\overline{d}\langle s\rangle.\overline{c}\langle d\rangle \mid d(x))$. This process preserves the secrecy of s, because s is output on the private channel d and received by the input on d, before the adversary gets to know d by the output of d on the public channel c. However, the Horn clause method cannot prove this property, because it treats this process like a variant with additional replications $(\nu d)(!\overline{d}\langle s\rangle.\overline{c}\langle d\rangle \mid !d(x))$, which does not preserve the secrecy of s. Similarly, the process $(\nu d)(\overline{d}\langle M\rangle \mid d(x).d(x).\mathtt{event}(e_1))$ never executes the event e_1, but the Horn clause method cannot prove this property because it treats this process like $(\nu d)(!\overline{d}\langle M\rangle \mid d(x).d(x).\mathtt{event}(e_1))$, which may execute e_1. The only exception to this implicit

replication of processes is the creation of new names: since session identifiers appear in patterns, the created name is precisely related to the session that creates it, so name creation cannot be unduly repeated inside the same session. Due to these approximations, our tool is not complete (it may produce false attacks) but, as we show below, it is sound (the security properties that it proves are always true).

5.2.3 Summary and Correctness

Let $\rho = \{a \mapsto a[\,] \mid a \in fn(P'_0)\}$. We define the clauses corresponding to the instrumented process P'_0 as:

$$\mathcal{R}_{P'_0,Init} = [\![P'_0]\!]\rho\emptyset \cup \{\text{attacker}(a[\,]) \mid a \in Init\} \cup \{(\text{Rn}), (\text{Rf}), (\text{Rg}), (\text{Rl}), (\text{Rs})\}$$

Example 7 The clauses for the process P of Section 2.3 are the clauses for the adversary, plus:

$$\text{attacker}(pk(sk_A[\,])) \tag{2}$$
$$\text{attacker}(pk(sk_B[\,])) \tag{3}$$
$$H_1 \Rightarrow \text{attacker}(pencrypt_p((a[x_pk_B, i_A], pk(sk_A[\,])), x_pk_B, r_1[x_pk_B, i_A])) \tag{4}$$
$$H_2 \Rightarrow \text{attacker}(pencrypt_p(x_b, x_pk_B, r_3[x_pk_B, p_2, i_A])) \tag{5}$$
$$H_3 \Rightarrow \text{event}(e_A(pk(sk_A[\,]), pk(sk_B[\,]), a[pk(sk_B[\,]), i_A], x_b)) \tag{6}$$
$$H_3 \Rightarrow \text{attacker}(sencrypt(sAa[\,], a[pk(sk_B[\,]), i_A])) \tag{7}$$
$$H_3 \Rightarrow \text{attacker}(sencrypt(sAb[\,], x_b)) \tag{8}$$

where $p_2 = pencrypt_p((a[x_pk_B, i_A], x_b, x_pk_B), pk(sk_A[\,]), x_r_2)$

$H_1 = \text{attacker}(x_pk_B) \wedge \text{m-event}(e_1(pk(sk_A[\,]), x_pk_B, a[x_pk_B, i_A]))$

$H_2 = H_1 \wedge \text{attacker}(p_2) \wedge \text{m-event}(e_3(pk(sk_A[\,]), x_pk_B, a[x_pk_B, i_A], x_b))$

$H_3 = H_2\{pk(sk_B[\,])/x_pk_B\}$

$$\begin{aligned}&\text{attacker}(p_1) \wedge \text{m-event}(e_2(x_pk_A, pk(sk_B[\,]), x_a, b[p_1, i_B]))\\ &\quad\Rightarrow \text{attacker}(pencrypt_p((x_a, b[p_1, i_B], pk(sk_B[\,])), x_pk_A, r_2[p_1, i_B]))\end{aligned} \tag{9}$$

where $p_1 = pencrypt_p((x_a, x_pk_A), pk(sk_B[\,]), x_r_1)$

$$H_4 \Rightarrow \text{event}(e_B(pk(sk_A[\,]), pk(sk_B[\,]), x_a, b[p'_1, i_B])) \tag{10}$$
$$H_4 \Rightarrow \text{attacker}(sencrypt(sBa[\,], x_a)) \tag{11}$$
$$H_4 \Rightarrow \text{attacker}(sencrypt(sBb[\,], b[p'_1, i_B])) \tag{12}$$

where $p'_1 = pencrypt_p((x_a, pk(sk_A[\,])), pk(sk_B[\,]), x_r_1)$

$H_4 = \text{attacker}(p'_1) \wedge \text{m-event}(e_2(pk(sk_A[\,]), pk(sk_B[\,]), x_a, b[p'_1, i_B])) \wedge \text{attacker}(pencrypt_p(b[p'_1, i_B], pk(sk_B[\,]), x_r_3))$

Clauses (2) and (3) correspond to the outputs in P; they mean that the adversary has the public keys of the participants. Clauses (4) and (5) correspond to the first two outputs in P_A. For example, (5) means that, if the attacker has x_pk_B and the second message of the protocol p_2 and the events $e_1(pk(sk_A[\,]), x_pk_B, a[x_pk_B, i_A])$ and $e_3(pk(sk_A[\,]), x_pk_B, a[x_pk_B, i_A], x_b)$ are allowed, then the attacker can get $pencrypt_p(x_b, x_pk_B, r_3[x_pk_B, p_2, i_A])$, because P_A sends this message after receiving x_pk_B and p_2 and executing the events e_1 and e_3. When furthermore $x_pk_B = pk(sk_B[\,])$, P_A executes event e_A and outputs the encryption of $sAa[\,]$ under $a[x_pk_B, i_A]$ and the encryption of $sBb[\,]$ under x_b. These event and outputs are taken into account by Clauses (6), (7), and (8) respectively. Similarly, Clauses (9), (11), and (12) correspond to the outputs in P_B and (10) to the event e_B. These clauses have been simplified using Remark 3, taking into account that e_1, e_2, and e_3 appear only on the right-hand side of $\rightsquigarrow$, and e_A and e_B only on the left-hand side of $\rightsquigarrow$ in the queries of Examples 1, 2, and 3.

Theorem 1 (Correctness of the clauses) *Let P_0 be a closed process and Q be an Init-adversary. Let $P'_0 = \mathrm{instr}(P_0)$ and $Q' = \mathrm{instrAdv}(Q)$. Consider a trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$, with $fn(P'_0) \cup Init \subseteq dom(E_0)$ and $E_0(a) = a[\,]$ for all $a \in dom(E_0)$. Assume that, if $\mathcal{T}$ satisfies $\mathrm{event}(p)$, then $\text{m-event}(p) \in \mathcal{F}_{\mathrm{me}}$. Finally, assume that $\mathcal{T}$ satisfies F. Then F is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\mathrm{me}}$.*

This result shows that, if the only executed events are those allowed in $\mathcal{F}_{\mathrm{me}}$ and a fact F is satisfied, then F is derivable from the clauses. It is proved in Appendix B. Using a technique similar to that of [1], its proof relies on a type system to express the soundness of the clauses on P'_0, and on the subject reduction of this type system to show that soundness of the clauses is preserved during all executions of the process.

6 Solving Algorithm

We first describe a basic solving algorithm without optimizations. Next, we list the optimizations that we use in our implementation, and we prove the correctness of the algorithm. The termination of the algorithm is discussed in Section 8.

6.1 The Basic Algorithm

To apply the previous results, we have to determine whether a fact is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\mathrm{me}}$. This may be undecidable, but in practice there exist algorithms that terminate on numerous examples of protocols. In particular, we can use variants of resolution algorithms, such as the algorithms described in [13, 14, 20, 69]. The algorithm that we describe here is the one of [14], extended with a second phase to determine derivability of any query. It also corresponds to the extension to m-event facts of the algorithm of [20].

We first define resolution: when the conclusion of a clause R unifies with an hypothesis F_0 of a clause R', we can infer a new clause $R \circ_{F_0} R'$, that corresponds to applying R and R' one after the other. Formally, this is defined as follows:

Definition 11 Let $R = H \Rightarrow C$ and $R' = H' \Rightarrow C'$ be two clauses. Assume that there exists $F_0 \in H'$ such that C and F_0 are unifiable, and σ is the most general unifier of C and F_0. In this case, we define $R \circ_{F_0} R' = \sigma(H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma C'$.

An important idea to obtain an efficient solving algorithm is to specify conditions that limit the application of resolution, while keeping completeness. The conditions that we use correspond to resolution with free selection [9, 35, 55]: a selection function chooses selected facts in each clause, and resolution is performed only on selected facts, that is, the clause $R \circ_{F_0} R'$ is generated only when the conclusion is selected in R and F_0 is selected in R'.

Definition 12 We denote by sel a selection function, that is, a function from clauses to sets of facts, such that $\mathsf{sel}(H \Rightarrow C) \subseteq H$. If $F \in \mathsf{sel}(R)$, we say that F is selected in R. If $\mathsf{sel}(R) = \emptyset$, we say that no hypothesis is selected in R, or that the conclusion of the clause is selected.

The choice of the selection function can change dramatically the speed of the algorithm. Since the algorithm combines clauses by resolution only when the facts unified in the resolution are selected, we will choose the selection function to reduce the number of possible unifications between selected facts. Having several selected facts slows down the algorithm, because it has more choices of resolutions to perform, therefore we will select at most one fact in each clause. In the case of protocols, facts of the form $\mathrm{attacker}(x)$, with x variable, can be unified will all facts of the form $\mathrm{attacker}(p)$. Therefore we should avoid selecting them. The m-event facts must never be selected since they are not defined by known clauses.

First phase: saturation

$\mathsf{saturate}(\mathcal{R}_0) =$

1. $\mathcal{R} \leftarrow \emptyset$.
   For each $R \in \mathcal{R}_0$, $\mathcal{R} \leftarrow \mathit{elim}(\mathit{simplify}(R) \cup \mathcal{R})$.
2. Repeat until a fixpoint is reached
   for each $R \in \mathcal{R}$ such that $\mathsf{sel}(R) = \emptyset$,
   for each $R' \in \mathcal{R}$, for each $F_0 \in \mathsf{sel}(R')$ such that $R \circ_{F_0} R'$ is defined,
   $\mathcal{R} \leftarrow \mathit{elim}(\mathit{simplify}(R \circ_{F_0} R') \cup \mathcal{R})$.
3. Return $\{R \in \mathcal{R} \mid \mathsf{sel}(R) = \emptyset\}$.

Second phase: backwards depth-first search

$$\mathsf{deriv}(R, \mathcal{R}, \mathcal{R}_1) = \begin{cases} \emptyset & \text{if } \exists R' \in \mathcal{R}, R' \sqsupseteq R \\ \{R\} & \text{otherwise, if } \mathsf{sel}(R) = \emptyset \\ \bigcup \{\mathsf{deriv}(\mathit{simplify}'(R' \circ_{F_0} R), \{R\} \cup \mathcal{R}, \mathcal{R}_1) \mid R' \in \mathcal{R}_1, & \\ \quad F_0 \in \mathsf{sel}(R) \text{ such that } R' \circ_{F_0} R \text{ is defined} \} & \text{otherwise} \end{cases}$$

$\mathsf{derivable}(F, \mathcal{R}_1) = \mathsf{deriv}(F \Rightarrow F, \emptyset, \mathcal{R}_1)$

Figure 4: Solving algorithm

Definition 13 We say that a fact F is *unselectable* when $F = \text{attacker}(x)$ for some variable x or $F = \text{m-event}(p)$ for some pattern p. Otherwise, we say that F is *selectable*.

We require that the selection function never selects unselectable hypotheses and that $\mathsf{sel}(H \Rightarrow \text{attacker}(x)) \neq \emptyset$ when H contains a selectable fact.

A basic selection function for security protocols is then

$$\mathsf{sel}_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if } \forall F \in H,\ F \text{ is unselectable} \\ \{F_0\} & \text{where } F_0 \in H \text{ and } F_0 \text{ is selectable, otherwise} \end{cases}$$

In the implementation, the hypotheses are represented by a list, and the selected fact is the first selectable element of the list of hypotheses.

The solving algorithm works in two phases, summarized in Figure 4. The first phase, saturate, transforms the set of clauses into an equivalent but simpler one. The second phase, derivable, uses a depth-first search to determine whether a fact can be inferred or not from the clauses.

The first phase contains 3 steps.

- The first step inserts in $\mathcal{R}$ the initial clauses representing the protocol and the attacker (clauses that are in $\mathcal{R}_0$), after simplification by *simplify* (defined below in Section 6.2) and elimination of subsumed clauses by *elim*. We say that $H_1 \Rightarrow C_1$ subsumes $H_2 \Rightarrow C_2$, and we write $(H_1 \Rightarrow C_1) \sqsupseteq (H_2 \Rightarrow C_2)$, when there exists a substitution σ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$. (H_1 and H_2 are multisets, and we use here multiset inclusion.) If R' subsumes R, and R and R' are in $\mathcal{R}$, then R is removed by $\mathit{elim}(\mathcal{R})$.

- The second step is a fixpoint iteration that adds clauses created by resolution. The composition of clauses R and R' is added only if no hypothesis is selected in R, and the hypothesis F_0 of R' that we unify is selected. When a clause is created by resolution, it is added to the set of clauses $\mathcal{R}$ after simplification. Subsumed clauses are eliminated from $\mathcal{R}$.

- At last, the third step returns the set of clauses of $\mathcal{R}$ with no selected hypothesis.

Basically, $\mathsf{saturate}$ preserves derivability: F is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\rm me}$ if and only if it is derivable from $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\rm me}$. A formal statement of this result is given in Lemma 2 below.

The second phase searches the facts that can be inferred from $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$. This is simply a backward depth-first search. The call $\mathsf{derivable}(F, \mathcal{R}_1)$ returns a set of clauses $R = H \Rightarrow C$ with empty selection, such that R can be obtained by resolution from $\mathcal{R}_1$, C is an instance of F, and all instances of F derivable from $\mathcal{R}_1$ can be derived by using as last clause a clause of $\mathsf{derivable}(F, \mathcal{R}_1)$. (Formally, if F' is an instance of F derivable from $\mathcal{R}_1$, then there are a clause $H \Rightarrow C \in \mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution σ such that $F' = \sigma C$ and σH is derivable from $\mathcal{R}_1$.)

The search itself is performed by $\mathsf{deriv}(R, \mathcal{R}, \mathcal{R}_1)$. The function deriv starts with $R = F \Rightarrow F$ and transforms the hypothesis of R by using a clause R' of $\mathcal{R}_1$ to derive an element F_0 of the hypothesis of R. So R is replaced with $R' \circ_{F_0} R$ (third case of the definition of deriv). The fact F_0 is chosen using the selection function sel. The obtained clause $R' \circ_{F_0} R$ is then simplified by the function *simplify*$'$ defined in Section 6.2. (Hence deriv derives the hypothesis of R using a backward depth-first search. At each step, the clause R can be obtained by resolution from clauses of $\mathcal{R}_1$, and R concludes an instance of F.) The set $\mathcal{R}$ is the set of clauses that we have already seen during the search. Initially, $\mathcal{R}$ is empty, and the clause R is added to $\mathcal{R}$ in the third case of the definition of deriv.

The transformation of R described above is repeated until one of the following two conditions is satisfied:

- R is subsumed by a clause in $\mathcal{R}$: we are in a cycle; we are looking for instances of facts that we have already looked for (first case of the definition of deriv);
- $\mathsf{sel}(R)$ is empty: we have obtained a suitable clause R and we return it (second case of the definition of deriv).

6.2 Simplification Steps

Before adding a clause to the clause base, it is first simplified using the following functions. Some of them are standard, such as the elimination of tautologies and of duplicate hypotheses; others are specific to protocols. The simplification functions take as input a clause or a set of clauses and return a set of clauses.

Decomposition of Data Constructors A data constructor is a constructor f of arity n that comes with associated destructors g_i for $i \in \{1, \dots, n\}$ defined by $g_i(f(x_1, \dots, x_n)) \to x_i$. Data constructors are typically used for representing data structures. Tuples are examples of data constructors. For each data constructor f, the following clauses are generated:

$$\mathrm{attacker}(x_1) \wedge \dots \wedge \mathrm{attacker}(x_n) \Rightarrow \mathrm{attacker}(f(x_1, \dots, x_n)) \qquad \text{(Rf)}$$

$$\mathrm{attacker}(f(x_1, \dots, x_n)) \Rightarrow \mathrm{attacker}(x_i) \qquad \text{(Rg)}$$

Therefore, $\mathrm{attacker}(f(p_1, \dots, p_n))$ is derivable if and only if $\forall i \in \{1, \dots, n\}$, $\mathrm{attacker}(p_i)$ is derivable. So the function *decomp* transforms clauses as follows. When a fact of the form $\mathrm{attacker}(f(p_1, \dots, p_n))$ is met, it is replaced with $\mathrm{attacker}(p_1) \wedge \dots \wedge \mathrm{attacker}(p_n)$. If this replacement is done in the conclusion of a clause $H \Rightarrow \mathrm{attacker}(f(p_1, \dots, p_n))$, n clauses are created: $H \Rightarrow \mathrm{attacker}(p_i)$ for each $i \in \{1, \dots, n\}$. This replacement is of course done recursively: if p_i itself is a data constructor application, it is replaced again. The function *decomphyp* performs this decomposition only in the hypothesis of clauses. The functions *decomp* and *decomphyp* leave the clauses (Rf) and (Rg) for data constructors unchanged. (When $\mathrm{attacker}(x)$ cannot be selected, the clauses (Rf) and (Rg) for data constructors are in fact not necessary, because they generate only tautologies during resolution. However, when $\mathrm{attacker}(x)$ can be selected, which cannot be excluded in extensions such as the one presented in Section 9.3, these clauses may become necessary for soundness.)

$\mathsf{solve}_{P_0',Init}(F) =$

1. Let $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_{P_0',Init})$.
2. For each $F' \in \mathcal{F}_{\mathrm{not}}$, if $\mathsf{derivable}(F', \mathcal{R}_1) \neq \emptyset$, then terminate with error.
3. Return $\mathsf{derivable}(F, \mathcal{R}_1)$.

Figure 5: Summary of the solving algorithm

Elimination of Tautologies The function *elimtaut* removes clauses whose conclusion is already in the hypotheses, since such clauses do not generate new facts.

Elimination of Duplicate Hypotheses The function *elimdup* eliminates duplicate hypotheses of clauses.

Elimination of Useless $\mathrm{attacker}(x)$ **Hypotheses** If a clause $H \Rightarrow C$ contains in its hypotheses $\mathrm{attacker}(x)$, where x is a variable that does not appear elsewhere in the clause, the hypothesis $\mathrm{attacker}(x)$ is removed by the function *elimattx*. Indeed, the attacker always has at least one message, so $\mathrm{attacker}(x)$ is always satisfied.

Secrecy Assumptions When the user knows that a fact F will not be derivable, he can tell it to the verifier. (When this fact is of the form $\mathrm{attacker}(p)$, the user tells that p remains secret; that is why we use the name "secrecy assumptions".) Let $\mathcal{F}_{\mathrm{not}}$ be a set of facts, for which the user claims that no instance of these facts is derivable. The function *elimnot* removes all clauses that have an instance of a fact in $\mathcal{F}_{\mathrm{not}}$ in their hypotheses. As shown in Figure 5, at the end of the saturation, the solving algorithm checks that the facts in $\mathcal{F}_{\mathrm{not}}$ are indeed underivable from the obtained clauses. If this condition is satisfied, $\mathsf{solve}_{P_0',Init}(F)$ returns clauses that conclude instances of F. Otherwise, the user has given erroneous information, so an error message is displayed. Even when the user gives erroneous secrecy assumptions, the verifier never wrongly claims that a protocol is secure.

Mentioning such underivable facts prunes the search space, by removing useless clauses. This speeds up the search process. In most cases, the secret keys of the principals cannot be known by the attacker, so examples of underivable facts are $\mathrm{attacker}(sk_A[\,])$ and $\mathrm{attacker}(sk_B[\,])$.

Elimination of Redundant Hypotheses When a clause is of the form $H \wedge H' \Rightarrow C$, and there exists σ such that $\sigma H \subseteq H'$ and σ does not change the variables of H' and C, then the clause is replaced with $H' \Rightarrow C$ by the function *elimredundanthyp*. These clauses are semantically equivalent: obviously, $H' \Rightarrow C$ subsumes $H \wedge H' \Rightarrow C$; conversely, if a fact can be derived by an instance $\sigma' H' \Rightarrow \sigma' C$ of $H' \Rightarrow C$, then it can also be derived by the instance $\sigma'\sigma H \wedge \sigma' H' \Rightarrow \sigma' C$ of $H \wedge H' \Rightarrow C$, since the elements of $\sigma'\sigma H$ can be derived because they are in $\sigma' H'$.

This replacement is especially useful when H contains m-event facts. Otherwise, the elements of H could be selected and transformed by resolution, until they are of the form $\mathrm{attacker}(x)$, in which case they are removed by *elimattx* if $\sigma x \neq x$ (because x does not occur in H' and C since σ does not change the variables of H' and C) or by *elimdup* if $\sigma x = x$ (because $\mathrm{attacker}(x) = \sigma\mathrm{attacker}(x) \in \sigma H \subseteq H'$). In contrast, m-event facts remain forever, because they are unselectable. Depending on user settings, this replacement can be applied for all H, applied only when H contains a m-event fact, or switched off, since testing this property takes time and slows down small examples. On the other hand, on big examples, such as some of those generated by TulaFale [12] for verifying Web services, this technique can yield important speedups.

Putting All Simplifications Together The function *simplify* groups all these simplifications. We define $simplify = elimattx \circ elimtaut \circ elimnot \circ elimredundanthyp \circ elimdup \circ decomp$. In this definition, the simplifications are ordered in such a way that $simplify \circ simplify = simplify$, so it is not necessary to repeat the simplification.

Similarly, $simplify' = elimattx \circ elimnot \circ elimredundanthyp \circ elimdup \circ decomphyp$. In $simplify'$, we use *decomphyp* instead of *decomp*, because the conclusion of the considered clause is the fact we want to derive, so it must not be modified.

6.3 Soundness

The following lemmas show the correctness of saturate and derivable (Figure 4). Proofs can be found in Appendix C. Intuitively, the correctness of saturate expresses that saturation preserves derivability, provided the secrecy assumptions are satisfied.

Lemma 2 (Correctness of saturate) *Let F be a closed fact. If, for all $F' \in \mathcal{F}_{\text{not}}$, no instance of F' is derivable from* $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$*, then F is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\text{me}}$ if and only if F is derivable from* $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$.

This result is proved by transforming a derivation of F from $\mathcal{R}_0 \cup \mathcal{F}_{\text{me}}$ into a derivation of F (or a fact in $\mathcal{F}_{\text{not}}$) from $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$. Basically, when the derivation contains a clause R' with $\mathsf{sel}(R') \neq \emptyset$, we replace in this derivation two clauses R, with $\mathsf{sel}(R) = \emptyset$, and R' that have been combined by resolution during the execution of saturate with a single clause $R \circ_{F_0} R'$. This replacement decreases the number of clauses in the derivation, so it terminates, and, upon termination, all clauses of the obtained derivation satisfy $\mathsf{sel}(R') = \emptyset$ so they are in $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$.

Intuitively, the correctness of derivable expresses that if F', instance of F, is derivable, then F' is derivable from $\mathcal{R}_1$ by a derivation in which the clause that concludes F' is in $\mathsf{derivable}(F, \mathcal{R}_1)$, provided the secrecy assumptions are satisfied.

Lemma 3 (Correctness of derivable) *Let F' be a closed instance of F. If, for all $F'' \in \mathcal{F}_{\text{not}}$,* $\mathsf{derivable}(F'', \mathcal{R}_1) = \emptyset$*, then F' is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$ if and only if there exist a clause $H \Rightarrow C$ in* $\mathsf{derivable}(F, \mathcal{R}_1)$ *and a substitution σ such that $\sigma C = F'$ and all elements of σH are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$.*

Basically, this result is proved by transforming a derivation of F' from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$ into a derivation of F' (or a fact in $\mathcal{F}_{\text{not}}$) whose last clause (the one that concludes F') is $H \Rightarrow C$ and whose other clauses are still in $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. The transformation relies on the replacement of clauses combined by resolution during the execution of derivable.

It is important to apply saturate before derivable, so that all clauses in $\mathcal{R}_1$ have no selected hypothesis. Then the conclusion of these clauses is in general not attacker(x) (with the simplifications of Section 6.2 and the selection function sel_0, it is never attacker(x)), so that we avoid unifying with attacker(x).

Finally, the following theorem shows the correctness of $\mathsf{solve}_{P'_0,Init}$ (Figure 5). Below, when we require that $\mathsf{solve}_{P'_0,Init}(F)$ has a certain value, we also implicitly require that $\mathsf{solve}_{P'_0,Init}(F)$ does not terminate with error. Intuitively, if an instance F' of F is satisfied by a trace $\mathcal{T}$, then F' is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\text{me}}$, so, by the soundness of the solving algorithm, it is derivable by a derivation whose last clause is in $\mathsf{solve}_{P'_0,Init}(F)$. Then there must exist a clause $H \Rightarrow C \in \mathsf{solve}_{P'_0,Init}(F)$ that can be used to derive F', so $F' = \sigma C$ and the hypothesis σH is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\text{me}}$. In particular, the events in σH are satisfied, that is, are in $\mathcal{F}_{\text{me}}$, so these events have been executed in the trace $\mathcal{T}$. Theorem 2 below states this result formally. It is proved by combining Lemmas 2 and 3, and Theorem 1.

Theorem 2 (Main theorem) *Let P_0 be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let Q be an Init-adversary and $Q' = \mathrm{instrAdv}(Q)$.*

Consider a trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', P'$, with $fn(P'_0) \cup Init \subseteq dom(E_0)$ and $E_0(a) = a[\,]$ for all $a \in dom(E_0)$.

If $\mathcal{T}$ satisfies an instance F' of F, then there exist a clause $H \Rightarrow C \in \mathsf{solve}_{P'_0,Init}(F)$ and a substitution σ such that $F' = \sigma C$ and, for all $\text{m-event}(p)$ in σH, $\mathcal{T}$ satisfies $\text{event}(p)$.

Proof Since for all $F'' \in \mathcal{F}_{\text{not}}$, $\mathsf{derivable}(F'', \mathcal{R}_1) = \emptyset$, by Lemma 3, no instance of F'' is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}} = \mathsf{saturate}(\mathcal{R}_{P'_0,Init}) \cup \mathcal{F}_{\text{me}}$. This allows us to apply Lemma 2.

Let $\mathcal{F}_{\text{me}} = \{\text{m-event}(p') \mid \mathcal{T} \text{ satisfies } \text{event}(p')\}$. By Theorem 1, since $\mathcal{T}$ satisfies F', F' is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\text{me}}$. By Lemma 2, F' is derivable from $\mathsf{saturate}(\mathcal{R}_{P'_0,Init}) \cup \mathcal{F}_{\text{me}} = \mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. By Lemma 3, there exist a clause $R = H \Rightarrow C$ in $\mathsf{solve}_{P'_0,Init}(F) = \mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution σ such that $\sigma C = F'$ and all elements of σH are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. For all $\text{m-event}(p)$ in σH, $\text{m-event}(p)$ is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. Since no clause in $\mathcal{R}_1$ has a conclusion of the form $\text{m-event}(p')$, $\text{m-event}(p) \in \mathcal{F}_{\text{me}}$. Given the choice of $\mathcal{F}_{\text{me}}$, this means that $\mathcal{T}$ satisfies $\text{event}(p)$. □

Theorem 2 is our main correctness result: it allows one to show that some events must have been executed. The correctness of the analysis for correspondences follows from this theorem.

Example 8 For the process P of Section 2.3, $Init = \{c\}$, and $P' = \mathrm{instr}(P)$, our tool shows that

$$\begin{aligned}\mathsf{solve}_{P',Init}(\text{event}(e_B(x_1,x_2,x_3,x_4))) = \{&\text{m-event}(e_1(pk_A, pk_B, p_a)) \wedge \\ &\text{m-event}(e_2(pk_A, pk_B, p_a, p_b)) \wedge \\ &\text{m-event}(e_3(pk_A, pk_B, p_a, p_b)) \\ &\Rightarrow \text{event}(e_B(pk_A, pk_B, p_a, p_b))\}\end{aligned}$$

$$\begin{aligned}\text{where } & pk_A = pk(sk_A[\,]),\ pk_B = pk(sk_B[\,]),\ p_a = a[pk_B, i_A] \\ & p_b = b[pencrypt_p((p_a, pk_A), pk_B, r_1[pk_B, i_A]), i_B]\end{aligned}$$

By Theorem 2, if $\mathcal{T}$ satisfies $\text{event}(e_B(p_1,p_2,p_3,p_4))$, this event is an instance of $\text{event}(e_B(x_1, x_2,x_3,x_4))$, so, given the value of $\mathsf{solve}_{P',Init}(\text{event}(e_B(x_1,x_2,x_3,x_4)))$, there exists σ such that $\text{event}(e_B(p_1,p_2,p_3,p_4)) = \sigma\text{event}(e_B(pk_A, pk_B, p_a, p_b))$ and $\mathcal{T}$ satisfies

$$\begin{aligned}\text{event}(\sigma e_1(pk_A, pk_B, p_a)) &= \text{event}(e_1(p_1,p_2,p_3)) \\ \text{event}(\sigma e_2(pk_A, pk_B, p_a, p_b)) &= \text{event}(e_2(p_1,p_2,p_3,p_4)) \\ \text{event}(\sigma e_3(pk_A, pk_B, p_a, p_b)) &= \text{event}(e_3(p_1,p_2,p_3,p_4))\end{aligned}$$

Therefore, if `event`$(e_B(M_1,M_2,M_3,M_4))$ has been executed, then `event`$(e_1(M_1,M_2,M_3))$, `event`$(e_2(M_1,M_2,M_3,M_4))$, and `event`$(e_3(M_1,M_2,M_3,M_4))$ have been executed.

7 Application to Correspondences

7.1 Non-injective Correspondences

Correspondences for instrumented processes can be checked as shown by the following theorem:

Theorem 3 *Let P_0 be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let p_{jk} ($j \in \{1,\dots,m\}$, $k \in \{1,\dots,l_j\}$) be patterns; let F and F_j ($j \in \{1,\dots,m\}$) be facts. Assume that for all $R \in \mathsf{solve}_{P'_0,Init}(F)$, there exist $j \in \{1,\dots,m\}$, σ', and H such that $R = H \wedge \text{m-event}(\sigma' p_{j1}) \wedge \ldots \wedge \text{m-event}(\sigma' p_{jl_j}) \Rightarrow \sigma' F_j$.*

Then P'_0 satisfies the correspondence $F \Rightarrow \bigvee_{j=1}^{m} \left(F_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \text{event}(p_{jk})\right)$ against Init-adversaries.

Proof Let Q be an *Init*-adversary and $Q' = \mathrm{instrAdv}(Q)$. Consider a trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$, with $fn(P'_0) \cup Init \subseteq dom(E_0)$ and $E_0(a) = a[\,]$ for all $a \in dom(E_0)$. Assume that $\mathcal{T}$ satisfies σF. By Theorem 2, there exist $R = H' \Rightarrow C' \in \mathsf{solve}_{P'_0,Init}(F)$ and σ'' such that $\sigma F = \sigma'' C'$ and for all m-event(p) in $\sigma'' H'$, $\mathcal{T}$ satisfies event(p). All clauses R in $\mathsf{solve}_{P'_0,Init}(F)$ are of the form $H \wedge \text{m-event}(\sigma' p_{j1}) \wedge \ldots \wedge \text{m-event}(\sigma' p_{jl_j}) \Rightarrow \sigma' F_j$ for some j and σ'. So, there exist j and σ' such that for all $k \in \{1, \ldots, l_j\}$, m-event$(\sigma' p_{jk}) \in H'$ and $C' = \sigma' F_j$. Hence $\sigma F = \sigma'' C' = \sigma'' \sigma' F_j$ and for all $k \in \{1, \ldots, l_j\}$, m-event$(\sigma'' \sigma' p_{jk}) \in \sigma'' H'$, so $\mathcal{T}$ satisfies event$(\sigma'' \sigma' p_{jk})$, so we have the result. □

From this theorem and Lemma 1, we obtain correspondences for standard processes.

Theorem 4 *Let P_0 be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let M_{jk} ($j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, l_j\}$) be terms; let α and α_j ($j \in \{1, \ldots, m\}$) be atoms. Let p_{jk}, F, F_j be the patterns and facts obtained by replacing names a with patterns $a[\,]$ in the terms and atoms M_{jk}, α, α_j respectively. Assume that, for all clauses R in $\mathsf{solve}_{P'_0,Init}(F)$, there exist $j \in \{1, \ldots, m\}$, σ', and H such that $R = H \wedge \text{m-event}(\sigma' p_{j1}) \wedge \ldots \wedge \text{m-event}(\sigma' p_{jl_j}) \Rightarrow \sigma' F_j$.*

Then P_0 satisfies the correspondence $\alpha \Rightarrow \bigvee_{j=1}^{m} \left(\alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk}) \right)$ against Init-adversaries.

Example 9 For the process P of Section 2.3, *Init* $= \{c\}$, and $P' = \mathrm{instr}(P)$, the value of $\mathsf{solve}_{P',Init}(\mathrm{event}(e_B(x_1, x_2, x_3, x_4)))$ given in Example 8 shows that P satisfies the correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_1(x_1, x_2, x_3)) \wedge \mathrm{event}(e_2(x_1, x_2, x_3, x_4)) \wedge \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ against *Init*-adversaries.

As particular cases of correspondences, we can show secrecy and non-injective agreement:

Corollary 1 (Secrecy) *Let P_0 be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let N be a term. Let p be the pattern obtained by replacing names a with patterns $a[\,]$ in the term N. Assume that $\mathsf{solve}_{P'_0,Init}(\mathrm{attacker}(p)) = \emptyset$. Then P_0 preserves the secrecy of all instances of N from Init.*

Intuitively, if no instance of attacker(p) is derivable from the clauses representing the protocol, then the adversary cannot have an instance of the term N corresponding to p.

Example 10 For the process P of Section 2.3, *Init* $= \{c\}$, and $P' = \mathrm{instr}(P)$, our tool shows that $\mathsf{solve}_{P',Init}(\mathrm{attacker}(sAa[\,])) = \emptyset$. So P preserves the secrecy of sAa from *Init*. The situation is similar for sAb, sBa, and sBb.

Corollary 2 (Non-injective agreement) *Let P_0 be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Assume that, for each $R \in \mathsf{solve}_{P'_0,Init}(\mathrm{event}(e(x_1, \ldots, x_n)))$ such that $R = H \Rightarrow \mathrm{event}(e(p_1, \ldots, p_n))$, we have $\text{m-event}(e'(p_1, \ldots, p_n)) \in H$. Then P_0 satisfies the correspondence $\mathrm{event}(e(x_1, \ldots, x_n)) \rightsquigarrow \mathrm{event}(e'(x_1, \ldots, x_n))$ against Init-adversaries.*

Intuitively, the condition means that, if $\mathrm{event}(e(p_1, \ldots, p_n))$ can be derived, $\text{m-event}(e'(p_1, \ldots, p_n))$ occurs in the hypotheses. Then the theorem says that, if $\mathtt{event}(e(M_1, \ldots, M_n))$ has been executed, then $\mathtt{event}(e'(M_1, \ldots, M_n))$ has been executed.

Example 11 For the process P of Section 2.3, *Init* $= \{c\}$, and $P' = \mathrm{instr}(P)$, the value of $\mathsf{solve}_{P',Init}(\mathrm{event}(e_B(x_1, x_2, x_3, x_4)))$ given in Example 8 also shows that P satisfies the correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ against *Init*-adversaries. The tool shows in a similar way that P satisfies the correspondence $\mathrm{event}(e_A(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_2(x_1, x_2, x_3, x_4))$ against *Init*-adversaries.

7.2 General Correspondences

In this section, we explain how to prove general correspondences. Moreover, we also show that, when our verifier proves injectivity, it proves recentness as well. For example, when it proves a correspondence $\text{event}(M) \rightsquigarrow \text{inj event}(M')$, it shows that, when the event $\mathtt{event}(M)$ has been executed, not only the event $\mathtt{event}(M')$ has been executed, but also this event has been executed recently. As explained by Lowe [54], the precise meaning of "recent" depends on the circumstances: it can be that $\mathtt{event}(M)$ is executed within the duration of the part of the process after $\mathtt{event}(M')$, or it can be within a certain number of time units. Here, we define recentness as follows: the runtime of the session that executes $\mathtt{event}(M)$ overlaps with the runtime of the session that executes the corresponding $\mathtt{event}(M')$ event.

We can formally define recent correspondences for instrumented processes as follows. We assume that, in P_0, the events are under at least one replication. We define an instrumented process $P_0' = \text{instr}'(P_0)$, where $\text{instr}'(P_0)$ is defined like $\text{instr}(P_0)$, except that the events $\mathtt{event}(M)$ in P_0 are replaced with $\mathtt{event}(M, i)$, where i is the session identifier that labels the down-most replication above $\mathtt{event}(M)$ in P_0. The session identifier i indicates the session in which the considered event is executed.

When $\overline{k} = k_1 \dots k_n$ is a non-empty sequence of indices, we denote by $\overline{k}\lceil$ the sequence obtained by removing the last index from $\overline{k}$: $\overline{k}\lceil = k_1 \dots k_{n-1}$.

Definition 14 Let P_0 be a closed process and $P_0' = \text{instr}'(P_0)$. We say that P_0' *satisfies the recent correspondence*

$$\text{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(p_j') \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} q_{jk} \right)$$

where

$$q_{\overline{jk}} = \text{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

against *Init*-adversaries if and only if for any *Init*-adversary Q, for any trace $\mathcal{T} = S_0, E_0, \{P_0', Q'\} \rightarrow^* S', E', \mathcal{P}'$, with $Q' = \text{instrAdv}(Q)$, $E_0(a) = a[\,]$ for all $a \in dom(E_0)$, and $fn(P_0') \cup Init \subseteq dom(E_0)$, there exists a function $\phi_{\overline{jk}}$ for each non-empty $\overline{jk}$, such that for all non-empty $\overline{jk}$, $\phi_{\overline{jk}}$ maps a subset of steps of $\mathcal{T}$ to steps of $\mathcal{T}$ and

- For all τ, if the event $\mathtt{event}(\sigma p, \lambda_\epsilon)$ is executed at step τ in $\mathcal{T}$ for some σ and λ_ϵ, then there exist σ' and $J = (j_{\overline{k}})_{\overline{k}}$ such that $\sigma' p'_{j_\epsilon} = \sigma p$ and, for all non-empty $\overline{k}$, $\phi_{\text{makejk}(\overline{k},J)}(\tau)$ is defined, $\mathtt{event}(\sigma' p_{\text{makejk}(\overline{k},J)}, \lambda_{\overline{k}})$ is executed at step $\phi_{\text{makejk}(\overline{k},J)}(\tau)$ in $\mathcal{T}$, and if $[\text{inj}]_{\text{makejk}(\overline{k},J)} = \text{inj}$, then the runtimes of $\text{session}(\lambda_{\overline{k}\lceil})$ and $\text{session}(\lambda_{\overline{k}})$ overlap (recentness).

  The runtime of $\text{session}(\lambda)$ begins when the rule $S, E, \mathcal{P} \cup \{\, !^i P \,\} \rightarrow S \setminus \{\lambda\}, E, \mathcal{P} \cup \{\, P\{\lambda/i\}, !^i P \,\}$ is applied and ends when $P\{\lambda/i\}$ has disappeared.

- For all non-empty $\overline{jk}$, if $[\text{inj}]_{\overline{jk}} = \text{inj}$, then $\phi_{\overline{jk}}$ is injective.

- For all non-empty $\overline{jk}$, for all j and k, if $\phi_{\overline{jk}jk}(\tau)$ is defined, then $\phi_{\overline{jk}}(\tau)$ is defined and $\phi_{\overline{jk}jk}(\tau) \leq \phi_{\overline{jk}}(\tau)$. For all j and k, if $\phi_{jk}(\tau)$ is defined, then $\phi_{jk}(\tau) \leq \tau$.

We do not define recentness for standard processes, since it is difficult to track formally the runtime of a session in these processes. Instrumented processes make that very easy thanks to session identifiers. It is easy to infer correspondences for standard processes from recent correspondences for instrumented processes, with a proof similar to that of Lemma 1.

Lemma 4 *Let P_0 be a closed process and $P'_0 = \text{instr}'(P_0)$. Let $M_{\overline{jk}}$, M, and M'_j be terms. Let $p_{\overline{jk}}, p, p'_j$ be the patterns obtained by replacing names a with patterns $a[\,]$ in the terms $M_{\overline{jk}}, M, M'_j$ respectively. If P'_0 satisfies the recent correspondence*

$$\text{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} q_{jk} \right)$$

where

$$q_{\overline{jk}} = \text{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

against Init-adversaries then P_0 satisfies the correspondence

$$\text{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(M'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} q'_{jk} \right)$$

where

$$q'_{\overline{jk}} = \text{event}(M_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q'_{\overline{jk}jk}$$

against Init-adversaries.

Let P_0 be a closed process and $P'_0 = \text{instr}'(P_0)$. We adapt the generation of clauses as follows: the set of clauses $\mathcal{R}'_{P'_0,Init}$ is defined as $\mathcal{R}_{P'_0,Init}$ except that

$$\begin{aligned}
&[\![\overline{M}\langle N\rangle.P]\!]\rho H = [\![P]\!]\rho H \cup \{H\{\rho_{|V_o \cup V_s}/\Box\} \Rightarrow \text{message}(\rho(M), \rho(N))\} \\
&[\![!^i P]\!]\rho H = [\![P]\!](\rho[i \mapsto i])(H\{\rho_{|V_o \cup V_s}/\Box\}) \\
&[\![\texttt{event}(M, i).P]\!]\rho H = [\![P]\!]\rho(H \wedge \text{m-event}(\rho(M), \Box)) \cup \{H \Rightarrow \text{event}(\rho(M), i)\}
\end{aligned}$$

where $\Box$ is a special variable. The predicate event has as additional argument the session identifier in which the event is executed. The predicate m-event has as additional argument an environment ρ that gives values that variables will contain at the first output or replication that follows the event; $\Box$ is a placeholder for this environment. (Recall that V_o is the set of ordinary variables and V_s the set of session identifier variables, so $\rho_{|V_o \cup V_s}$ is the environment restricted to variables, names being excluded.) We define $\mathsf{solve}'_{P'_0,Init}$ as $\mathsf{solve}_{P'_0,Init}$ except that it applies to $\mathcal{R}'_{P'_0,Init}$ instead of $\mathcal{R}_{P'_0,Init}$.

Let us first consider the particular case of injective correspondences. We consider general correspondences in Theorem 5 below.

Proposition 2 (Injective correspondences) *Let P_0 be a closed process and $P'_0 = \text{instr}'(P_0)$. We assume that, in P_0, all events are of the form $\texttt{event}(f(M_1, \dots, M_n))$ and that different occurrences of* `event` *have different root function symbols.*

We also assume that the patterns p, p'_j, p_{jk} satisfy the following conditions: p and p'_j for $j \in \{1, \dots, m\}$ are of the form $f(\dots)$ for some function symbol f and for all j, k such that $[\text{inj}]_{jk} = \text{inj}$, $p_{jk} = f_{jk}(\dots)$ for some function symbol f_{jk}.

Let $\mathsf{solve}'_{P'_0,Init}(\text{event}(p, i)) = \{R_{jr} \mid j \in \{1, \dots, m\}, r \in \{1, \dots, n_j\}\}$. Assume that there exist x_{jk}, i_{jr}, and ρ_{jrk} ($j \in \{1, \dots, m\}$, $r \in \{1, \dots, n_j\}$, $k \in \{1, \dots, l_j\}$) such that

- *For all $j \in \{1, \dots, m\}$, for all $r \in \{1, \dots, n_j\}$, there exist H and σ such that $R_{jr} = H \wedge \text{m-event}(\sigma p_{j1}, \rho_{jr1}) \wedge \dots \wedge \text{m-event}(\sigma p_{jl_j}, \rho_{jrl_j}) \Rightarrow \text{event}(\sigma p'_j, i_{jr})$.*

- *For all* $j \in \{1,\dots,m\}$*, for all* r *and* r' *in* $\{1,\dots,n_j\}$*, for all* $k \in \{1,\dots,l_j\}$ *such that* $[\text{inj}]_{jk} = \text{inj}$*,* $\rho_{jrk}(x_{jk})\{\lambda/i_{jr}\}$ *does not unify with* $\rho_{jr'k}(x_{jk})\{\lambda'/i_{jr'}\}$ *when* $\lambda \neq \lambda'$*.*

Then P'_0 *satisfies the recent correspondence*

$$\text{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk}\text{event}(p_{jk})\right)$$

against Init-adversaries.

This proposition is a particular case of Theorem 5 below. It is proved in Appendix E. By Theorem 3, after deleting session identifiers and environments, the first item shows that P'_0 satisfies the correspondence

$$\text{event}(p) \Rightarrow \bigvee_{j=1..m,r} \left(\text{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \text{event}(p_{jk})\right) \tag{13}$$

The environments and session identifiers as well as the second item serve in proving injectivity. Suppose that $[\text{inj}]_{jk} = \text{inj}$, and denote by $_$ an unknown term. If two instances of $\mathtt{event}(p, i)$ are executed in P'_0 for the branch j of the correspondence, by the first item, they are instances of $\mathtt{event}(\sigma_{jr}p'_j, i_{jr})$ for some r, so they are $\mathtt{event}(\sigma'_1\sigma_{jr_1}p'_j, \sigma'_1 i_{jr_1})$ and $\mathtt{event}(\sigma'_2\sigma_{jr_2}p'_j, \sigma'_2 i_{jr_2})$ for some σ'_1 and σ'_2. Furthermore, there is only one occurrence of $\mathtt{event}(f(\dots), i)$ in P'_0, so the event $\mathtt{event}(f(\dots), i)$ can be executed at most once for each value of the session identifier i, so $\sigma'_1 i_{jr_1} \neq \sigma'_2 i_{jr_2}$. Then, by the first item, corresponding events $\mathtt{event}(\sigma'_1\sigma_{jr_1}p_{jk}, _)$ and $\mathtt{event}(\sigma'_2\sigma_{jr_2}p_{jk}, _)$ have been executed, with associated environments $\sigma'_1\rho_{jr_1k}$ and $\sigma'_2\rho_{jr_2k}$. By the second item, $\rho_{jr_1k}(x_{jk})\{\lambda_1/i_{jr_1}\}$ does not unify with $\rho_{jr_2k}(x_{jk})\{\lambda_2/i_{jr_2}\}$ for different values $\lambda_1 = \sigma'_1 i_{jr_1}$ and $\lambda_2 = \sigma'_2 i_{jr_2}$ of the session identifier. (In this condition, r_1 can be equal to r_2, and when $r_1 = r_2 = r$, the condition simply means that i_{jr} occurs in ρ_{jrk}.) So $\sigma'_1\rho_{jr_1k}(x_{jk}) \neq \sigma'_2\rho_{jr_2k}(x_{jk})$, so the events $\mathtt{event}(\sigma'_1\sigma_{jr_1}p_{jk}), _)$ and $\mathtt{event}(\sigma'_2\sigma_{jr_2}p_{jk}), _)$ are distinct, which shows injectivity. This point is very similar to the fact that injective agreement is implied by non-injective agreement when the parameters of events contain nonces generated by the agent to whom authentication is being made, because the event can be executed at most once for each value of the nonce. (The session identifier i_{jr} in our theorem plays the role of the nonce.) [Andrew Gordon, personal communication].

Corollary 3 (Recent injective agreement) *Let* P_0 *be a closed process and* $P'_0 = \text{instr}'(P_0)$*. We assume that, in* P_0*, all events are of the form* $\mathtt{event}(f(M_1,\dots,M_k))$ *and that different occurrences of* `event` *have different root function symbols. Let* $\{R_1,\dots,R_n\} = \mathsf{solve}'_{P'_0,Init}(\text{event}(e(x_1,\dots,x_m), i))$*. Assume that there exist* x*,* i_r*, and* ρ_r *(*$r \in \{1,\dots,n\}$*) such that*

- *For all* $r \in \{1,\dots,n\}$*,* $R_r = H \wedge \text{m-event}(e'(p_1,\dots,p_m), \rho_r) \Rightarrow \text{event}(e(p_1,\dots,p_m), i_r)$ *for some* $p_1,\dots,p_m$*, and* H*.*

- *For all* r *and* r' *in* $\{1,\dots,n\}$*,* $\rho_r(x)\{\lambda/i_r\}$ *does not unify with* $\rho_{r'}(x)\{\lambda'/i_{r'}\}$ *when* $\lambda \neq \lambda'$*.*

Then P'_0 *satisfies the recent correspondence* $\text{event}(e(x_1,\dots,x_m)) \rightsquigarrow \text{inj event}(e'(x_1,\dots,x_m))$ *against Init-adversaries.*

Proof This result is an immediate consequence of Proposition 2. □

Example 12 For the process P of Section 2.3, $P' = \text{instr}'(P)$, and $\mathit{Init} = \{c\}$, we have

$$\begin{aligned}
&\mathsf{solve}'_{P',\mathit{Init}}(\text{event}(e_B(x_1,x_2,x_3,x_4),i)) = \\
&\quad\{H \wedge \text{m-event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]),\rho) \\
&\qquad \Rightarrow \text{event}(e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i_{B0})\} \\
&\text{where } pk_A = pk(sk_A[\,]),\ pk_B = pk(sk_B[\,]) \\
&\qquad p_1 = pencrypt_p((a[pk_B, i_{A0}], pk_A), pk_B, r_1[pk_B, i_{A0}]) \\
&\qquad p_2 = pencrypt_p((a[pk_B, i_{A0}], b[p_1, i_{B0}], pk_B), pk_A, r_2[p_1, i_{B0}]) \\
&\qquad \rho = \{i_A \mapsto i_{A0}, x_pk_B \mapsto pk_B, m \mapsto p_2\}
\end{aligned}$$

Intuitively, this result shows that each event $e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$, executed in the session of index $i_B = i_{B0}$ is preceded by an event $e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_A = i_{A0}$ with $x_pk_B = pk_B$ and $m = p_2$. Since i_{B0} occurs in this event (or in its environment[4]), different executions of e_B, which have different values of i_{B0}, cannot correspond to the same execution of e_3, so we have injectivity. More formally, the second hypothesis of Corollary 3 is satisfied because $\rho(m)\{\lambda/i_{B0}\}$ does not unify with $\rho(m)\{\lambda'/i_{B0}\}$ when $\lambda \neq \lambda'$, since i_{B0} occurs in $\rho(m) = p_2$. Then, P' satisfies the recent correspondence $\text{event}(e_B(x_1,x_2,x_3,x_4)) \rightsquigarrow \text{inj event}(e_3(x_1,x_2,x_3,x_4))$ against *Init*-adversaries.

The tool shows in a similar way that P' satisfies the recent correspondence $\text{event}(e_A(x_1,x_2, x_3,x_4)) \rightsquigarrow \text{inj event}(e_2(x_1,x_2,x_3,x_4))$ against *Init*-adversaries.

Let us now consider the case of general correspondences. The basic idea is to decompose the general correspondence to prove into several correspondences. For instance, the correspondence $\text{event}(e_B(x_1,x_2,x_3,x_4)) \rightsquigarrow (\text{event}(e_3(x_1,x_2,x_3,x_4)) \rightsquigarrow \text{event}(e_2(x_1,x_2,x_3,x_4)))$ is implied by the conjunction of the correspondences $\text{event}(e_B(x_1,x_2,x_3,x_4)) \rightsquigarrow \text{event}(e_3(x_1,x_2,x_3,x_4))$ and $\text{event}(e_3(x_1,x_2,x_3,x_4)) \rightsquigarrow \text{event}(e_2(x_1,x_2,x_3,x_4))$. However, as noted in Section 3.3, this proof technique would often fail because, in order to prove that $e_2(x_1,x_2,x_3,x_4)$ has been executed, we may need to know that $e_B(x_1,x_2,x_3,x_4)$ has been executed, and not only that $e_3(x_1,x_2,x_3, x_4)$ has been executed. To solve this problem, we use the following idea: when we know that $e_B(x_1,x_2,x_3,x_4)$ has been executed, we may be able to show that certain particular instances of $e_3(x_1,x_2,x_3,x_4)$ have been executed, and we can exploit this information in order to prove that $e_2(x_1,x_2,x_3,x_4)$ has been executed. In other words, we rather prove the correspondences $\text{event}(e_B(x_1,x_2,x_3,x_4)) \Rightarrow \bigvee_{r=1}^{m} \sigma_r\text{event}(e_B(x_1,x_2,x_3,x_4)) \rightsquigarrow \sigma_r\text{event}(e_3(x_1,x_2,x_3,x_4))$ and for all $r \leq m$, $\sigma_r\text{event}(e_3(x_1,x_2,x_3,x_4)) \rightsquigarrow \sigma_r\text{event}(e_2(x_1,x_2,x_3,x_4))$. When the considered general correspondence has several nesting levels, we perform such a decomposition recursively. The next theorem generalizes and formalizes these ideas.

Below, the notation $(\mathit{Env}_{\overline{jk}})_{\overline{jk}}$ represents a family $\mathit{Env}_{\overline{jk}}$ of sets of pairs (ρ, i) where ρ is an environment and i is a session identifier, one for each non-empty $\overline{jk}$. The notation $(\mathit{Env}_{jk\overline{jk}})_{\overline{jk}}$ represents a subfamily of $(\mathit{Env}_{\overline{jk}})_{\overline{jk}}$ in which the first two indices are jk, and this family is reindexed by omitting the fixed indices jk.

Theorem 5 *Let P_0 be a closed process and $P'_0 = \text{instr}'(P_0)$. We assume that, in P_0, all events are of the form* `event`$(f(M_1,\ldots,M_n))$ *and that different occurrences of* `event` *have different root function symbols.*

Let us define $\text{verify}(q', (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$, *where $\overline{jk}$ is non-empty, by:*

V1. If $q' = \text{event}(p)$ for some p, then $\text{verify}(q', (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$ *is true.*

[4]In general, the environment may contain more variables than the event itself, so looking for the session identifiers in the environment instead of the event is more powerful.

V2. If $q' = \text{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} q'_{jk}\right)$ *and* $q'_{jk} = \text{event}(p_{jk}) \rightsquigarrow \dots$ *for some* p, p'_j, *and* p_{jk}, *where* $m \neq 1$, $l_j \neq 0$, *or* $p \neq p'_1$, *then* $\text{verify}(q', (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$ *is true if and only if there exists* $(\sigma_{jr})_{jr}$ *such that the following three conditions hold:*

V2.1. We have $\mathsf{solve}'_{P'_0,\mathit{Init}}(\text{event}(p,i)) \subseteq \{H \wedge \bigwedge_{k=1}^{l_j} \text{m-event}(\sigma_{jr} p_{jk}, \rho_{jrk}) \Rightarrow \text{event}(\sigma_{jr} p'_j, i_{jr})$ *for some* H, $j \in \{1, \dots, m\}$, r, *and* $(\rho_{jrk}, i_{jr}) \in \mathit{Env}_{jk}$ *for all* $k\}$.

V2.2. For all j, r, k_0, *the common variables between* $\sigma_{jr} q'_{jk_0}$ *on the one hand and* $\sigma_{jr} p'_j$ *and* $\sigma_{jr} q'_{jk}$ *for all* $k \neq k_0$ *on the other hand occur in* $\sigma_{jr} p_{jk_0}$.

V2.3. For all j, r, k, $\text{verify}(\sigma_{jr} q'_{jk}, (\mathit{Env}_{jk\overline{jk}})_{\overline{jk}})$ *is true.*

Consider the following recent correspondence:

$$q = \text{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} q_{jk}\right)$$

where

$$q_{\overline{jk}} = \text{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

We assume that the patterns in the correspondence satisfy the following conditions: p *and* p'_j *for* $j \in \{1, \dots, m\}$ *are of the form* $f(\dots)$ *for some function symbol* f *and, for all non-empty* $\overline{jk}$ *such that* $[\text{inj}]_{\overline{jk}} = \text{inj}$, $p_{\overline{jk}} = f_{\overline{jk}}(\dots)$ *for some function symbol* $f_{\overline{jk}}$. *We also assume that if* inj *occurs in* $q_{\overline{jk}}$, *then* $[\text{inj}]_{\overline{jk}} = \text{inj}$.

Assume that there exist $(\mathit{Env}_{\overline{jk}})_{\overline{jk}}$ *and* $(x_{\overline{jk}})_{\overline{jk}}$, *where* $\overline{jk}$ *is non-empty, such that*

H1. $\text{verify}(q, (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$ *is true.*

H2. For all non-empty $\overline{jk}$, *if* $[\text{inj}]_{\overline{jk}} = \text{inj}$, *then for all* $(\rho, i), (\rho', i') \in \mathit{Env}_{\overline{jk}}$, $\rho(x_{\overline{jk}})\{\lambda/i\}$ *does not unify with* $\rho'(x_{\overline{jk}})\{\lambda'/i'\}$ *when* $\lambda \neq \lambda'$.

Then P'_0 *satisfies the recent correspondence* q *against* Init*-adversaries.*

This theorem is rather complex, so we give some intuition here. Its proof can be found in Appendix E.

Point V2.1 allows us to infer correspondences by Theorem 3: after deleting session identifiers and environments, P'_0 satisfies the correspondences:

$$\text{event}(p) \Rightarrow \bigvee_{j=1..m,r} \left(\text{event}(\sigma_{jr} p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \text{event}(\sigma_{jr} p_{jk})\right) \tag{14}$$

and, using the recursive calls of Point V2.3,

$$\text{event}(\sigma'_{\overline{jrk}\lceil} p_{\overline{jk}}) \Rightarrow \bigvee_{j=1..m_{\overline{jk}},r} \left(\text{event}(\sigma'_{\overline{jrk}jr} p_{\overline{jk}}) \rightsquigarrow \bigwedge_{k=1}^{l_{\overline{jk}j}} \text{event}(\sigma'_{\overline{jrk}jr} p_{\overline{jk}jk})\right) \tag{15}$$

against *Init*-adversaries, where $\sigma'_{\overline{jrk}jr} = \sigma_{\overline{jrk}jr} \sigma_{\overline{jrk}\lceil} \dots \sigma_{jr}$ and we denote by $\sigma_{\overline{jrk}jr}$ the substitution σ_{jr} obtained in recursive calls to verify indexed by $\overline{jrk}$. In order to infer the desired correspondence, we need to show injectivity properties and to combine the correspondences (14) and (15) into a single correspondence. Injectivity comes from Hypothesis H2: this hypothesis generalizes the second item of Proposition 2 to the case of general correspondences.

The correspondences (14) and (15) are combined into a single correspondence using Point V2.2. We illustrate this point on the simple example of the correspondence $\text{event}(p) \Rightarrow (\text{event}(p'_1) \rightsquigarrow (\text{event}(p_{11}) \rightsquigarrow \text{event}(p_{1111})))$. By V2.1 and the recursive call of V2.3, we have correspondences of the form:

$$\text{event}(p) \Rightarrow \bigvee_r \left(\text{event}(\sigma_{1r}p'_1) \rightsquigarrow \text{event}(\sigma_{1r}p_{11})\right) \tag{16}$$

$$\text{event}(\sigma_{1r}p_{11}) \Rightarrow \bigvee_{r'} (\text{event}(\sigma_{1r11r'}\sigma_{1r}p_{11}) \rightsquigarrow \text{event}(\sigma_{1r11r'}\sigma_{1r}p_{1111})) \tag{17}$$

for some σ_{1r} and $\sigma_{1r11r'}$. The correspondence (17) implies the simpler correspondence

$$\text{event}(\sigma_{1r}p_{11}) \rightsquigarrow \text{event}(\sigma_{1r}p_{1111}). \tag{18}$$

Furthermore, if an instance of $\text{event}(p)$ is executed, $e_1 = \text{event}(\sigma p)$, then by (16), for some r and σ'_1 such that $\sigma p = \sigma'_1\sigma_{1r}p'_1$, the event $e_2 = \text{event}(\sigma'_1\sigma_{1r}p_{11})$ has been executed before e_1. By (18), for some σ'_2 such that $\sigma'_1\sigma_{1r}p_{11} = \sigma'_2\sigma_{1r}p_{11}$, the event $e_3 = \text{event}(\sigma'_2\sigma_{1r}p_{1111})$ has been executed before e_2. We now need to reconcile the substitutions σ'_1 and σ'_2; this can be done thanks to V2.2. Let us define σ'' such that $\sigma''x = \sigma'_1x$ for $x \in \mathit{fv}(\sigma_{1r}p_{11}) \cup \mathit{fv}(\sigma_{1r}p'_1)$ and $\sigma''x = \sigma'_2x$ for $x \in \mathit{fv}(\sigma_{1r}p_{1111}) \cup \mathit{fv}(\sigma_{1r}p_{11})$. Such a substitution σ'' exists because the common variables between $\mathit{fv}(\sigma_{1r}p_{11}) \cup \mathit{fv}(\sigma_{1r}p'_1)$ and $\mathit{fv}(\sigma_{1r}p_{1111}) \cup \mathit{fv}(\sigma_{1r}p_{11})$ occur in $\sigma_{1r}p_{11}$ by V2.2, and for the variables $x \in \mathit{fv}(\sigma_{1r}p_{11})$, $\sigma'_1x = \sigma'_2x$ since $\sigma'_1\sigma_{1r}p_{11} = \sigma'_2\sigma_{1r}p_{11}$. So, for some r and σ'' such that $\sigma p = \sigma''\sigma_{1r}p'_1$, the event $e_2 = \text{event}(\sigma''\sigma_{1r}p_{11})$ has been executed before e_1 and $e_3 = \text{event}(\sigma''\sigma_{1r}p_{1111})$ has been executed before e_2. This result proves the desired correspondence $\text{event}(p) \Rightarrow (\text{event}(p'_1) \rightsquigarrow (\text{event}(p_{11}) \rightsquigarrow \text{event}(p_{1111})))$. Point V2.2 generalizes this technique to any correspondence.

In the implementation, the hypotheses of this theorem are checked as follows. In order to check $\text{verify}(q', (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$, we first compute $\mathsf{solve}'_{P'_0,\mathit{Init}}(\text{event}(p, i))$. By matching, we check V2.1 and obtain the values of σ_{jr}, ρ_{jrk}, and i_{jr} for all j, r, and k. We add (ρ_{jrk}, i_{jr}) to Env_{jk}. We compute $\sigma_{jr}p'_j$ and $\sigma_{jr}q'_{jk}$ for each j, r, and k, and check V2.2 and V2.3.

After checking $\text{verify}(q', (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$, we finally check Hypothesis H2 for each $\overline{jk}$. We start with a set that contains the whole domain of ρ for some $(\rho, i) \in \mathit{Env}_{\overline{jk}}$. For each (ρ, i) and (ρ', i') in $\mathit{Env}_{\overline{jk}}$, we remove from this set the variables x such that $\rho(x)\{\lambda/i\}$ unifies with $\rho'(x)\{\lambda'/i'\}$ for $\lambda \neq \lambda'$. When the obtained set is non-empty, Hypothesis H2 is satisfied by taking for $x_{\overline{jk}}$ any element of the obtained set. Otherwise, Hypothesis H2 is not satisfied.

Example 13 For the example P of Section 2.3, the previous theorem does not enable us to prove the correspondence $\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_1(x_1, x_2, x_3))))$ directly. Indeed, Theorem 5 would require that we show a correspondence of the form $\text{event}(\sigma e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(\sigma e_1(x_1, x_2, x_3))$. However, such a correspondence does not hold, because after executing a single event e_1, the adversary can replay the first message of the protocol, so that B executes several events e_2.

It is still possible to prove this correspondence by combining the automatic proof of the slightly weaker correspondence $q = \text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_1(x_1, x_2, x_3)) \wedge \text{inj event}(e_2(x_1, x_2, x_3, x_4))))$, which does not order the events e_1 and e_2, with a simple manual argument. (This technique applies to many other examples.) Let us first prove the latter correspondence.

Let $P' = \text{instr}'(P)$ and $\mathit{Init} = \{c\}$. We have

$$\begin{aligned}&\mathsf{solve}'_{P',\mathit{Init}}(\text{event}(e_B(x_1, x_2, x_3, x_4), i)) =\\ &\quad\{H \wedge \text{m-event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), \rho_{111})\\ &\qquad \Rightarrow \text{event}(e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i_{B0})\}\end{aligned}$$

$$
\begin{aligned}
&\mathsf{solve}'_{P',Init}(\text{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i)) = \\
&\quad \{\text{m-event}(e_1(pk_A, pk_B, a[pk_B, i_{A0}]), \rho_{111111}) \\
&\quad \wedge \text{m-event}(e_2(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), \rho_{111112}) \\
&\qquad \Rightarrow \text{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i_{A0})\} \\
&\text{where } pk_A = pk(sk_A[\,]),\ pk_B = pk(sk_B[\,]) \\
&\qquad p_1 = pencrypt_p((a[pk_B, i_{A0}], pk_A), pk_B, r_1[pk_B, i_{A0}]) \\
&\qquad p_2 = pencrypt_p((a[pk_B, i_{A0}], b[p_1, i_{B0}], pk_B), pk_A, r_2[p_1, i_{B0}]) \\
&\qquad \rho_{111} = \rho_{111111} = \{i_A \mapsto i_{A0}, x_pk_B \mapsto pk_B, m \mapsto p_2\} \\
&\qquad \rho_{111112} = \{i_B \mapsto i_{B0}, m' \mapsto p_1\}
\end{aligned}
$$

Intuitively, as in Example 12, the value of $\mathsf{solve}'_{P',Init}(\text{event}(e_B(x_1, x_2, x_3, x_4), i))$ guarantees that each event $e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$, executed in the session of index $i_B = i_{B0}$ is preceded by an event $e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_A = i_{A0}$ with $x_pk_B = pk_B$ and $m = p_2$. Since i_{B0} occurs in this event (or in its environment), we have injectivity. The value of $\mathsf{solve}'_{P',Init}(\text{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i))$ guarantees that each event $e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_A = i_{A0}$ is preceded by events $e_1(pk_A, pk_B, a[pk_B, i_{A0}])$ executed in the session of index $i_A = i_{A0}$ with $x_pk_B = pk_B$ and $m = p_2$, and $e_2(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_B = i_{B0}$ with $m' = p_1$. Since i_{A0} occurs in these events (or in their environments), we have injectivity. So we obtain the desired correspondence $\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_1(x_1, x_2, x_3)) \wedge \text{inj event}(e_2(x_1, x_2, x_3, x_4))))$.

More formally, let us show that we can apply Theorem 5. We have $p = p'_1 = e_B(x_1, x_2, x_3, x_4)$, $p_{11} = e_3(x_1, x_2, x_3, x_4)$, $p_{1111} = e_1(x_1, x_2, x_3)$, $p_{1112} = e_2(x_1, x_2, x_3, x_4)$. We show $\text{verify}(q, (Env_{\overline{jk}})_{\overline{jk}})$. Given the first value of $\mathsf{solve}'_{P',Init}$ shown above, we satisfy V2.1 by letting $\sigma_{11} = \{x_1 \mapsto pk_A, x_2 \mapsto pk_B, x_3 \mapsto a[pk_B, i_{A0}], x_4 \mapsto b[p_1, i_{B0}]\}$ and $i_{11} = i_{B0}$, with $(\rho_{111}, i_{11}) \in Env_{11}$. The common variables between $\sigma_{11}q_{11} = \text{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])) \rightsquigarrow (\text{inj event}(e_1(pk_A, pk_B, a[pk_B, i_{A0}])) \wedge \text{inj event}(e_2(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])))$ and $\sigma_{11}p'_1 = e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ are i_{A0} and i_{B0}, and they occur in $\sigma_{11}p_{11} = e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$. So we have V2.2. Recursively, in order to obtain V2.3, we have to show $\text{verify}(\sigma_{11}q_{11}, (Env_{11\overline{jk}})_{\overline{jk}})$. Given the second value of $\mathsf{solve}'_{P',Init}$ shown above, we satisfy V2.1 by letting $\sigma_{11111} = \text{Id}$ and $i_{11111} = i_{A0}$, with $(\rho_{111111}, i_{11111}) \in Env_{1111}$ and $(\rho_{111112}, i_{11111}) \in Env_{1112}$. (We prefix the indices with 111 in order to represent that these values concern the recursive call with $j = 1$, $r = 1$, and $k = 1$.) V2.2 holds trivially, because $\sigma_{11111}\sigma_{11}q_{111k_0} = \sigma_{11111}\sigma_{11}\text{event}(p_{111k_0})$, since the considered correspondence has one nesting level only. V2.3 holds because q_{1111} reduces to $\text{event}(p_{1111})$, so $\text{verify}(\sigma_{11111}\sigma_{11}q_{1111}, (Env_{1111\overline{jk}})_{\overline{jk}})$ holds by V1, and the situation is similar for q_{1112}. Therefore, we obtain H1. In order to show H2, we have to find x_{11} such that $\rho_{111}(x_{11})\{\lambda/i_{11}\}$ does not unify with $\rho_{111}(x_{11})\{\lambda'/i_{11}\}$ when $\lambda \neq \lambda'$. This property holds with $x_{11} = m$, because $i_{11} = i_{B0}$ occurs in $\rho_{111}(m) = p_2$. Similarly, $\rho_{111111}(x_{1111})\{\lambda/i_{11111}\}$ does not unify with $\rho_{111111}(x_{1111})\{\lambda'/i_{11111}\}$ when $\lambda \neq \lambda'$, for $x_{1111} = i_A$, since $i_{11111} = i_{A0}$ occurs in $\rho_{111111}(i_A)$. Finally, $\rho_{111112}(x_{1112})\{\lambda/i_{11111}\}$ does not unify with $\rho_{111112}(x_{1112})\{\lambda'/i_{11111}\}$ when $\lambda \neq \lambda'$ for $x_{1112} = m'$, since $i_{11111} = i_{A0}$ occurs in $\rho_{111112}(m') = p_1$. So, by Theorem 5, the process P' satisfies the recent correspondence $\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_1(x_1, x_2, x_3)) \wedge \text{inj event}(e_2(x_1, x_2, x_3, x_4))))$ against *Init*-adversaries.

We can then show that P' satisfies the recent correspondence $\text{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\text{inj event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \text{inj event}(e_1(x_1, x_2, x_3))))$. We just have to show that the event $e_2(x_1, x_2, x_3, x_4)$ is executed after $e_1(x_1, x_2, x_3)$. The nonce a is created just before executing $e_1(x_1, x_2, x_3) = e_1(pk_A, x_pk_B, a)$, and the event $e_2(x_1, x_2, x_3, x_4) = e_2(x_pk_A, pk_B, x_a, b)$ contains a in the variable $x_3 = x_a$. So e_2 has been executed after receiving a message that contains a, so after a has been sent in some message, so after executing

event e_1.

8 Termination

In this section, we study termination properties of our algorithm. We first show that it terminates on a restricted class of protocols, named *tagged protocols*. Then, we study how to improve the choice of the selection function in order to obtain termination in other cases.

8.1 Termination for Tagged Protocols

Intuitively, a tagged protocol is a protocol in which each application of a constructor can be immediately distinguished from others in the protocol, for example by a tag: for instance, when we want to encrypt m under k, we add the constant tag ct_0 to m, so that the encryption becomes $\mathit{sencrypt}((ct_0, m), k)$ where the tag ct_0 is a different constant for each encryption in the protocol. The tags are checked when destructors are applied. This condition is easy to realize by adding tags, and it is also a good protocol design: the participants use the tags to identify the messages unambiguously, thus avoiding type flaw attacks [50].

In [20], in collaboration with Andreas Podelski, we have given conditions on the clauses that intuitively correspond to tagged protocols, and we have shown that, for tagged protocols using only public channels, public-key cryptography with atomic keys, shared-key cryptography and hash functions, and for secrecy properties, the solving algorithm using the selection function sel_0 terminates.

Here, we extend this result by giving a definition of tagged protocols for processes and showing that the clause generation algorithm yields clauses that satisfy the conditions of [20], so that the solving algorithm terminates. (A similar result has been proved for strong secrecy in the technical report [16].)

Definition 15 (Tagged protocol) A tagged protocol is a process P_0 together with a signature of constructors and destructors such that:

C1. The only constructors and destructors are those of Figure 2, plus *equal*.

C2. In every occurrence of $M(x)$ and $\overline{M}\langle N\rangle$ in P_0, M is a name free in P_0.

C3. In every occurrence of $f(\ldots)$ with $f \in \{\mathit{sencrypt}, \mathit{sencrypt}_p, \mathit{pencrypt}_p, \mathit{sign}, \mathit{nmrsign}, h, \mathit{mac}\}$ in P_0, the first argument of f is a tuple $(ct, M_1, \ldots, M_n)$, where the tag ct is a constant. Different occurrences of f have different values of the tag ct.

C4. In every occurrence of $\mathit{let}\ x = g(\ldots)\ \mathit{in}\ P\ \mathit{else}\ Q$, for $g \in \{\mathit{sdecrypt}, \mathit{sdecrypt}_p, \mathit{pdecrypt}_p, \mathit{checksignature}, \mathit{getmessage}\}$ in P_0, $P = \mathit{let}\ y = 1\mathit{th}_n(x)\ \mathit{in}\ \mathit{if}\ y = ct\ \mathit{then}\ P'$ for some ct and P'.

In every occurrence of *nmrchecksign* in P_0, its third argument is $(ct, M_1, \ldots, M_n)$ for some $ct, M_1, \ldots, M_n$.

C5. The destructor applications (including equality tests) have no *else* branches. There exists a trace of P_0 (without adversary) in which all program points are executed exactly once.

C6. The second argument of $\mathit{pencrypt}_p$ in the trace of Condition C5 is of the form $\mathit{pk}(M)$ for some M.

C7. The arguments of pk and host in the trace of Condition C5 are atomic constants (free names or names created by restrictions not under inputs, non-deterministic destructor applications, or replications) and they are not tags.

Condition C1 limits the set of allowed constructors and destructors. We could give conditions on the form of allowed destructor rules, but these conditions are complex, so it is simpler and more intuitive to give an explicit list. Condition C2 states that all channels must be public. This condition avoids the need for the predicate message. Condition C3 guarantees that tags are added in all messages, and Condition C4 guarantees that tags are always checked.

In most cases, the trace of Condition C5 is simply the intended execution of the protocol. All terms that occur in the trace of Condition C5 have pairwise distinct tags (since each program point is executed at most once, and tags at different program points are different by Condition C3). We can prove that it also guarantees that the terms of all clauses generated for the process P_0 have instances in the set of terms that occur in the trace of Condition C5 (using the fact that all program points are executed at least once). These properties are key in the termination proof. More concretely, Condition C5 means that, after removing replications of P_0, the resulting process has a trace that executes each program point (at least) once. In this trace, all destructor applications succeed and the process reduces to a configuration with an empty set of processes. Since, after removing replications, the number of traces of a process is always finite, Condition C5 is decidable.

Condition C6 means that, in its intended execution, the protocol uses public-key encryption only with public keys, and Condition C7 means that long-term secret (symmetric and asymmetric) keys are atomic constants.

Example 14 A tagged protocol can easily be obtained by tagging the Needham-Schroeder-Lowe protocol. The tagged protocol consists of the following messages:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B: & \{ct_0, a, pk_A\}_{pk_B} \\ \text{Message 2.} & B \rightarrow A: & \{ct_1, a, b, pk_B\}_{pk_A} \\ \text{Message 3.} & A \rightarrow B: & \{ct_2, b\}_{pk_B} \end{array}$$

Each encryption is tagged with a different tag ct_0, ct_1, and ct_2. This protocol can be represented in our calculus by the following process P:

$$\begin{aligned}
&P_A(sk_A, pk_A, pk_B) = !c(x_pk_B).(\nu a)\mathtt{event}(e_1(pk_A, x_pk_B, a)).\\
&\quad (\nu r_1)\overline{c}\langle pencrypt_p((ct_0, a, pk_A), x_pk_B, r_1)\rangle.\\
&\quad c(m).let\ (= ct_1, = a, x_b, = x_pk_B) = pdecrypt_p(m, sk_A)\ in\\
&\quad \mathtt{event}(e_3(pk_A, x_pk_B, a, x_b)).(\nu r_3)\overline{c}\langle pencrypt_p((ct_2, x_b), x_pk_B, r_3)\rangle\\
&\quad if\ x_pk_B = pk_B\ then\ \mathtt{event}(e_A(pk_A, x_pk_B, a, x_b)).\\
&\quad \overline{c}\langle sencrypt((ct_3, sAa), a)\rangle.\overline{c}\langle sencrypt((ct_4, sAb), x_b)\rangle\\
&P_B(sk_B, pk_B, pk_A) = !c(m').let\ (= ct_1, x_a, x_pk_A) = pdecrypt_p(m, sk_B)\ in\\
&\quad (\nu b)\mathtt{event}(e_2(x_pk_A, pk_B, x_a, b)).\\
&\quad (\nu r_2)\overline{c}\langle pencrypt_p((ct_2, x_a, b, pk_B), x_pk_A, r_2)\rangle.\\
&\quad c(m'').let\ (= ct_3, = b) = pdecrypt_p(m'', sk_B)\ in\\
&\quad if\ x_pk_A = pk_A\ then\ \mathtt{event}(e_B(x_pk_A, pk_B, x_a, b)).\\
&\quad \overline{c}\langle sencrypt((ct_5, sBa), x_a)\rangle.\overline{c}\langle sencrypt((ct_6, sBb), b)\rangle\\
&P_T = !c(x_1).c(x_2).\overline{c}\langle x_2\rangle.(c(x_3).c(x_4) \mid c(x_5).c(x_6))\\
&P = (\nu sk_A)(\nu sk_B)let\ pk_A = pk(sk_A)\ in\ let\ pk_B = pk(sk_B)\ in\\
&\quad \overline{c}\langle pk_A\rangle\overline{c}\langle pk_B\rangle.(P_A(sk_A, pk_A, pk_B) \mid P_B(sk_B, pk_B, pk_A) \mid P_T)
\end{aligned}$$

The encryptions that are used for testing the secrecy of nonces are also tagged, with tags ct_3 to ct_6. Furthermore, a process P_T is added in order to satisfy Condition C5, because, without P_T, in the absence of adversary, the process would block when it tries to send the public keys

pk_A and pk_B. The execution of Condition C5 is the intended execution of the protocol. In this execution, the process P_T receives the public keys pk_A and pk_B; it forwards pk_B on channel c to P_A, so that a session between A and B starts. Then A and B run this session normally, and finally output the encryptions of *sAa*, *sAb*, *sBa*, and *sBb*; these encryptions are received by P_T. The other conditions of Definition 15 are easy to check, so P is tagged.

Proposition 3 below applies to P, and also to the process without P_T, because the addition of P_T in fact does not change the clauses. (The only clause generated from P_T is a tautology, immediately removed by *elimtaut*.)

We prove the following termination result in Appendix D:

Proposition 3 *For* $\mathsf{sel} = \mathsf{sel}_0$*, the algorithm terminates on tagged protocols for queries of the form* $\alpha \rightsquigarrow \text{false}$ *when* α *is closed and all facts in* $\mathcal{F}_{\text{not}}$ *are closed.*

The proof first considers the particular case in which *pk* and *host* have a single argument in the execution of Condition C5, and then generalizes by mapping all arguments of *pk* and *host* (which are atomic constants by Condition C7) to a single constant. The proof of the particular case proceeds in two steps. The first step shows that the clauses generated from a tagged protocol satisfy the conditions of [20]. Basically, these conditions require that the clauses for the protocol satisfy the following properties:

T1. The patterns in the clauses are *tagged*, that is, the first argument of all occurrences of constructors except tuples, *pk*, and *host* is of the form $(ct, M_1, \dots, M_n)$. The proof of this property relies on Conditions C3 and C4.

T2. Let S_1 be the set of subterms of patterns that correspond to the terms that occur in the execution of Condition C5. Every clause has an instance in which all patterns are in S_1. The proof of this property relies on Condition C5.

T3. Each non-variable, non-data tagged pattern has at most one instance in S_1. (A pattern is said to be *non-data* when it is not of the form $f(\dots)$ with f a data constructor, that is, here, a tuple.) This property comes from Condition C3 which guarantees that the tags at distinct occurrences are distinct and, for $pk(p)$ and $host(p)$, from the hypothesis that *pk* and *host* have a single argument in the execution of Condition C5.

Note that the patterns in the clauses (Rf) and (Rg) that come from constructors and destructors are not tagged, so we need to handle them specially; Conditions C1 and C6 are useful for that.

The second step of the proof uses the result of [20] in order to conclude termination. Basically, this result shows that Properties T1 and T2 are preserved by resolution. The proof of this result relies on the fact that, if two non-variable non-data tagged patterns unify and have instances in S_1, then their instances in S_1 are equal (by T3). So, when unifying two such patterns, their unification still has an instance in S_1. Furthermore, we show that the size of the instance in S_1 of a clause obtained by resolution is not greater than the size of the instance in S_1 of one of the initial clauses. Hence, we can bound the size of the instance in S_1 of generated clauses, which shows that only finitely many clauses are generated.

The hypothesis that all facts in $\mathcal{F}_{\text{not}}$ are closed is not really a restriction, since we can always remove facts from $\mathcal{F}_{\text{not}}$ without changing the result. (It may just slow down the resolution.) The restriction to queries $\alpha \rightsquigarrow \text{false}$ allows us to remove m-event facts from clauses (by Remark 3). For more general queries, m-event facts may occur in clauses, and one can find examples on which the algorithm does not terminate. Here is such an example:

$$\begin{aligned}P_S = {} & c'_1(y); let\ z = sencrypt((ct_0, y), k_{SB})\ in\\ & \overline{c'_2}\langle sencrypt((ct_2, sencrypt((ct_1, z), k_{SA})), k_{SB})\rangle; \mathtt{event}(h((ct_3, y))); \overline{c'_3}\langle z\rangle\end{aligned}$$

$$P_B = c_2'(z'); c_3'(z); let\ (= ct_0, y) = sdecrypt(z, k_{SB})\ in$$
$$let\ (= ct_2, y') = sdecrypt(z', k_{SB})\ in\ \mathtt{event}(h((ct_4, y, y'))); \overline{c_4'}\langle y' \rangle$$
$$P_0 = (\nu k_{SB}); (\overline{c_1'}\langle C_0 \rangle \mid !P_S \mid !P_B \mid c_4'(y'))$$

This example has been built on purpose for exhibiting non-termination, since we did not meet such non-termination cases in our experiments with real protocols. One can interpret this example as follows. The participant A shares a key k_{SA} with a server S. Similarly, B shares a key k_{SB} with S. The code of S is represented by P_S, the code of B by P_B, and A is assumed to be dishonest, so it is represented by the adversary. The process P_S builds two tickets $sencrypt((ct_0, y), k_{SB})$ and $sencrypt((ct_2, sencrypt((ct_1, sencrypt((ct_0, y), k_{SB})), k_{SA})), k_{SB})$. The first ticket is for B, the second ticket should first be decrypted by B, then sent to A, which is going to decrypt it again and sent it back to B. In the example, P_B just decrypts the two tickets and forwards the second one to A. It is easy to check that this process is a tagged protocol. This process generates the following clauses:

$$\text{attacker}(y) \Rightarrow \text{attacker}(sencrypt((ct_2, sencrypt((ct_1, sencrypt((ct_0, y), k_{SB})), k_{SA})), k_{SB})) \tag{19}$$

$$\text{attacker}(y) \wedge \text{m-event}(h((ct_3, y))) \Rightarrow \text{attacker}(sencrypt((ct_0, y), k_{SB})) \tag{20}$$

$$\text{attacker}(sencrypt((ct_0, y), k_{SB})) \wedge \text{attacker}(sencrypt((ct_2, y'), k_{SB})) \wedge \text{m-event}(h((ct_4, y, y'))) \Rightarrow \text{attacker}(y') \tag{21}$$

$$\text{attacker}(C_0) \tag{22}$$

The first two clauses come from P_S, the third one from P_B, and the last one from the output in P_0. Obviously, clauses (Init) (in particular attacker(k_{SA}) since $k_{SA} \in fn(P_0)$), (Rf) for *sencrypt* and h, and (Rg) for *sdecrypt* are also generated. Assuming the first hypothesis is selected in (21), the solving algorithm performs a resolution step between (20) and (21), which yields:

$$\text{attacker}(y) \wedge \text{attacker}(sencrypt((ct_2, y'), k_{SB})) \wedge \text{m-event}(h((ct_3, y))) \wedge \text{m-event}(h((ct_4, y, y'))) \Rightarrow \text{attacker}(y')$$

The second hypothesis is selected in this clause. By resolving with (19), we obtain

$$\text{attacker}(y) \wedge \text{attacker}(y') \wedge \text{m-event}(h((ct_3, y))) \wedge \text{m-event}(h((ct_4, y, sencrypt((ct_1, sencrypt((ct_0, y'), k_{SB})), k_{SA})))) \Rightarrow \text{attacker}(sencrypt((ct_1, sencrypt((ct_0, y'), k_{SB})), k_{SA}))$$

By applying (Rg) for *sdecrypt* and resolving with attacker(ct_1) and attacker(k_{SA}), we obtain:

$$\text{attacker}(y) \wedge \text{attacker}(y') \wedge \text{m-event}(h((ct_3, y))) \wedge \text{m-event}(h((ct_4, y, sencrypt((ct_1, sencrypt((ct_0, y'), k_{SB})), k_{SA})))) \Rightarrow \text{attacker}(sencrypt((ct_0, y'), k_{SB}))$$

This clause is similar to (20), so we can repeat this resolution process, resolving with (21), (19), and decrypting the conclusion. Hence we obtain

$$\bigwedge_{j=1}^{n} \text{attacker}(y_j) \wedge \text{m-event}(h((ct_3, y_1))) \wedge \bigwedge_{j=1}^{n-1} \text{m-event}(h((ct_4, y_j, sencrypt((ct_1, sencrypt((ct_0, y_{j+1}), k_{SB})), k_{SA})))) \Rightarrow \text{attacker}(sencrypt((ct_0, y_n), k_{SB}))$$

for all $n > 0$, so the algorithm does not terminate.

As noticed in [20], termination could be obtained in the presence of m-event facts with an additional simplification:

Elimination of useless m-event facts: *elim-m-event* eliminates m-event facts in which a variable x occurs, and x only occurs in m-event facts and in attacker(x) hypotheses.

This simplification is always sound, because it creates a stronger clause. It does not lead to a loss of precision when all variables of events after $\rightsquigarrow$ also occur in the event before $\rightsquigarrow$. (This happens in particular for non-injective agreement.) Indeed, assume that m-event(p) contains a variable which does not occur in the conclusion. This is preserved by resolution, so when we obtain a clause m-event(p') $\wedge H \Rightarrow$ event(p''), where m-event(p') comes from m-event(p), p' contains a variable that does not occur in p'', so this occurrence of m-event(p') cannot be used to prove the desired correspondence. However, in the general case, this simplification leads to a loss of precision. (It may miss some m-event facts.) That is why this optimization was present in early implementations which verified only authentication, and was later abandoned. We could reintroduce it when all variables of events after $\rightsquigarrow$ also occur in the event before $\rightsquigarrow$, if we had termination problems coming from m-event facts for practical examples. No such problems have occurred up to now.

8.2 Choice of the Selection Function

Unfortunately, not all protocols are tagged. In particular, protocols using a Diffie-Hellman key agreement (see Section 9.1) are not tagged in the sense of Definition 15. The algorithm still terminates for some of them (Skeme [52] for secrecy, SSH) with the previous selection function sel_0. However, it does not terminate with the selection function sel_0 for some other examples (Skeme [52] for one authentication property, the Needham-Schroeder shared-key protocol [60], some versions of the Woo-Lam shared-key protocol [70] and [5, Example 6.2].) In this section, we present heuristics to improve the choice of the selection function, in order to avoid most simple non-termination cases. As reported in more detail in Section 10, these heuristics provide termination for Skeme [52] and the Needham-Schroeder shared-key protocol [60].

Let us determine which constraints the selection function should satisfy to avoid loops in the algorithm. First, assume that there is a clause $H \wedge F \Rightarrow \sigma F$, where σ is a substitution such that all $\sigma^n F$ are distinct for $n \in \mathbb{N}$.

- Assume that F is selected in this clause, and there is a clause $H' \Rightarrow F'$, where F' unifies with F, and the conclusion is selected in $H' \Rightarrow F'$. Let σ' be the most general unifier of F and F'. So the algorithm generates:

$$\sigma' H' \wedge \sigma' H \Rightarrow \sigma' \sigma F \quad \ldots \quad \sigma' H' \wedge \bigwedge_{i=0}^{n-1} \sigma' \sigma^i H \Rightarrow \sigma' \sigma^n F$$

  assuming that the conclusion is selected in all these clauses, and that no clause is removed because it is subsumed by another clause. So the algorithm would not terminate. Therefore, in order to avoid this situation, we should avoid selecting F in the clause $H \wedge F \Rightarrow \sigma F$.

- Assume that the conclusion is selected in the clause $H \wedge F \Rightarrow \sigma F$, and there is a clause $H' \wedge \sigma' F \Rightarrow C$ (up to renaming of variables), where σ' commutes with σ (in particular, when σ and σ' have disjoint supports), and that $\sigma' F$ is selected in this clause. So the algorithm generates:

$$\sigma' H \wedge \sigma H' \wedge \sigma' F \Rightarrow \sigma C \quad \ldots \quad \bigwedge_{i=0}^{n-1} \sigma' \sigma^i H \wedge \sigma^n H' \wedge \sigma' F \Rightarrow \sigma^n C$$

  assuming that $\sigma' F$ is selected in all these clauses, and that no clause is removed because it is subsumed by another clause. So the algorithm would not terminate. Therefore, in

order to avoid this situation, if the conclusion is selected in the clause $H \wedge F \Rightarrow \sigma F$, we should avoid selecting facts of the form $\sigma' F$, where σ' and σ have disjoint supports, in other clauses.

In particular, since there are clauses of the form $\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$, by the first remark, the facts $\text{attacker}(x_i)$ should not be selected in this clause. So the conclusion will be selected in this clause and, by the second remark, facts of the form $\text{attacker}(x)$ with x variable should not be selected in other clauses. We find again the constraint used in the definition of sel_0.

We also have the following similar remarks after swapping conclusion and hypothesis. Assume that there is a clause $H \wedge \sigma F \Rightarrow F$, where σ is a substitution such that all $\sigma^n F$ are distinct for $n \in \mathbb{N}$. We should avoid selecting the conclusion in this clause and, if we select σF in this clause, we should avoid selecting conclusions of the form $\sigma' F$, where σ' and σ have disjoint supports, in other clauses.

We define a selection function that takes into account all these remarks. For a clause $H \Rightarrow C$, we define the weight $w_{\text{hyp}}(F)$ of a fact $F \in H$ by:

$$w_{\text{hyp}}(F) = \begin{cases} -\infty & \text{if } F \text{ is an unselectable fact} \\ -2 & \text{if } \exists \sigma, \sigma F = C \\ -1 & \text{otherwise, if } F \in S_{\text{hyp}} \\ 0 & \text{otherwise.} \end{cases}$$

The set S_{hyp} is defined as follows: at the beginning, $S_{\text{hyp}} = \emptyset$; if we generate a clause $H \wedge F \Rightarrow \sigma F$ where σ is a substitution that maps variables of F to terms that are not all variables and, in this clause, we select the conclusion, then we add to S_{hyp} all facts $\sigma' F$ with σ and σ' of disjoint support (and renamings of these facts). For simplicity, we have replaced the condition "all $\sigma^n F$ are distinct for $n \in \mathbb{N}$" with "σ maps variables of F to terms that are not all variables". (The former implies the latter but the converse is wrong.) Our aim is only to obtain good heuristics, since there exists no perfect selection function that would provide termination in all cases. The set S_{hyp} can easily be represented finitely: just store the facts F with, for each variable, a flag indicating whether this variable can be substituted by any term by σ', or only by a variable.

Similarly, we define the weight of the conclusion:

$$w_{\text{concl}} = \begin{cases} -2 & \text{if } \exists \sigma, \exists F \in H, \sigma C = F \\ -1 & \text{otherwise, if } C \in S_{\text{concl}} \\ 0 & \text{otherwise.} \end{cases}$$

The set S_{concl} is defined as follows: at the beginning, $S_{\text{concl}} = \emptyset$; if we generate a clause $H \wedge \sigma F \Rightarrow F$ where σ is a substitution that maps variables of F to terms that are not all variables and, in this clause, we select σF, then we add to S_{concl} all facts $\sigma' F$ with σ and σ' of disjoint support (and renamings of these facts).

Finally, we define

$$\mathsf{sel}_1(H \Rightarrow C) = \begin{cases} \emptyset & \text{if } \forall F \in H, w_{\text{hyp}}(F) < w_{\text{concl}}, \\ \{F_0\} & \text{where } F_0 \in H \text{ of maximum weight, otherwise.} \end{cases}$$

Therefore, we avoid unifying facts of smallest weight when that is possible. The selected fact F_0 can be any element of H of maximum weight. In the implementation, the hypotheses are represented by a list, and the selected fact is the first element of the list of hypotheses of maximum weight.

We can also notice that the bigger the fact is, the stronger are constraints to unify it with another fact. So selecting a bigger fact should reduce the possible unifications. Therefore, we consider sel_2, defined as sel_1 except that $w_{\text{hyp}}(F) = \mathit{size}(F)$ instead of 0 in the last case.

When selecting a fact that has a negative weight, we are in one of the cases when termination will probably not be achieved. We therefore emit a warning in this case, so that the user can stop the program.

9 Extensions

In this section, we briefly sketch a few extensions to the framework presented previously. The extensions of Sections 9.1, 9.2, and 9.3 were presented in [18] for the proof of process equivalences. We sketch here how to adapt them to the proof of correspondences.

9.1 Equational Theories and Diffie-Hellman Key Agreements

Up to now, we have defined cryptographic primitives by associating rewrite rules to destructors. Another way of defining primitives is by equational theories, as in the applied pi calculus [4]. This allows us to model, for instance, variants of encryption for which the failure of decryption cannot be detected or more complex primitives such as Diffie-Hellman key agreements. The Diffie-Hellman key agreement [38] enables two principals to build a shared secret. It is used as an elementary step in more complex protocols, such as Skeme [52], SSH, SSL, and IPsec.

As shown in [18], our verifier can be extended to handle some equational theories. Basically, one shows that each trace in a model with an equational theory corresponds to a trace in a model in which function symbols are equipped with additional rewrite rules, and conversely. (We could adapt [18, Lemma 1] to show that this result also applies to correspondences.) Therefore, we can show that a correspondence proved in the model with rewrite rules implies the same correspondence in the model with an equational theory. Moreover, we have implemented algorithms that compute the rewrite rules from an equational theory.

In the experiments reported in this paper, we use equational theories only for the Diffie-Hellman key agreement, which can be modeled by using two functions f and f' that satisfy the equation

$$f(y, f'(x)) = f(x, f'(y)). \tag{23}$$

In practice, the functions are $f(x, y) = y^x \mod p$ and $f'(x) = b^x \mod p$, where p is prime and b is a generator of $\mathbb{Z}_p^*$. The equation $f(y, f'(x)) = (b^x)^y \mod p = (b^y)^x \mod p = f(x, f'(y))$ is satisfied. In our verifier, following the ideas used in the applied pi calculus [4], we do not consider the underlying number theory; we work abstractly with the equation (23). The Diffie-Hellman key agreement involves two principals A and B. A chooses a random name x_0, and sends $f'(x_0)$ to B. Similarly, B chooses a random name x_1, and sends $f'(x_1)$ to A. Then A computes $f(x_0, f'(x_1))$ and B computes $f(x_1, f'(x_0))$. Both values are equal by (23), and they are secret: assuming that the attacker cannot have x_0 or x_1, it can compute neither $f(x_0, f'(x_1))$ nor $f(x_1, f'(x_0))$.

In our verifier, the equation (23) is translated into the rewrite rules

$$f(y, f'(x)) \rightarrow f(x, f'(y)) \quad f(x, y) \rightarrow f(x, y).$$

Notice that this definition of f is non-deterministic: a term such as $f(a, f'(b))$ can be reduced to $f(b, f'(a))$ and $f(a, f'(b))$, so that $f(a, f'(b))$ reduces to its two forms modulo the equational theory. The fact that these rewrite rules model the equation (23) correctly follows from [18, Section 5].

When using this model, we have to adapt the verification of correspondences. Indeed, the conditions on the clauses must be checked *modulo the equational theory.* (Using the rewrite rules, we can implement unification modulo the equational theory, basically by rewriting the terms by the rewrite rules before performing syntactic unification.) For example, in the case of non-injective agreement, even if the process P_0 satisfies non-injective agreement against *Init*-adversaries, it may happen that a clause $\text{m-event}(e'(p_1, \dots, p_n)\{f(p_2, f'(p_1))/z\}) \Rightarrow \text{event}(e(p_1,$

$\dots, p_n)\{f(p_1, f'(p_2))/z\})$ is in $\mathsf{solve}_{P'_0,Init}(\text{event}(e(x_1,\dots,x_n)))$. The specification is still satisfied in this case, because $(p_1,\dots,p_n)\{f(p_1, f'(p_2))/z\} = (p_1,\dots,p_n)\{f(p_2, f'(p_1))/z\}$ modulo the equational theory. So we have to test that, if $H \Rightarrow \text{event}(e(p_1,\dots,p_n))$ is in $\mathsf{solve}_{P'_0,Init}(\text{event}(e(x_1,\dots,x_n)))$, then there exist $p'_1,\dots,p'_n$ equal to $p_1,\dots,p_n$ modulo the equational theory such that $\text{m-event}(e'(p'_1,\dots,p'_n)) \in H$. More generally, the equality $R = H \wedge \text{m-event}(\sigma' p_{j1}) \wedge \dots \wedge \text{m-event}(\sigma' p_{jl_j}) \Rightarrow \text{event}(\sigma' p'_j)$ in the hypothesis of Theorem 3 is checked modulo the equational theory (using matching modulo the equational theory to find σ'). Point V2.1 of the definition of verify and Hypothesis H2 of Theorem 5 are also checked modulo the equational theory. Furthermore, the following condition is added to Point V2.2 of the definition of verify:

> For all j, r, and k, we let $q_c = \sigma_{jr} q_{jk}$ and $p_c = \sigma_{jr} p_{jk}$, and we require that, for all substitutions σ and σ', if $\sigma p_c = \sigma' p_c$ and for all $x \in fv(q_c) \setminus fv(p_c)$, $\sigma x = \sigma' x$, then $\sigma q_c = \sigma' q_c$ (where equalities are considered modulo the equational theory).

This property is useful in the proof of Theorem 5 (see Appendix E). It always holds when the equational theory is empty, because $\sigma p_c = \sigma' p_c$ implies that for all $x \in fv(p_c)$, $\sigma x = \sigma' x$, so for all $x \in fv(q_c)$, $\sigma x = \sigma' x$. However, it does not hold in general for any equational theory, so we need to check it explicitly when the equational theory is non-empty. In the implementation, this condition is checked as follows. Let θ be a renaming of variables of p_c to fresh variables. We check that, for every σ_u most general unifier of p_c and θp_c modulo the equational theory, $\sigma_u q_c = \sigma_u \theta q_c$ modulo the equational theory. When this check succeeds, we can prove the condition above as follows. Let σ_0 be defined by, for all $x \in fv(q_c)$, $\sigma_0 x = \sigma x$ and, for all $x \in fv(\theta p_c)$, $\sigma_0 x = \sigma' \theta^{-1} x$. If $\sigma p_c = \sigma' p_c$, then $\sigma_0 p_c = \sigma p_c = \sigma' p_c = \sigma_0 \theta p_c$, so σ_0 unifies p_c and θp_c, hence there exist σ_1 and a most general unifier σ_u of p_c and θp_c such that $\sigma_0 = \sigma_1 \sigma_u$. We have $\sigma_u q_c = \sigma_u \theta q_c$, so $\sigma q_c = \sigma_0 q_c = \sigma_1 \sigma_u q_c = \sigma_1 \sigma_u \theta q_c = \sigma_0 \theta q_c = \sigma' q_c$.

This treatment of equations has the advantage that resolution can still use syntactic unification, so it remains efficient. However, it also has limitations; for example, it cannot handle associative functions, such as XOR, because it would generate an infinite number of rewrite rules for the destructors. We refer to [28, 31] for treatments of XOR and to [27, 48, 56, 58] for treatments of Diffie-Hellman key agreements with more detailed algebraic relations. The NRL protocol analyzer handles a limited version of associativity for strings of bounded length [43], which we could handle.

9.2 Precise Treatment of *else* Branches

In the generation of clauses described in Section 5.2, we consider that the *else* branch of destructor applications may always be executed. Our implementation takes into account these *else* branches more precisely. In order to do that, it uses a set of special variables $GVar$ and a predicate nounif, also used in [18], such that, for all closed patterns p and p', $\text{nounif}(p, p')$ holds if and only if there is no closed substitution σ with domain $GVar$ such that $\sigma p = \sigma p'$. The fact $\text{nounif}(p, p')$ means that $p \neq p'$ for all values of the special variables in $GVar$.

One can then check the failure of an equality test $M = M'$ by $\text{nounif}(\rho(M), \rho(M'))$ and the failure of a destructor application $g(M_1,\dots,M_n)$ by $\bigwedge_{g(p_1,\dots,p_n)\to p \in \text{def}(g)} \text{nounif}((\rho(M_1),\dots,\rho(M_n)), GVar(p_1,\dots,p_n))$, where $GVar(p)$ is the pattern p after renaming all its variables to elements of $GVar$ and ρ is the environment that maps variables to their corresponding patterns. Intuitively, the rewrite rule $g(p_1,\dots,p_n) \to p$ can be applied if and only if $(\rho(M_1),\dots,\rho(M_n))$ is an instance of $(p_1,\dots,p_n)$. So the rewrite rule $g(p_1,\dots,p_n) \to p$ cannot be applied if and only if $\text{nounif}((\rho(M_1),\dots,\rho(M_n)), GVar(p_1,\dots,p_n))$.

The predicate nounif is handled by specific simplification steps in the solver, described and proved correct in [18].

9.3 Scenarios with Several Stages

Some protocols can be broken into several parts, or stages, numbered 0, 1, ..., such that when the protocol starts, stage 0 is executed; at some point in time, stage 0 stops and stage 1 starts; later, stage 1 stops and stage 2 starts, and so on. Therefore, stages allow us to model a global clock. Our verifier can be extended to such scenarios with several stages, as summarized in [18]. We add a construct $t : P$ to the syntax of processes, which means that process P runs only in stage t, where t is an integer.

The generation of clauses can easily be extended to processes with stages. We use predicates attacker_t and message_t for each stage t, generate the clauses for the attacker for each stage, and the clauses for the protocol with predicates attacker_t and message_t for each process that runs in stage t. Furthermore, we add clauses

$$\text{attacker}_t(x) \Rightarrow \text{attacker}_{t+1}(x) \tag{Rt}$$

in order to transmit attacker knowledge from each stage t to the next stage $t+1$.

Scenarios with several stages allow us to model properties related to the compromise of keys. For example, we can model forward secrecy properties as follows. Consider a public-key protocol P (without stage prefix) and the process $P' = 0 : P \mid 1 : \overline{c}\langle sk_A \rangle; \overline{c}\langle sk_B \rangle$, which runs P in stage 0 and later outputs the secret keys of A and B on the public channel c in stage 1. If we prove that P' preserves the secrecy of the session keys of P, then the attacker cannot obtain these session keys even if it later compromises the private keys of A and B, which is forward secrecy.

9.4 Compromise of Session Keys

We consider the situation in which the attacker compromises some session keys of the protocol. Our goal is then to show that the other session keys of the protocol are still safe. For example, this property does not hold for the Needham-Schroeder shared-key protocol [60]: in this protocol, when an attacker manages to get some session keys, then it can also get the secrets of other sessions.

If we assume that the compromised sessions are all run before the standard sessions (to model that the adversary needs time to break the session keys before being able to use the obtained information against standard sessions), then this can be modeled as a scenario with two stages: in stage 0, the process runs a modified version of the protocol that outputs its session keys; in stage 1, the standard sessions runs; we prove the security of the sessions of stage 1.

However, we can also consider a stronger model, in which the compromised sessions may run in parallel with the non-compromised ones. In this case, we have a single stage.

Let P_0 be the process representing the whole protocol. We consider that the part of P_0 not under replications corresponds to the creation of long-term secrets, and the part of P_0 under at least one replication corresponds to the sessions. We say that the names generated under at least one replication in P_0 are *session names.* We add one argument i_c to the function symbols $a[\ldots]$ that encode session names in the instrumented process P_0'; this additional argument is named *compromise identifier* and can take two values, s_0 or s_1. We consider that, during the execution of the protocol, each replicated subprocess $!Q_X$ of P_0 generates two sets of copies of Q_X, one with compromise identifier s_0, one with s_1. The attacker compromises sessions that involve only copies of processes Q_X with the compromise identifier s_0. It does not compromise sessions that involve at least one copy of some process Q_X with compromise identifier s_1.

The clauses for the process P_0 are generated as in Section 5.2 (except for the addition of a variable compromise identifier as argument of session names). The following clauses are added:

For each constructor f, $\text{comp}(x_1) \wedge \ldots \wedge \text{comp}(x_k) \Rightarrow \text{comp}(f(x_1, \ldots, x_k))$

For each $(\nu a : a[\ldots])$ under n replications and k inputs and non-deterministic destructor applications in P'_0,

$$\begin{array}{ll} \text{comp}(x_1) \wedge \ldots \wedge \text{comp}(x_k) \Rightarrow \text{comp}(a[x_1, \ldots, x_k]) & \text{if } n = 0 \\ \text{comp}(x_1) \wedge \ldots \wedge \text{comp}(x_k) \Rightarrow \text{comp}(a[x_1, \ldots, x_k, i_1, \ldots, i_n, s_0]) & \text{if } n > 0 \\ \text{comp}(x_1) \wedge \ldots \wedge \text{comp}(x_k) \Rightarrow \text{attacker}(a[x_1, \ldots, x_k, i_1, \ldots, i_n, s_0]) & \text{if } n > 0 \end{array}$$

The predicate comp is such that comp(p) is true when all session names in p have compromise identifier s_0. These clauses express that the attacker has the session names that contain only the compromise identifier s_0.

In order to prove the secrecy of a session name s, we query the fact attacker($s[x_1, \ldots, x_k, i_1, \ldots, i_n, s_1]$). If this fact is underivable, then the protocol does not have the weakness of the Needham-Schroeder shared-key protocol mentioned above: the attacker cannot have the secret s of a session that it has not compromised. In contrast, attacker($s[x_1, \ldots, x_k, i_1, \ldots, i_n, s_0]$) is always derivable, since the attacker has compromised the sessions with identifier s_0.

We can also prove correspondences in the presence of key compromise. We want to prove that the non-compromised sessions are secure, so we prove that, if an event $\texttt{event}(M)$ has been executed in a copy of some Q_X with compromise identifier s_1, then the required events $\texttt{event}(M_{\overline{jk}})$ have been executed in any process. (A copy of Q_X with compromise identifier s_1 may interact with a copy of Q_Y with compromise identifier s_0 and, in this case, the events $\texttt{event}(M_{\overline{jk}})$ may be executed in the copy of Q_Y with compromise identifier s_0.) We obtain this result by adding the compromise identifier i_c as argument of the predicates m-event and event in clauses, and correspondingly adding s_1 as argument of $\texttt{event}(M)$ and $\texttt{event}(M_j)$, and a fresh variable as argument of the other events $\texttt{event}(M_{\overline{jk}})$ in queries. We can then prove the correspondence in the same way as in the absence of key compromise. The treatment of correspondences attacker(M) $\rightsquigarrow \ldots$ and message(M, M') $\rightsquigarrow \ldots$ in which M and M' do not contain bound names remains unchanged.

10 Experimental Results

We have implemented our verifier in Ocaml and have performed tests on various protocols of the literature. The tests reported here concern secrecy and authentication properties for simple examples of protocols. More complex examples have been studied, using our technique for proving correspondences. We do not detail them in this paper, because they have been the subject of specific papers [2, 3, 19].

Our results are summarized in Figure 6, with references to the papers that describe the protocols and the attacks. In these tests, the protocols are fully modeled, including interaction with the server for all versions of the Needham-Schroeder, Woo-Lam shared key, Denning-Sacco, Otway-Rees, and Yahalom protocols. The first column indicates the name of the protocol; we use the following abbreviations: NS for Needham-Schroeder, PK for public-key, SK for shared-key, corr. for corrected, tag. for tagged, unid. for unidirectional, and bid. for bidirectional. We have tested the Needham-Schroeder shared key protocol with the modeling of key compromise mentioned in Section 9.4, in which the compromised sessions can be executed in parallel with the non-compromised ones (version marked "comp." in Figure 6). The second column indicates the number of Horn clauses that represent the protocol. The third column indicates the total number of resolution steps performed for analyzing the protocol.

The fourth column gives the execution time of our analyzer, in ms, on a Pentium M 1.8 GHz. Several secrecy and agreement specifications are checked for each protocol. The time given is the total time needed to check all specifications. The following factors influence the speed of the system:

- We use secrecy assumptions to speed up the search. These assumptions say that the

Protocol	# cl.	# res. steps	Time (ms)	Cases with attacks Secrecy	*A*	*B*	Ref.
NS PK [60]	32	1988	95	Nonces *B*	None	All	[53]
NS PK corr. [53]	36	1481	51	None	None	None	
Woo-Lam PK [70]	23	104	7			All	[40]
Woo-Lam PK corr. [72]	27	156	6			None	
Woo-Lam SK [46]	25	184	8			All	[8]
Woo-Lam SK corr. [46]	21	244	4			None	
Denning-Sacco [37]	30	440	18	Key *B*		All	[5]
Denning-Sacco corr. [5]	30	438	16	None		Inj	
NS SK [60], tag.	31	2721	41	None	None	None	
NS SK corr. [61], tag.	32	2102	57	None	None	None	
NS SK [60], tag., comp.	50	25241	167	Key *B*	None	Inj	[37]
NS SK corr. [61], tag., comp.	53	23956	225	None	None	None	
Yahalom [26]	26	1515	34	None	Key	None	
Simpler Yahalom [26], unid.	21	1479	30	None	Key	None	
Simpler Yahalom [26], bid.	24	3685	91	None	All	None	[67]
Otway-Rees [62]	34	1878	59	None	Key	Inj,Key	[26]
Simpler Otway-Rees [5]	28	1934	31	None	All	All	[63]
Otway-Rees, variant of [63]	35	3349	87	Key *B*	All	All	[63]
Main mode of Skeme [52]	39	4139	154	None	None	None	

Figure 6: Experimental results

secret keys of the principals, and the random values of the Diffie-Hellman key agreement in the Skeme protocol, remain secret. On average, the verifier is two times slower without secrecy assumptions, in our tests.

- We mentioned several selection functions, and the speed of the system can vary substantially depending on the selection function. In the tests of Figure 6, we used the selection function sel_2. With sel_1, the system is two times slower on average on Needham-Schroeder shared-key, Otway-Rees, the variant of [63] of Otway-Rees, and Skeme but faster on the bidirectional simplified Yahalom (59 ms instead of 91 ms). The speed is almost unchanged for our other tests. On average, the verifier is 1.8 times slower with sel_1 than with sel_2, in our tests.

  The selection function sel_0 gives approximately the same speed as sel_1, except for Skeme, for which the analysis does not terminate with sel_0. (We comment further on termination below.)

- The tests of Figure 6 have been performed without elimination of redundant hypotheses. With elimination of redundant hypotheses that contain m-event facts, we obtain approximately the same speed. With elimination of all redundant hypotheses, the verifier is 1.3 times slower on average in these tests, because of the time spent testing whether hypotheses are redundant.

When our tool successfully proves that a protocol satisfies a certain specification, we are sure that this specification indeed holds, by our soundness theorems. When our tool does not manage to prove that a protocol satisfies a certain specification, it finds at least one clause and a derivation of this clause that contradicts the specification. The existence of such a clause does not prove that there is an attack: it may correspond to a false attack, due to the approximations introduced by the Horn clause model. However, using an extension of the technique of [6] to events, in most cases, our tool reconstructs a trace of the protocol, and thus proves that

there is actually an attack against the considered specification. In the tests of Figure 6, this reconstruction succeeds in all cases for secrecy and non-injective correspondences, in the absence of key compromise. The trace reconstruction is not implemented yet in the presence of key compromise (Section 9.4) or for injective correspondences. (It presents additional difficulties in the latter case, since the trace should execute some event twice and others once in order to contradict injectivity, while the derivation corresponds to the execution of events once, with badly related session identifiers.) In the cases in which trace reconstruction is not implemented, we have checked manually that the protocol is indeed subject to an attack, so our tool found no false attack in the tests of Figure 6: for all specifications that hold, it has proved them.

The last four columns give the results of the analysis. The column "Secrecy" concerns secrecy properties, the column A concerns agreement specifications $\text{event}(e(x_1, \dots, x_n)) \rightsquigarrow$ [inj] $\text{event}(e'(x_1, \dots, x_n))$ in which A executes the event $\texttt{event}(e(M_1, \dots, M_n))$, the column B agreement specifications $\text{event}(e(x_1, \dots, x_n)) \rightsquigarrow$ [inj] $\text{event}(e'(x_1, \dots, x_n))$ in which B executes the event $\texttt{event}(e(M_1, \dots, M_n))$. The last column gives the reference of the attacks when attacks are found. The first six protocols of Figure 6 (Needham-Schroeder public key and Woo-Lam one-way authentication protocols) are authentication protocols. For them, we have tested non-injective and recent injective agreement on the name of the participants, and non-injective and injective full agreement (agreement on all atomic data). For the Needham-Schroeder public key protocol, we have also tested the secrecy of nonces. "Nonces B" means that the nonces N_a and N_b manipulated by B may not be secret, "None" means all tested specifications are satisfied (there is no attack), "All" that our tool finds an attack against all tested specifications. The Woo and Lam protocols are *one-way* authentication protocols: they are intended to authenticate A to B, but not B to A, so we have only tested them with B containing $\texttt{event}(e(M_1, \dots, M_n))$.

Numerous versions of the Woo and Lam shared-key protocol have been published in the literature [70], [8], [5, end of Example 3.2], [5, Example 6.2], [72], [46] (flawed and corrected versions). Our tool terminates and proves the correctness of the corrected versions of [8] and of [46]; it terminates and finds an attack on the flawed version of [46]. (The messages received or sent by A do not depend on the host A wants to talk to, so A may start a session with the adversary C, and the adversary can reuse the messages of this session to talk to B in A's name.) We can easily see that the versions of [70] and [5, Example 6.2] are also subject to this attack, even if our tool does not terminate on them. The only difference between the protocol of [46] and that of [70] is that [46] adds tags to distinguish different encryption sites. As shown in Section 8.1, adding tags enforces termination. Our tool finds the attack of [29, bottom of page 52] on the versions of [5, end of Example 3.2] and [72]. For example, the version of [72] is

Message 1.	$A \rightarrow B$:	A
Message 2.	$B \rightarrow A$:	N_B
Message 3.	$A \rightarrow B$:	$\{A, B, N_B\}_{K_{AS}}$
Message 4.	$B \rightarrow S$:	$\{A, B, \{A, B, N_B\}_{K_{AS}}\}_{K_{BS}}$
Message 5.	$S \rightarrow B$:	$\{A, B, N_B\}_{K_{BS}}$

and the attack is

Message 1.	$I(A) \rightarrow B$:	A
Message 2.	$B \rightarrow I(A)$:	N_B
Message 3.	$I(A) \rightarrow B$:	N_B
Message 4.	$B \rightarrow I(A)$:	$\{A, B, N_B\}_{K_{BS}}$
Message 5.	$I(A) \rightarrow B$:	$\{A, B, N_B\}_{K_{BS}}$

In message 3, the adversary sends N_B instead of $\{A, B, N_B\}_{K_{AS}}$. B cannot see the difference and, acting as defined in the protocol, B unfortunately sends exactly the message needed by the adversary as message 5. So B thinks he talks to A, while A and S can perfectly be dead. The attack found against the version of [5, end of Example 3.2] is very similar.

The last five protocols exchange a session key, so we have tested agreement on the names of the participants, and agreement on both the participants and the session key (instead of full agreement, since agreement on the session key is more important than agreement on other values). In Figure 6, "Key B" means that the key obtained by B may not be secret, "Key" means that agreement on the session key is wrong, "Inj" means that injective agreement is wrong, "All" and "None" are as before.

In the Needham-Schroeder shared key protocol [60], the last messages are

$$\begin{array}{lll} \text{Message 4.} & B \to A: & \{N_B\}_K \\ \text{Message 5.} & A \to B: & \{N_B - 1\}_K \end{array}$$

where N_B is a nonce. Representing $N_B - 1$ with a function $\mathsf{minusone}(x) = x - 1$, with associated destructor $\mathsf{plusone}$ defined by $\mathsf{plusone}(\mathsf{minusone}(x)) \to x$, the algorithm does not terminate with the selection function sel_0. The selection functions sel_1 or sel_2 given in Section 8.2 however yield termination. We can also notice that the purpose of the subtraction is to distinguish the reply of A from B's message. As mentioned in [5], it would be clearer to have:

$$\begin{array}{lll} \text{Message 4.} & B \to A: & \{\text{Message 4} : N_B\}_K \\ \text{Message 5.} & A \to B: & \{\text{Message 5} : N_B\}_K \end{array}$$

We have used this encoding in the tests shown in Figure 6. Our tool then terminates with selection functions sel_0, sel_1, and sel_2. [20] explains in more detail why these two messages encoded with $\mathsf{minusone}$ prevent termination with sel_0, and why the addition of tags "Message 4", "Message 5" yields termination. Adding the tags may strengthen the protocol (for instance, in the Needham-Schroeder shared key protocol, it prevents replaying Message 5 as a Message 4), so the security of the tagged version does not imply the security of the original version. As mentioned in [5], using the tagged version is a better design choice because it prevents confusing different messages, so this version should be implemented. Our tool also does not terminate on Skeme with selection function sel_0, for an authentication query, but terminates with selection functions sel_1 or sel_2. All other examples of Figure 6 terminate with the three selection functions sel_0, sel_1, and sel_2.

Among the examples of Figure 6, only the Woo-Lam shared key protocol, flawed and corrected versions of [46] and the Needham-Schroeder shared key protocol have explicit tags. Our tool terminates on all other protocols, even if they are not tagged. The termination can partly be explained by the notion of "implicitly tagged" protocols [20]: the various messages are not distinguished by explicit tags, but by other properties of their structure, such as the arity of the tuples that they contain. In Figure 6, the Denning-Sacco protocol and the Woo-Lam public key protocol are implicitly tagged. Still, the tool terminates on many examples that are not even implicitly tagged.

For the Yahalom protocol, we show that, if B thinks that k is a key to talk with A, then A also thinks that k is a key to talk with B. The converse is clearly wrong, because the session key is sent from A to B in the last message, so the adversary can intercept this message, so that A has the key but not B.

For the Otway-Rees protocol, we do not have agreement on the session key, since the adversary can intercept messages in such a way that one participant has the key and the other one has no key. There is also an attack in which both participants get a key, but not the same one [44]. The latter attack is not found by our tool, since it stops with the former attacks.

For the simplified version of the Otway-Rees protocol given in [5], B can execute its event $\mathtt{event}(e(M_1, \dots, M_n))$ with A dead, and A can execute its event $\mathtt{event}(e(M_1, \dots, M_n))$ with B dead. As Burrows, Abadi, and Needham already noted in [26], even the original protocol does not guarantee to B that A is alive (attack against injective agreement that we also find). [46] said that the protocol satisfied its authentication specifications, because they showed that neither A nor B can conclude that k is a key for talking between A and B without the server

first saying so. (Of course, this property is also important, and could also be checked with our verifier.)

11 Conclusion

We have extended previous work on the verification of security protocols by logic programming techniques, from secrecy to a very general class of correspondences, including not only authentication but also, for instance, correspondences that express that the messages of the protocol have been sent and received in the expected order. This technique enables us to check correspondences in a fully automatic way, without bounding the number of sessions of the protocols. This technique also yields an efficient verifier, as the experimental results demonstrate.

Acknowledgments

We would like to thank Martín Abadi, Jérôme Feret, Cédric Fournet, and Andrew Gordon for helpful discussions on this paper. This work was partly done at Max-Planck-Institut für Informatik, Saarbrücken, Germany.

References

[1] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[2] M. Abadi and B. Blanchet. Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, 58(1–2):3–27, Oct. 2005. Special issue SAS'03.

[3] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*, 10(3):1–59, July 2007.

[4] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, London, United Kingdom, Jan. 2001. ACM Press.

[5] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[6] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005. IEEE.

[7] R. Amadio and S. Prasad. The game of the name in cryptographic tables. In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science - ASIAN'99*, volume 1742 of *Lecture Notes on Computer Science*, pages 15–27, Phuket, Thailand, Dec. 1999. Springer.

[8] R. Anderson and R. Needham. Programming Satan's computer. In J. van Leeuven, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes on Computer Science*, pages 426–440. Springer, 1995.

[9] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 19–100. North Holland, 2001.

[10] M. Backes, A. Cortesi, and M. Maffei. Causality-based abstraction of multiplicity in security protocols. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 355–369, Venice, Italy, July 2007. IEEE.

[11] M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO 1993*, volume 773 of *Lecture Notes on Computer Science*, pages 232–249, Santa Barbara, California, Aug. 1993. Springer.

[12] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *Lecture Notes on Computer Science*, pages 197–222, Leiden, The Netherlands, Nov. 2003. Springer. Paper and tool available at `http://securing.ws/`.

[13] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[14] B. Blanchet. From secrecy to authenticity in security protocols. In M. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, Sept. 2002. Springer.

[15] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[16] B. Blanchet. Automatic proof of strong secrecy for security protocols. Technical Report MPI-I-2004-NWG1-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany, July 2004.

[17] B. Blanchet. Security protocols: From linear to classical logic by abstract interpretation. *Information Processing Letters*, 95(5):473–479, Sept. 2005.

[18] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, Feb.–Mar. 2008.

[19] B. Blanchet and A. Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Symposium on Security and Privacy*, pages 417–431, Oakland, CA, May 2008. IEEE.

[20] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, Mar. 2005. Special issue FoSSaCS'03.

[21] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[22] P. Broadfoot, G. Lowe, and B. Roscoe. Automating data independence. In *6th European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *Lecture Notes on Computer Science*, pages 175–190, Toulouse, France, Oct. 2000. Springer.

[23] P. J. Broadfoot and A. W. Roscoe. Embedding agents within the intruder to detect parallel attacks. *Journal of Computer Security*, 12(3/4):379–408, 2004.

[24] M. Bugliesi, R. Focardi, and M. Maffei. Analysis of typed analyses of authentication protocols. In *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 112–125, Aix-en-Provence, France, June 2005. IEEE Comp. Soc. Press.

[25] M. Bugliesi, R. Focardi, and M. Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[26] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

[27] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. Deciding the security of protocols with Diffie-Hellman exponentiation and products in exponents. In P. K. Pandya and J. Radhakrishnan, editors, *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference*, volume 2914 of *Lecture Notes on Computer Science*, pages 124–135, Mumbai, India, Dec. 2003. Springer.

[28] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An NP decision procedure for protocol insecurity with XOR. *Theoretical Computer Science*, 338(1–3):247–274, June 2005.

[29] J. Clark and J. Jacob. A survey of authentication protocol literature: Version1.0. Technical report, University of York, Department of Computer Science, Nov. 1997.

[30] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[31] H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In *Symposium on Logic in Computer Science (LICS'03)*, pages 271–280, Ottawa, Canada, June 2003. IEEE Computer Society.

[32] V. Cortier, J. Millen, and H. Rueß. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 97–108, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[33] C. J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Ph.D. dissertation, Eindhoven University of Technology, Nov. 2006.

[34] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.

[35] H. de Nivelle. *Ordering Refinements of Resolution.* PhD thesis, Technische Universiteit Delft, Oct. 1995.

[36] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. A new algorithm for the automatic verification of authentication protocols: From specifications to flaws and attack scenarios. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, New Jersey, Sept. 1997.

[37] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, Aug. 1981.

[38] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov. 1976.

[39] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, Mar. 1983.

[40] A. Durante, R. Focardi, and R. Gorrieri. CVS at work: A report on new failures upon some cryptographic protocols. In V. Gorodetski, V. Skormin, and L. Popyack, editors, *Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS'01)*,

volume 2052 of *Lecture Notes on Computer Science*, pages 287–299, St. Petersburg, Russia, May 2001. Springer.

[41] N. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[42] S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.

[43] S. Escobar, C. Meadows, and J. Meseguer. Equational cryptographic reasoning in the Maude-NRL protocol analyzer. *Electronic Notes in Theoretical Computer Science*, 171(4):23–36, July 2007.

[44] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[45] A. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In M. Okada, B. Pierce, A. Scedriv, H. Tokuda, and A. Yonezawa, editors, *Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium, ISSS 2002*, volume 2609 of *Lecture Notes on Computer Science*, pages 263–282, Tokyo, Japan, Nov. 2002. Springer.

[46] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

[47] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2004.

[48] J. Goubault-Larrecq, M. Roger, and K. N. Verma. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 64(2):219–251, Aug. 2005.

[49] J. D. Guttman and F. J. T. Fábrega. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2002.

[50] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 255–268, Cambridge, England, July 2000.

[51] J. Heather and S. Schneider. A decision procedure for the existence of a rank function. *Journal of Computer Security*, 13(2):317–344, 2005.

[52] H. Krawczyk. SKEME: A versatile secure key exchange mechanism for internet. In *Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

[53] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes on Computer Science*, pages 147–166. Springer, 1996.

[54] G. Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop (CSFW '97)*, pages 31–43, Rockport, Massachusetts, June 1997. IEEE Computer Society.

[55] C. Lynch. Oriented equational logic programming is complete. *Journal of Symbolic Computation*, 21(1):23–45, 1997.

[56] C. Meadows and P. Narendran. A unification algorithm for the group Diffie-Hellman protocol. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, Jan. 2002.

[57] C. A. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[58] J. Millen and V. Shmatikov. Symbolic protocol analysis with an abelian group operator or Diffie-Hellman exponentiation. *Journal of Computer Security*, 13(3):515–564, 2005.

[59] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *1997 IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[60] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

[61] R. M. Needham and M. D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.

[62] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[63] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[64] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2, 3):147–190, 1999.

[65] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. *Theoretical Computer Science*, 299(1–3):451–475, Apr. 2003.

[66] D. X. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[67] P. Syverson. A taxonomy of replay attacks. In *7th IEEE Computer Security Foundations Workshop (CSFW-94)*, pages 131–136, Franconia, New Hampshire, June 1994. IEEE Computer Society.

[68] P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1/2):27–59, 1996.

[69] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–328, Trento, Italy, July 1999. Springer.

[70] T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, Jan. 1992.

[71] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 1993.

[72] T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. In D. Denning and P. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 319–355. ACM Press and Addison-Wesley, Oct. 1997.

Appendices

A Instrumented Processes

Let $\mathrm{last}(s)$ be the last element of the sequence of session identifiers s, or $\emptyset$ when $s = \emptyset$. Let $\mathit{label}(\ell)$ be defined by $\mathit{label}(a[t,s]) = (a, \mathrm{last}(s))$ and $\mathit{label}(b_0[a[s]]) = (a, \mathrm{last}(s))$. We define the multiset $\mathit{Label}(P)$ as follows: $\mathit{Label}((\nu a : \ell)P) = \{\mathit{label}(\ell))\} \cup \mathit{Label}(P)$, $\mathit{Label}(!^i P) = \emptyset$, and in all other cases, $\mathit{Label}(P)$ is the union of the $\mathit{Label}(P')$ for all immediate subprocesses P' of P. Let $\mathit{Label}(E) = \{\mathit{label}(E(a)) \mid a \in \mathit{dom}(E)\}$ and $\mathit{Label}(S) = \{(a, \lambda) \mid \lambda \in S, a$ any name function symbol$\}$.

Definition 16 An *instrumented semantic configuration* is a triple $S, E, \mathcal{P}$ such that S is a countable set of constant session identifiers, the environment E is a mapping from names to closed patterns, and $\mathcal{P}$ is a multiset of closed processes. The instrumented semantic configuration is $S, E, \mathcal{P}$ *well-labeled* when the multiset $\mathit{Label}(S) \cup \mathit{Label}(E) \cup \bigcup_{P \in \mathcal{P}} \mathit{Label}(P)$ contains no duplicates.

Lemma 5 *Let P_0 be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let Q be an Init-adversary and $Q' = \mathrm{instrAdv}(Q)$. Let E_0 such that $\mathit{fn}(P'_0) \cup \mathit{Init} \subseteq \mathit{dom}(E_0)$ and, for all $a \in \mathit{dom}(E_0)$, $E_0(a) = a[\,]$. The configuration $S_0, E_0, \{P'_0, Q'\}$ is a well-labeled instrumented semantic configuration.*

Proof We have $\mathit{Label}(E_0) = \{(a, \emptyset) \mid a \in \mathit{dom}(E_0)\}$, $\mathit{Label}(P'_0) = \{(a, \emptyset) \mid (\nu a : a[\ldots])$ occurs in P'_0 not under a replication$\}$, and $\mathit{Label}(Q') = \{(a, \emptyset) \mid (\nu a : b_0[a[\,]])$ occurs in Q' not under a replication$\}$. These multisets contain no duplicates since the bound names of P'_0 and Q' are pairwise distinct and distinct from names in $\mathit{dom}(E_0)$. So the multiset $\mathit{Label}(S_0) \cup \mathit{Label}(E_0) \cup \mathit{Label}(P'_0) \cup \mathit{Label}(Q')$ contains no duplicates. □

Lemma 6 *If $S, E, \mathcal{P}$ is a well-labeled instrumented semantic configuration and $S, E, \mathcal{P} \rightarrow S', E', \mathcal{P}'$ then $S', E', \mathcal{P}'$ is a well-labeled instrumented semantic configuration.*

Proof We proceed by cases on the reduction $S, E, \mathcal{P} \rightarrow S', E', \mathcal{P}'$. The rule (Red Repl) removes the labels (a, λ) for a certain λ from $\mathit{Label}(S)$ and adds some of them to $\mathit{Label}(\mathcal{P})$. The rule (Red Res) removes a label from $\mathit{Label}(\mathcal{P})$ and adds it to $\mathit{Label}(E)$. Other rules can remove labels when they remove a subprocess, but they do not add labels. □

Lemma 7 *Let $S, E, \mathcal{P}$ be an instrumented semantic configuration. Let σ be a substitution and σ' be defined by $\sigma' x = E(\sigma x)$ for all x. For all terms M, $E(\sigma M) = \sigma' E(M)$ and, for all atoms α, $E(\sigma \alpha) = \sigma' E(\alpha)$.*

Proof We prove the result for terms M by induction on M.

- If $M = x$, $E(\sigma x) = \sigma' x = \sigma' E(x)$ by definition of σ'.
- If $M = a$, $E(\sigma a) = E(a) = \sigma' E(a)$, since $E(a)$ is closed.
- If M is a composite term $M = f(M_1, \ldots, M_n)$, $E(\sigma M) = f(E(\sigma M_1), \ldots, E(\sigma M_n)) = f(\sigma' E(M_1), \ldots, \sigma' E(M_n)) = \sigma' E(M)$, by induction hypothesis.

The extension to atoms is similar to the case of composite terms. □

Lemma 8 *If $S, E, \mathcal{P}$ is a well-labeled instrumented semantic configuration, M and M' are closed terms, and $E(M) = E(M')$, then $M = M'$.*

Proof The multiset $\mathit{Label}(E)$ does not contain duplicates, hence different names in E have different associated patterns, therefore different terms have different associated patterns. □

Lemma 9 *If $S, E, \mathcal{P}$ is a well-labeled instrumented semantic configuration, M' is a closed term, and $E(M') = \sigma E(M)$, then there exists a substitution σ' such that $M' = \sigma' M$ and, for all variables x of M, $E(\sigma' x) = \sigma x$. We have a similar result for atoms and for tuples containing terms and atoms.*

Proof We prove the result for terms by induction on M.

- If $M = x$, $E(M') = \sigma E(M) = \sigma x$. We define σ' by $\sigma' x = M'$.
- If M is a name, $E(M)$ is closed, so $E(M') = \sigma E(M) = E(M)$. By Lemma 8, $M' = M = \sigma' M$ for any substitution σ'.
- If M is a composite term $M = f(M_1, \ldots, M_n)$, $E(M') = f(\sigma E(M_1), \ldots, \sigma E(M_n))$. Therefore, $M' = f(M'_1, \ldots, M'_n)$ with $E(M'_i) = \sigma E(M_i)$ for all $i \in \{1, \ldots, n\}$. By induction hypothesis, for all $i \in \{1, \ldots, n\}$, there exists σ'_i such that $M'_i = \sigma'_i M_i$ and, for all variables x of M_i, $E(\sigma'_i x) = \sigma x$. For all i, j, if x occurs in M_i and M_j, $E(\sigma'_i x) = \sigma x = E(\sigma'_j x)$, so by Lemma 8, $\sigma'_i x = \sigma'_j x$. Thus we can merge all substitutions σ'_i into a substitution σ' defined by $\sigma' x = \sigma'_i x$ when x occurs in M_i. So we have $M' = \sigma' M$ and, for all variables x of M, $E(\sigma' x) = \sigma x$.

The extension to atoms and to tuples of terms and atoms is similar to the case of composite terms. □

Proof (of Lemma 1) Let Q be an *Init*-adversary and $Q' = \text{instrAdv}(Q)$. Let E_0 containing $\mathit{fn}(P_0) \cup \mathit{Init} \cup \mathit{fn}(\alpha) \cup \bigcup_j \mathit{fn}(\alpha_j) \cup \bigcup_{j,k} \mathit{fn}(M_{jk})$. Consider a trace $\mathcal{T} = E_0, \{P_0, Q\} \to E_1, \mathcal{P}_1$. Let σ such that $\mathcal{T}$ satisfies $\sigma\alpha$. By Proposition 1, letting $E'_0 = \{a \mapsto a[\,] \mid a \in E_0\}$, there is a trace $\mathcal{T}' = S_0, E'_0, \{P'_0, Q'\} \to^* S', E'_1, \mathcal{P}'_1$, $\text{unInstr}(\mathcal{P}'_1) = \mathcal{P}_1$, and both traces satisfy the same atoms, so $\mathcal{T}'$ also satisfies $\sigma\alpha$. Since E'_0 contains the names of α, α_j, and M_{jk}, and E'_1 is an extension of E'_0, $E'_1(\alpha) = E'_0(\alpha) = F$, $E'_1(\alpha_j) = E'_0(\alpha_j) = F_j$, and $E'_1(M_{jk}) = E'_0(M_{jk}) = p_{jk}$. Let σ'' be defined by $\sigma'' x = E_1(\sigma x)$ for all x. By Lemma 7, $E'_1(\sigma\alpha) = \sigma'' E'_1(\alpha)$, so $E'_1(\sigma\alpha) = \sigma'' F$. Hence $\mathcal{T}'$ satisfies $\sigma'' F$. Since P'_0 satisfies the given correspondence, there exist σ''_0 and $j \in \{1, \ldots, m\}$ such that $\sigma''_0 F_j = \sigma'' F$ and for all $k \in \{1, \ldots, l_j\}$, $\mathcal{T}'$ satisfies $\text{event}(\sigma''_0 p_{jk})$, so there exists M''_k such that $E'_1(M''_k) = \sigma''_0 p_{jk}$ and $\mathcal{T}'$ satisfies $\text{event}(M''_k)$. Hence $E'_1(M''_k) = \sigma''_0 E'_1(M_{jk})$ and $E'_1(\sigma\alpha) = \sigma'' F = \sigma''_0 F_j = \sigma''_0 E'_1(\alpha_j)$, that is, $E'_1((M''_1, \ldots, M''_{l_j}, \sigma\alpha)) = \sigma''_0 E'_1(M_{j1}, \ldots, M_{jl_j}, \alpha_j)$. By Lemma 9, there exists σ_0 such that $(M''_1, \ldots, M''_{l_j}, \sigma\alpha) = \sigma_0(M_{j1}, \ldots, M_{jl_j}, \alpha_j)$. So $\sigma\alpha = \sigma_0 \alpha_j$ and for all $k \in \{1, \ldots, l_j\}$, $\mathcal{T}'$ satisfies $\text{event}(\sigma_0 M_{jk})$, so $\mathcal{T}$ also satisfies $\text{event}(\sigma_0 M_{jk})$. □

B Proof of Theorem 1

The correctness proof uses a type system as a convenient way of expressing invariants of processes. This type system can be seen as a modified version of the type system of [1, Section 7], which was used to prove the correctness of our protocol verifier for secrecy properties. In this type system, the types are closed patterns:

$T ::=$	types
$\quad a[T_1, \ldots, T_n, \lambda_1, \ldots, \lambda_k]$	name
$\quad f(T_1, \ldots, T_n)$	constructor application

The symbols $\lambda_1, \ldots, \lambda_k$ are constant session identifiers, in a set S_0. Let $\mathcal{F}_{P'_0,\mathit{Init}}$ be the set of closed facts derivable from $\mathcal{R}_{P'_0,\mathit{Init}} \cup \mathcal{F}_{\text{me}}$.

The type rules are defined in Figure 7. The environment E is a function from names and variables in V_o to types and from variables in V_s to constant session identifiers. The mapping E is extended to all terms as a substitution by $E(f(M_1, \ldots, M_n)) = f(E(M_1), \ldots, E(M_n))$ and to

$$\frac{\text{message}(E(M), E(N)) \in \mathcal{F}_{P'_0,Init} \qquad E \vdash P}{E \vdash \overline{M}\langle N \rangle.P} \tag{Output}$$

$$\frac{\forall T' \text{ such that message}(E(M), T') \in \mathcal{F}_{P'_0,Init}, E[x \mapsto T'] \vdash P}{E \vdash M(x).P} \tag{Input}$$

$$\frac{}{E \vdash 0} \tag{Nil}$$

$$\frac{E \vdash P \qquad E \vdash Q}{E \vdash P \mid Q} \tag{Parallel}$$

$$\frac{\forall \lambda, E[i \mapsto \lambda] \vdash P}{E \vdash !^i P} \tag{Replication}$$

$$\frac{E[a \mapsto E(\ell)] \vdash P}{E \vdash (\nu a : \ell)P} \tag{Restriction}$$

$$\frac{\forall T \text{ such that } g(E(M_1), \dots, E(M_n)) \to T, E[x \mapsto T] \vdash P \qquad E \vdash Q}{E \vdash \mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q} \tag{Destructor application}$$

$$\frac{\text{event}(E(M)) \in \mathcal{F}_{P'_0,Init} \qquad \text{if m-event}(E(M)) \in \mathcal{F}_{P'_0,Init} \text{ then } E \vdash P}{E \vdash \texttt{event}(M).P} \tag{Event}$$

Figure 7: Type rules

restriction labels by $E(a[M_1, \dots, M_n, i_1, \dots, i_{n'}]) = a[E(M_1), \dots, E(M_n), E(i_1), \dots, E(i_{n'})]$ and $E(b_0[a[i_1, \dots, i_{n'}]]) = b_0[a[E(i_1), \dots, E(i_{n'})]]$, so that it maps closed terms and restriction labels to types. The rules define the judgment $E \vdash P$, which means that the process P is well-typed in the environment E. We do not consider the case of conditionals here, since it is a particular case of destructor applications.

We say that an instrumented semantic configuration $S, E, \mathcal{P}$ is well-typed, and we write $\vdash S, E, \mathcal{P}$, when it is well-labeled and $E \vdash P$ for all $P \in \mathcal{P}$.

Proof sketch (of Theorem 1) Let P_0 be the considered process and $P'_0 = \text{instr}(P_0)$. Let Q be an *Init*-adversary and $Q' = \text{instrAdv}(Q)$. Let E_0 such that $\mathit{fn}(P'_0) \cup \mathit{Init} \subseteq \mathit{dom}(E_0)$ and for all $a \in \mathit{dom}(E_0)$, $E_0(a) = a[\,]$.

1. *Typability of the adversary:* Let P' be a subprocess of Q'. Let E be an environment such that $\forall a \in \mathit{fn}(P')$, $\text{attacker}(E(a)) \in \mathcal{F}_{P'_0,Init}$ and $\forall x \in \mathit{fv}(P')$, $\text{attacker}(E(x)) \in \mathcal{F}_{P'_0,Init}$. (In particular, E is defined for all free names and free variables of P'.) We show that $E \vdash P'$, by induction on P'. This result is similar to [1, Lemma 5.1.4]. In particular, we obtain $E_0 \vdash Q'$.

2. *Typability of P'_0:* We prove by induction on the process P, subprocess of P'_0, that, if (a) ρ binds all free names and variables of P, (b) $\mathcal{R}_{P'_0,Init} \supseteq [\![P]\!]\rho H$, (c) σ is a closed substitution, and (d) σH can be derived from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\text{me}}$, then $\sigma\rho \vdash P$. This result is similar to [1, Lemma 7.2.2].

   In particular, $\mathcal{R}_{P'_0,Init} \supseteq [\![P'_0]\!]\rho\emptyset$, where $\rho = \{a \mapsto a[\,] \mid a \in \mathit{fn}(P'_0)\}$. So, with $E = \sigma\rho = \{a \mapsto a[\,] \mid a \in \mathit{fn}(P'_0)\}$, $E \vdash P'_0$. A fortiori, $E_0 \vdash P'_0$.

3. *Properties of P'_0, Q':* By Lemma 5, $S_0, E_0, \{P'_0, Q'\}$ is well-labeled. So, using the first two points, $\vdash S_0, E_0, \{P'_0, Q'\}$.

4. *Substitution lemma:* Let $E' = E[x \mapsto E(M)]$. We show by induction on M' that $E(M'\{M/x\}) = E'(M')$. We show by induction on P that, if $E' \vdash P$, then $E \vdash P\{M/x\}$. This result is similar to [1, Lemma 5.1.1].

5. *Subject reduction:* Assume that $\vdash S, E, \mathcal{P}$ and $S, E, \mathcal{P} \rightarrow S', E', \mathcal{P}'$. Furthermore, assume that, if the reduction $S, E, \mathcal{P} \rightarrow S', E', \mathcal{P}'$ executes $\texttt{event}(M)$, then $\text{m-event}(E(M)) \in \mathcal{F}_{\text{me}}$. Then $\vdash S', E', \mathcal{P}'$. This is proved by cases on the derivation of $S, E, P \rightarrow S', E', P'$. This result is similar to [1, Lemma 5.1.3].

6. Consider the trace $\mathcal{T} = S_0, E_0, \{P_0', Q'\} \rightarrow^* S', E', \mathcal{P}'$. By the hypothesis of the theorem, if $\texttt{event}(M)$ has been executed in $\mathcal{T}$, then $\mathcal{T}$ satisfies $\text{event}(E'(M))$, so $\text{m-event}(E'(M)) \in \mathcal{F}_{\text{me}}$. If the reduction that executes $\texttt{event}(M)$ is $S, E, \mathcal{P} \rightarrow S, E, \mathcal{P}''$, we have $E(M) = E'(M)$, since E' is an extension of E, and E already contains the names of M. Hence we obtain the hypothesis of subject reduction. So, by Items 3 and 5, we infer that all configurations in the trace are well-typed.

   When $F = \text{event}(p)$, since $\mathcal{T}$ satisfies $\text{event}(p)$, there exists M such that $\mathcal{T}$ satisfies $\text{event}(M)$ and $E'(M) = p$. So $\mathcal{T}$ contains a reduction $S_1, E_1, \mathcal{P}_1 \cup \{\texttt{event}(M).P\} \rightarrow S_1, E_1, \mathcal{P}_1 \cup \{P\}$. Therefore $E_1 \vdash \texttt{event}(M).P$, so $\text{event}(E_1(M)) \in \mathcal{F}_{P_0',\mathit{Init}}$. Moreover, $E_1(M) = E'(M)$ since E' is an extension of E_1, therefore $\text{event}(E'(M)) = \text{event}(p) = F$ is derivable from $\mathcal{R}_{P_0',\mathit{Init}} \cup \mathcal{F}_{\text{me}}$.

   When $F = \text{message}(p, p')$, since $\mathcal{T}$ satisfies $\text{message}(p, p')$, there exist M and M' such that $\mathcal{T}$ satisfies $\text{message}(M, M')$, $E'(M) = p$, and $E'(M') = p'$. So $\mathcal{T}$ contains a reduction $S_1, E_1, \mathcal{P}_1 \cup \{\overline{M}\langle M'\rangle.P, M(x).Q\} \rightarrow S_1, E_1, \mathcal{P}_1 \cup \{P, Q\{M/x\}\}$. Therefore $E_1 \vdash \overline{M}\langle M'\rangle.P$. This judgment must have been derived by (Output), so $\text{message}(E_1(M), E_1(M')) \in \mathcal{F}_{P_0',\mathit{Init}}$. Moreover, $E_1(M) = E'(M)$ and $E_1(M') = E'(M')$ since E' is an extension of E_1, so $\text{message}(E'(M), E'(M')) = \text{message}(p, p') = F$ is derivable from $\mathcal{R}_{P_0',\mathit{Init}} \cup \mathcal{F}_{\text{me}}$.

   When $F = \text{attacker}(p')$, $\mathcal{T}$ also satisfies $\text{message}(c[\,], p')$ for some $c \in \mathit{Init}$. Therefore, by the previous case, $\text{message}(c[\,], p')$ is derivable from $\mathcal{R}_{P_0',\mathit{Init}} \cup \mathcal{F}_{\text{me}}$. Since $c \in \mathit{Init}$, $\text{attacker}(c[\,])$ is in $\mathcal{R}_{P_0',\mathit{Init}}$. So, by Clause (Rl), $\text{attacker}(p') = F$ is derivable from $\mathcal{R}_{P_0',\mathit{Init}} \cup \mathcal{F}_{\text{me}}$. □

C Correctness of the Solving Algorithm

In terms of security, the soundness of our analysis means that, if a protocol is found secure by the analysis, then it is actually secure. Showing soundness in this sense essentially amounts to showing that no derivable fact is missed by the resolution algorithm, which, in terms of logic programming, is the completeness of the resolution algorithm. Accordingly, in terms of security, the completeness of our analysis would mean that all secure protocols can be proved secure by our analysis. Completeness in terms of security corresponds, in terms of logic programming, to the correctness of the resolution algorithm, which means that the resolution algorithm does not derive false facts.

The completeness of "binary resolution with free selection", which is our basic algorithm, was proved in [9, 35, 55]. We extend these proofs by showing that completeness still holds with our simplifications of clauses. (These simplifications are often specific to security protocols.)

As a preliminary, we define a sort system, with three sorts: session identifiers, ordinary patterns, and environments. Name function symbols expect session identifiers as their last k arguments where k is the number of replications above the restriction that defines the considered name function symbol, and ordinary patterns as other arguments. The pattern $a[p_1, \ldots, p_n, i_1, \ldots, i_k]$ is an ordinary pattern. Constructors f expect ordinary patterns as arguments and $f(p_1, \ldots, p_n)$ is an ordinary pattern. The predicates attacker and message expect

ordinary patterns as arguments. The predicate event expects an ordinary pattern and, for injective events, a session identifier. The predicate m-event expects an ordinary pattern and, for injective events, an environment. We say that a pattern, fact, clause, set of clauses is *well-sorted* when these constraints are satisfied.

Lemma 10 *All clauses manipulated by the algorithm are well-sorted, and if a variable occurs in the conclusion of a clause and is not a session identifier, then it also occurs in non-*m-event *facts in its hypothesis.*

Proof It is easy to check that all patterns and facts are well-sorted in the clause generation algorithm. One only unifies patterns of the same sort. The environment ρ and the substitutions always map a variable to a pattern of the same sort. During the building of clauses, the variables in the image of ρ that are not session identifiers also occur in non-m-event facts in H, and the variables in the conclusion of generated clauses are in the image of ρ. Hence, the clauses in $\mathcal{R}_{P_0',Init}$ satisfy Lemma 10.

Furthermore, this property is preserved by resolution. Resolution generates a clause $R'' = \sigma_u H \wedge \sigma_u H' \Rightarrow \sigma_u C'$ from clauses $R = H \Rightarrow C$ and $R' = H' \wedge F_0 \Rightarrow C'$ that satisfy Lemma 10, where σ_u is the most general unifier of C and F_0. The substitution σ_u unifies elements of the same sort, so σ_u maps each variable to an element of the same sort, so R'' is well-sorted. If a non-session identifier variable x occurs in $\sigma_u C'$, then there is a non-session identifier variable y in C' such that x occurs in $\sigma_u y$. Then y occurs in non-m-event facts in the hypothesis of R', $H' \wedge F_0$. First case: y occurs in non-m-event facts in H', so x occurs in $\sigma_u H'$, so x occurs in non-m-event facts in the hypothesis of R''. Second case: y occurs in F_0, so x occurs in $\sigma_u F_0 = \sigma_u C$, so there is a non-session identifier variable z such that z occurs in C and x occurs in $\sigma_u z$, so z occurs in non-m-event facts in H, so x occurs in non-m-event facts in $\sigma_u H$, so x occurs in non-m-event facts in the hypothesis of R''. In both cases, x occurs in non-m-event facts in the hypothesis of R''. Therefore, R'' satisfies Lemma 10.

This property is also preserved by the simplification functions. □

Definition 17 (Derivation) Let F be a closed fact. Let $\mathcal{R}$ be a set of clauses. A derivation of F from $\mathcal{R}$ is a finite tree defined as follows:

1. Its nodes (except the root) are labeled by clauses $R \in \mathcal{R}$.
2. Its edges are labeled by closed facts. (Edges go from a node to each of its sons.)
3. If the tree contains a node labeled by R with one incoming edge labeled by F_0 and n outgoing edges labeled by $F_1, \dots, F_n$, then $R \sqsupseteq \{F_1, \dots, F_n\} \Rightarrow F_0$.
4. The root has one outgoing edge, labeled by F. The unique son of the root is named the *subroot*.

In a derivation, if there is a node labeled by R with one incoming edge labeled by F_0 and n outgoing edges labeled by $F_1, \dots, F_n$, then the clause R can be used to infer F_0 from $F_1, \dots, F_n$. Therefore, there exists a derivation of F from $\mathcal{R}$ if and only if F can be inferred from clauses in $\mathcal{R}$ (in classical logic).

The key idea of the proof of Lemma 2 is the following. Assume that F is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\mathrm{me}}$ and consider a derivation of F from $\mathcal{R}_0 \cup \mathcal{F}_{\mathrm{me}}$. Assume that the clauses R and R' are applied one after the other in the derivation of F. Also assume that these clauses have been combined by $R \circ_{F_0} R'$, yielding clause R''. In this case, we replace R and R' with R'' in the derivation of F. When no more replacement can be done, we show that all remaining clauses have no selected hypothesis. So all these clauses are in $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$, and we have built a derivation of F from $\mathcal{R}_1$.

To show that this replacement process terminates, we remark that the total number of nodes of the derivation strictly decreases.

Next, we introduce the notion of data-decomposed derivation. This notion is useful for proving the correctness of the decomposition of data constructors. (In the absence of data constructors, all derivations are data-decomposed.)

Definition 18 A derivation D is *data-decomposed* if and only if, for all edges $\eta' \rightarrow \eta$ in D labeled by $\text{attacker}(f(p_1, \ldots, p_n))$ for some data constructor f, the node η' is labeled by a clause $\text{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \text{attacker}(x_i)$ for some i or the node η is labeled by the clause $\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$.

Intuitively, a derivation is data-decomposed when all intermediate facts proved in that derivation are decomposed as much as possible using data-destructor clauses $\text{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \text{attacker}(x_i)$ before being used to prove other facts. We are going to transform the initial derivation into a data-decomposed derivation. Further transformations of the derivation will keep it data-decomposed.

The next lemma shows that two nodes in a derivation can be replaced by one when combining their clauses by resolution.

Lemma 11 *Consider a data-decomposed derivation containing a node η', labeled R'. Let F_0 be a hypothesis of R'. Then there exists a son η of η', labeled R, such that the edge $\eta' \rightarrow \eta$ is labeled by an instance of F_0, $R \circ_{F_0} R'$ is defined, and, if $\mathsf{sel}(R) = \emptyset$ and $F_0 \in \mathsf{sel}(R')$, one obtains a data-decomposed derivation of the same fact by replacing the nodes η and η' with a node η'' labeled $R'' = R \circ_{F_0} R'$.*

Proof This proof is illustrated in Figure 8. Let $R' = H' \Rightarrow C'$, H'_1 be the multiset of the labels of the outgoing edges of η', and C'_1 the label of its incoming edge. We have $R' \sqsupseteq (H'_1 \Rightarrow C'_1)$, so there exists σ such that $\sigma H' \subseteq H'_1$ and $\sigma C' = C'_1$. Hence there is an outgoing edge of η' labeled σF_0, since $\sigma F_0 \in H'_1$. Let η be the node at the end of this edge, let $R = H \Rightarrow C$ be the label of η. We rename the variables of R such that they are distinct from the variables of R'. Let H_1 be the multiset of the labels of the outgoing edges of η. So $R \sqsupseteq (H_1 \Rightarrow \sigma F_0)$. By the above choice of distinct variables, we can then extend σ such that $\sigma H \subseteq H_1$ and $\sigma C = \sigma F_0$.

The edge $\eta' \rightarrow \eta$ is labeled σF_0, instance of F_0. Since $\sigma C = \sigma F_0$, the facts C and F_0 are unifiable, so $R \circ_{F_0} R'$ is defined. Let σ' be the most general unifier of C and F_0, and σ'' such that $\sigma = \sigma''\sigma'$. We have $R \circ_{F_0} R' = \sigma'(H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma' C'$. Moreover, $\sigma''\sigma'(H \cup (H' \setminus \{F_0\})) \subseteq H_1 \cup (H'_1 \setminus \{\sigma F_0\})$ and $\sigma''\sigma' C' = \sigma C' = C'_1$. Hence $R'' = R \circ_{F_0} R' \sqsupseteq (H_1 \cup (H'_1 \setminus \{\sigma F_0\})) \Rightarrow C'_1$. The multiset of labels of outgoing edges of η'' is precisely $H_1 \cup (H'_1 \setminus \{\sigma F_0\})$ and the label of its incoming edge is C'_1, therefore we have obtained a correct derivation by replacing η and η' with η''.

Let us show that the obtained derivation is data-decomposed. Consider an edge $\eta'_1 \rightarrow \eta_1$ in this derivation, labeled by $F = \text{attacker}(f(p_1, \ldots, p_n))$, where f is a data constructor.

- If η'_1 and η_1 are different from η'', then the same edge exists in the initial derivation, so it is of the desired form.
- If $\eta'_1 = \eta''$, then there is an edge $\eta \rightarrow \eta_1$ labeled by F in the initial derivation. Since the initial derivation is data-decomposed, η is labeled by $R = \text{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \text{attacker}(x_i)$ or η_1 is labeled by $R_1 = \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$. The former case is impossible because $\mathsf{sel}(R) = \emptyset$. In the latter case, η_1 is labeled by R_1, so we have the desired form in the obtained derivation.
- If $\eta_1 = \eta''$, then there is an edge $\eta'_1 \rightarrow \eta'$ labeled by F in the initial derivation. Since the initial derivation is data-decomposed, η'_1 is labeled by $R'_1 = \text{attacker}(f(x_1, \ldots, x_n)) \Rightarrow$

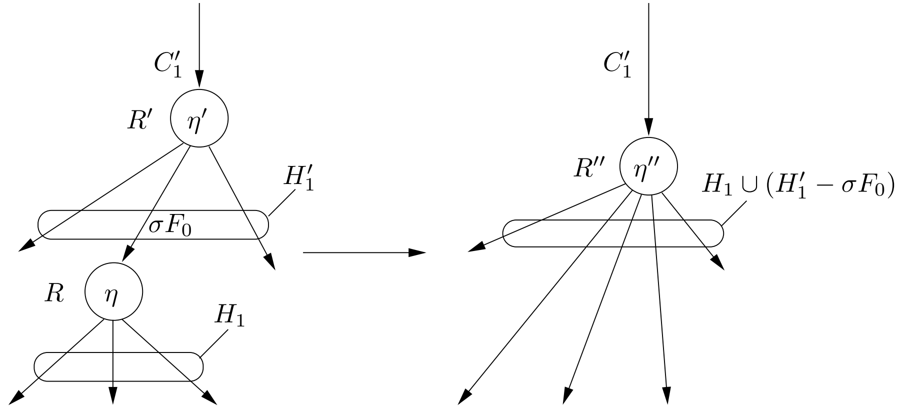

Figure 8: Merging of nodes of Lemma 11

attacker(x_i) or η' is labeled by $R' = \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$. The latter case is impossible because $\mathsf{sel}(R) \neq \emptyset$. In the former case, η'_1 is labeled by R'_1, so we have the desired form in the obtained derivation.

Hence the obtained derivation is data-decomposed. □

Lemma 12 *If a node η of a data-decomposed derivation D is labeled by R, then one obtains a data-decomposed derivation D' of the same fact as D by relabeling η with a clause R' such that $R' \sqsupseteq R$.*

Proof Let H be the multiset of labels of outgoing edges of the considered node η, and C be the label of its incoming edge. We have $R \sqsupseteq H \Rightarrow C$. By transitivity of $\sqsupseteq$, $R' \sqsupseteq H \Rightarrow C$. So we can relabel η with R'.

Let us show that the obtained derivation D' is data-decomposed. Consider an edge $\eta'_1 \to \eta_1$ in D', labeled by $F = \text{attacker}(f(p_1, \ldots, p_n))$, where f is a data constructor.

- If η'_1 and η_1 are different from η, then the same edge exists in the initial derivation D, so it is of the desired form.

- If $\eta'_1 = \eta$, then there is an edge $\eta'_1 \to \eta_1$ in D, labeled by F. Since D is data-decomposed, $\eta'_1 = \eta$ is labeled by $R = \text{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \text{attacker}(x_i)$ or η_1 is labeled by $R_1 = \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$ in D. In the latter case, we have the desired form in D'. In the former case, let $R' = H' \Rightarrow C'$. We have $R' \sqsupseteq R$, so there exists σ such that $\sigma H' \subseteq \{\text{attacker}(f(x_1, \ldots, x_n))\}$ and $\sigma C' = \text{attacker}(x_i)$. Hence $C' = \text{attacker}(y)$ where $\sigma y = x_i$, and $H' = \emptyset$ or $H' = \text{attacker}(z)$ with $\sigma z = f(x_1, \ldots, x_n)$ or $H' = \text{attacker}(f(y_1, \ldots, y_n))$ with $\sigma y_j = x_j$ for all $j \leq n$. By Lemma 10, y occurs in H', so $H' \neq \emptyset$. If we had $H' = \text{attacker}(z)$, $\sigma z \neq \sigma y$, so $z \neq y$, so this case is impossible. Hence $H' = \text{attacker}(f(y_1, \ldots, y_n))$. Moreover, $\sigma y_j \neq \sigma y$ for all $j \neq i$, so $y_j \neq y$ for all $j \neq i$. Since y occurs in H', $y = y_i$. Hence $R' = R$ up to renaming, and we have the desired form in D'.

- If $\eta_1 = \eta$, then there is an edge $\eta'_1 \to \eta_1$ in D, labeled by F. Since D is data-decomposed, η'_1 is labeled by $R'_1 = \text{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \text{attacker}(x_i)$ or $\eta_1 = \eta$ is labeled by $R = \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$ in D. In the former case, we have the desired form in D'. In the latter case, let $R' = H' \Rightarrow C'$. We have $R' \sqsupseteq R$, so there exists σ such that $\sigma H' \subseteq \{\text{attacker}(x_1), \ldots, \text{attacker}(x_n)\}$ and $\sigma C' = \text{attacker}(f(x_1, \ldots, x_n))$. Hence $H' = \bigwedge_{j \in J} \text{attacker}(y_j)$ where $J \subseteq \{1, \ldots, n\}$ and $\sigma y_j = x_j$ for all $j \in J$, and $C' = \text{attacker}(y)$ with $\sigma y = f(x_1, \ldots, x_n)$ or $C' = \text{attacker}(f(y'_1, \ldots, y'_n))$

with $\sigma y'_j = x_j$ for all $j \leq n$. By Lemma 10, if $C' = \text{attacker}(y)$, y occurs in H', but this is impossible because $\sigma y_j \neq \sigma y$ for all $j \in J$. So $C' = \text{attacker}(f(y'_1, \ldots, y'_n))$. By Lemma 10, y'_j occurs in H' for all $j \leq n$, so $J = \{1, \ldots, n\}$ and $y'_j = y_j$ for all $j \leq n$. Hence $R' = R$ up to renaming, and we have the desired form in D'.

Hence the obtained derivation D' is data-decomposed. □

Definition 19 We say that $\mathcal{R} \sqsupseteq_{\text{Set}} \mathcal{R}'$ if, for all clauses R in $\mathcal{R}'$, R is subsumed by a clause of $\mathcal{R}$.

Lemma 13 *If $\mathcal{R} \sqsupseteq_{\text{Set}} \mathcal{R}'$ and D is a data-decomposed derivation containing a node η labeled by $R \in \mathcal{R}'$, then one can build a data-decomposed derivation D' of the same fact as D by relabeling η with a clause in $\mathcal{R}$.*

Proof Obvious by Lemma 12. □

Lemma 14 *If $\mathcal{R} \sqsupseteq_{\text{Set}} \mathcal{R}'$, then elim$(\mathcal{R}) \sqsupseteq_{\text{Set}} \mathcal{R}'$.*

Proof This is an immediate consequence of the transitivity of $\sqsupseteq$. □

Lemma 15 *At the end of* saturate*, $\mathcal{R}$ satisfies the following properties:*

1. *For all $R \in \mathcal{R}_0$, $\mathcal{R} \sqsupseteq_{\text{Set}}$ simplify(R);*
2. *Let $R \in \mathcal{R}$ and $R' \in \mathcal{R}$. Assume that $\mathsf{sel}(R) = \emptyset$ and there exists $F_0 \in \mathsf{sel}(R')$ such that $R \circ_{F_0} R'$ is defined. In this case, $\mathcal{R} \sqsupseteq_{\text{Set}}$ simplify$(R \circ_{F_0} R')$.*

Proof To prove the first property, let $R \in \mathcal{R}_0$. We show that, after the addition of R to $\mathcal{R}$, $\mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*(R).

In the first step of saturate, we execute the instruction $\mathcal{R} \leftarrow$ *elim*(*simplify*$(R) \cup \mathcal{R}$). We have *simplify*$(R) \cup \mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*(R), so, by Lemma 14, after execution of this instruction, $\mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*(R).

Assume that we execute $\mathcal{R} \leftarrow$ *elim*(*simplify*$(R'') \cup \mathcal{R}$), and before this execution $\mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*(R). Hence *simplify*$(R'') \cup \mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*(R), so, by Lemma 14, after the execution of this instruction, $\mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*(R).

The second property simply means that the fixpoint is reached at the end of saturate, so $\mathcal{R} =$ *elim*(*simplify*$(R \circ_{F_0} R') \cup \mathcal{R}$). Since *simplify*$(R \circ_{F_0} R') \cup \mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*$(R \circ_{F_0} R')$, by Lemma 14, *elim*(*simplify*$(R \circ_{F_0} R') \cup \mathcal{R}) \sqsupseteq_{\text{Set}}$ *simplify*$(R \circ_{F_0} R')$, so $\mathcal{R} \sqsupseteq_{\text{Set}}$ *simplify*$(R \circ_{F_0} R')$. □

Lemma 16 *Let $f \in$ {elimattx, elimtaut, elimnot, elimredundanthyp, elimdup, decomp, decomphyp, simplify, simplify'}.*

If the data-decomposed derivation D contains a node η labeled R, then one obtains a data-decomposed derivation D' of the same fact as D or of an instance of a fact in $\mathcal{F}_{\text{not}}$ by relabeling η with some $R' \in f(R)$ or removing η, and possibly deleting nodes. Furthermore, if D' is not a derivation of the same fact as D, then η is removed.

If D' contains a node labeled $R' \in f(R)$, then there exists a derivation D using R, the clauses of D' except R', and the clauses of $\mathcal{R}_0$ that derives the same fact as D'.

When R is unchanged by f, that is, $f(R) = \{R\}$, this lemma is obvious. So, in the proofs below, we consider only the cases in which R is modified by f.

Proof (for *elimattx***)** The direct part is obvious: R' is built from R by removing some hypotheses, so we just remove the subtrees corresponding to removed hypotheses of R.

Conversely, let p be a closed pattern such that $\text{attacker}(p)$ is derivable from $\mathcal{R}_0$. (There exists an infinite number of such p.) We build a derivation D by replacing R' with R in D and adding a derivation of $\text{attacker}(p)$ as a subtree of the nodes labeled by R' in D. □

Proof (for *elimtaut***)** Assume that R is a tautology. For the direct part, we remove η and replace it with one of its subtrees. The converse is obvious since $\mathit{elimtaut}(R) = \emptyset$. □

Proof (for *elimnot***)** Assume that R contains as hypothesis an instance F of a fact in $\mathcal{F}_{\text{not}}$. Then $\mathit{elimnot}(R) = \emptyset$. Since D is a derivation, a son η' of η infers an instance of F. We let D' be the sub-derivation with subroot η'. D' is a derivation of an instance of a fact in $\mathcal{F}_{\text{not}}$, so we obtain the direct part. The converse is obvious since $\mathit{elimnot}(R) = \emptyset$. □

Proof (for *elimredundanthyp***)** We have $R = H \wedge H' \Rightarrow C$, $\sigma H \subseteq H'$, σ does not change the variables of H' and C, and $R' = H' \Rightarrow C$.

For the direct part, R' is built from R by removing some hypotheses, so we just remove the subtrees corresponding to removed hypotheses of R.

For the converse, we obtain a derivation D by duplicating the subtrees proving instances of elements of H' that are also in σH and replacing R' with R. □

Proof (for *elimdup***)** For the direct part, R' is built from R by removing some hypotheses, so we just remove the subtrees corresponding to removed hypotheses of R.

Conversely, we can form a derivation using R instead of R' by duplicating the subtrees that derive the duplicate hypotheses of R. □

Proof (for *decomp* **and** *decomphyp***)** If R is modified by *decomp* or *decomphyp*, then R is of one of the following forms:

- $R = \text{attacker}(f(p_1, \dots, p_n)) \wedge H \Rightarrow C$, where f is a data constructor (for both *decomp* and *decomphyp*).

  For the direct part, let η' be the son of η corresponding to the hypothesis $\text{attacker}(f(p_1, \dots, p_n))$. The edge $\eta \to \eta'$ is labeled by an instance of $\text{attacker}(f(p_1, \dots, p_n))$, so, since D is data-decomposed, η' is labeled by $\text{attacker}(x_1) \wedge \dots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \dots, x_n))$. (The clause R that labels η cannot be $\text{attacker}(f(x_1, \dots, x_n)) \Rightarrow \text{attacker}(x_i)$, since this clause would be unmodified by *decomp* and *decomphyp*.) Then we build D' by relabeling η with $R' = \text{attacker}(p_1) \wedge \dots \wedge \text{attacker}(p_n) \wedge H \Rightarrow C$ and deleting η'.

  For the converse, we replace $R' = \text{attacker}(p_1) \wedge \dots \wedge \text{attacker}(p_n) \wedge H \Rightarrow C$ in D' with $\text{attacker}(x_1) \wedge \dots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \dots, x_n))$ and $R = \text{attacker}(f(p_1, \dots, p_n)) \wedge H \Rightarrow C$ in D.

- $R = H \Rightarrow \text{attacker}(f(p_1, \dots, p_n))$, where f is a data constructor (for *decomp* only).

  For the direct part, let η' be the father of η. The edge $\eta' \to \eta$ is labeled by an instance of $\text{attacker}(f(p_1, \dots, p_n))$, so, since D is data-decomposed, η' is labeled by $\text{attacker}(f(x_1, \dots, x_n)) \Rightarrow \text{attacker}(x_i)$ for some i. (The clause R that labels η cannot be $\text{attacker}(x_1) \wedge \dots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \dots, x_n))$ since this clause would be unmodified by *decomp*.) Then we build D' by relabeling η with $R' = H \Rightarrow \text{attacker}(p_i)$ and deleting η'.

  For the converse, we replace $R' = H \Rightarrow \text{attacker}(p_i)$ in D' with $R = H \Rightarrow \text{attacker}(f(p_1, \dots, p_n))$ and $\text{attacker}(f(x_1, \dots, x_n)) \Rightarrow \text{attacker}(x_i)$ in D. □

Proof (for *simplify* **and** *simplify'***)** For *simplify* and *simplify'*, the result is obtained by applying Lemma 16 for the functions that compose *simplify* and *simplify'*. □

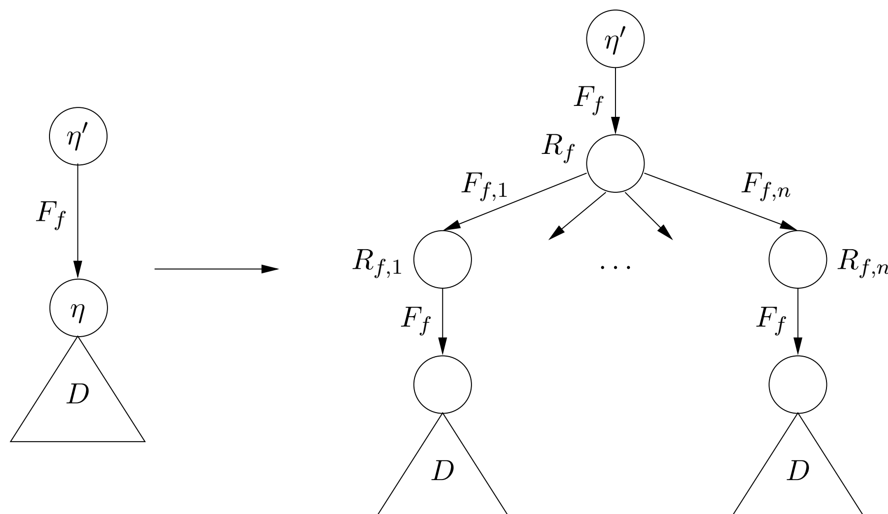

Figure 9: Construction of a data-decomposed derivation

Proof of Lemma 2 *Let F be a closed fact. If, for all $F' \in \mathcal{F}_{\text{not}}$, no instance of F' is derivable from* $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$*, then F is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\text{me}}$ if and only if F is derivable from* $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$.

Proof Assume that F is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\text{me}}$ and consider a derivation of F from $\mathcal{R}_0 \cup \mathcal{F}_{\text{me}}$. We show that F or an instance of a fact in $\mathcal{F}_{\text{not}}$ is derivable from $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\text{me}}$.

We first transform the derivation of F into a data-decomposed derivation. We say that an edge $\eta' \rightarrow \eta$ is *offending* when it is labeled by $F_f = \text{attacker}(f(p_1, \ldots, p_n))$ for some data constructor f, η' is not labeled by $R_{f,i} = \text{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \text{attacker}(x_i)$ for some i, and η is not labeled by $R_f = \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$. We consider an offending edge $\eta' \rightarrow \eta$ such that the subtree D of root η contains no offending edge. We copy the subtree D, which concludes F_f, n times and add the clauses $R_{f,i}$ for $i = 1, \ldots n$, to conclude $F_{f,i} = \text{attacker}(p_i)$, then use the clause R_f to conclude F_f again, as in Figure 9. This transformation decreases the total number of data constructors at the root of labels of offending edges. Indeed, since there are no offending edges in D, the only edges that may be offending in the new subtree of root η' are those labeled by $F_1, \ldots, F_n$. The total number of data constructors at the root of their labels is the total number of data constructors at the root of $p_1, \ldots, p_n$, which is one less than the total number of data constructors at the root of $f(p_1, \ldots, p_n)$. Hence, this transformation terminates and, upon termination, the obtained derivation contains no offending edge, so it is data-decomposed.

We consider the value of the set of clauses $\mathcal{R}$ at the end of $\mathsf{saturate}$. For each clause R in $\mathcal{R}_0$, $\mathcal{R} \sqsupseteq_{\text{Set}} \mathit{simplify}(R)$ (Lemma 15, Property 1). Assume that there exists a node labeled by $R \in \mathcal{R}_0 \setminus \mathcal{R}$ in this derivation. By Lemma 16, we can replace R with some $R'' \in \mathit{simplify}(R)$ or remove R. (After this replacement, we may obtain a derivation of an instance of a fact in $\mathcal{F}_{\text{not}}$ instead of a derivation of F.) If R is replaced with R'', by Lemma 13, we can replace R'' with a clause in $\mathcal{R}$. This transformation decreases the number of nodes labeled by clauses not in $\mathcal{R}$. So this transformation terminates and, upon termination, no node of the obtained derivation is labeled by a clause in $\mathcal{R}_0 \setminus \mathcal{R}$. Therefore, we obtain a data-decomposed derivation D of F or of an instance of a fact in $\mathcal{F}_{\text{not}}$ from $\mathcal{R} \cup \mathcal{F}_{\text{me}}$.

Next, we build a data-decomposed derivation of F or of an instance of a fact in $\mathcal{F}_{\text{not}}$ from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$, where $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$. If D contains a node labeled by a clause not in $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$, we can transform D as follows. Let η' be a lowest node of D labeled by a clause not in $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. So all sons of η' are labeled by elements of $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. Let R' be the clause labeling η'. Since

$R' \notin \mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$, $\mathsf{sel}(R') \neq \emptyset$. Take $F_0 \in \mathsf{sel}(R')$. By Lemma 11, there exists a son of η of η' labeled by R, such that $R \circ_{F_0} R'$ is defined. Since all sons of η' are labeled by elements of $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$, $R \in \mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. By definition of the selection function, F_0 is not a m-event fact, so $R \notin \mathcal{F}_{\mathrm{me}}$, so $R \in \mathcal{R}_1$. Hence $\mathsf{sel}(R) = \emptyset$. So, by Lemma 15, Property 2, $\mathcal{R} \sqsupseteq_{\mathrm{Set}} \mathit{simplify}(R \circ_{F_0} R')$. So, by Lemma 11, we can replace η and η' with η'' labeled by $R \circ_{F_0} R'$. By Lemma 16, we can replace $R \circ_{F_0} R'$ with some $R''' \in \mathit{simplify}(R \circ_{F_0} R')$ or remove $R \circ_{F_0} R'$.

- If $R \circ_{F_0} R'$ is replaced with R''', then by Lemma 13, we can replace R''' with a clause in $\mathcal{R}$. The total number of nodes strictly decreases since η and η' are replaced with a single node.
- If $R \circ_{F_0} R'$ is removed, then the total number of nodes strictly decreases since η and η' are removed.

So in all cases, we obtain a derivation D' of F or of an instance of a fact in $\mathcal{F}_{\mathrm{not}}$ from $\mathcal{R} \cup \mathcal{F}_{\mathrm{me}}$, such that the total number of nodes strictly decreases. Hence, this replacement process terminates. Upon termination, all clauses are in $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. So we obtain a data-decomposed derivation of F or of an instance of a fact in $\mathcal{F}_{\mathrm{not}}$ from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$, which is the expected result.

For the converse implication, notice that if a fact is derivable from $\mathcal{R}_1$ then it is derivable from $\mathcal{R}$, and that all clauses added to $\mathcal{R}$ do not create new derivable facts: when composing two clauses R and R', the created clause can derive facts that could also by derived by R and R'. □

Proof of Lemma 3 *Let F' be a closed instance of F. If, for all $F'' \in \mathcal{F}_{\mathrm{not}}$, $\mathsf{derivable}(F'', \mathcal{R}_1) = \emptyset$, then F' is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$ if and only if there exist a clause $H \Rightarrow C$ in $\mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution σ such that $\sigma C = F'$ and all elements of σH are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$.*

Proof Let us prove the direct implication. Let $\mathcal{F} = \{(F, F')\} \cup \{(F'', \sigma F'') \mid F'' \in \mathcal{F}_{\mathrm{not}}, \sigma$ any substitution$\}$. We show that, if F' is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$, then there exist a clause $H \Rightarrow C$ in $\mathsf{derivable}(F_{\mathrm{g}}, \mathcal{R}_1)$ and a substitution σ such that $(F_{\mathrm{g}}, \sigma C) \in \mathcal{F}$ and all elements of σH are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. (This property proves the desired result. If, for all $F'' \in \mathcal{F}_{\mathrm{not}}$, $\mathsf{derivable}(F'', \mathcal{R}_1) = \emptyset$ and F' is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$, then there exist a clause $H \Rightarrow C$ in $\mathsf{derivable}(F_{\mathrm{g}}, \mathcal{R}_1)$ and a substitution σ such that $(F_{\mathrm{g}}, \sigma C) \in \mathcal{F}$ and all elements of σH are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. Since, for all $F'' \in \mathcal{F}_{\mathrm{not}}$, $\mathsf{derivable}(F'', \mathcal{R}_1) = \emptyset$, we have $F_{\mathrm{g}} = F$ and $F \notin \mathcal{F}_{\mathrm{not}}$. Since $(F, \sigma C) \in \mathcal{F}$, we have then $\sigma C = F'$.)

Let $\mathcal{D}$ be the set of derivations D' of a fact F_{i} such that, for some F_{g} and $\mathcal{R}$, $(F_{\mathrm{g}}, F_{\mathrm{i}}) \in \mathcal{F}$, the clause R' at the subroot of D' satisfies $\mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{derivable}(F_{\mathrm{g}}, \mathcal{R}_1)$ and $\forall R'' \in \mathcal{R}, R'' \not\sqsupseteq R'$, and the other clauses of D' are in $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$.

Let attacker$'$ be a new predicate symbol. Let D be a derivation. If D is a derivation of attacker(p), we let D' be the derivation obtained by replacing the clause $H \Rightarrow \mathrm{attacker}(p_1)$ with $H \Rightarrow \mathrm{attacker}'(p_1)$ and the fact attacker(p) derived by D with attacker$'(p)$. If D is not a derivation of attacker(p), we let D' be D. We say that the derivation D is *almost-data-decomposed* when D' is data-decomposed. We first show that all derivations D in $\mathcal{D}$ are almost-data-decomposed. Let D' be the transformed derivation as defined above. Let $\eta' \to \eta$ be an edge of D' labeled by $F = \mathrm{attacker}(f(p_1, \ldots, p_n))$, where f is a data constructor. This edge is not the outgoing edge of the root of D', because D' does not conclude attacker(p) for any p. So the clause that labels η is of the form $R = H \Rightarrow \mathrm{attacker}(p)$ and it is in $\mathcal{R}_1$. In order to obtain a contradiction, assume that p is a variable x. Since $\mathsf{sel}(R) = \emptyset$, H contains only unselectable facts. By Lemma 10, x occurs in non-m-event facts in H, so H contains attacker(x). So R is a tautology. This is impossible because R would have been removed from $\mathcal{R}_1$ by *elimtaut*. So p is not a variable. Hence $p = f(p'_1, \ldots, p'_n)$. If R was different from $\mathrm{attacker}(x_1) \wedge \ldots \wedge \mathrm{attacker}(x_n) \Rightarrow \mathrm{attacker}(f(x_1, \ldots, x_n))$, R would have been transformed by

decomp, so R would not be in $\mathcal{R}_1$. Hence $R = \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$. Therefore, D' is data-decomposed, so D is almost-data-decomposed. Below, when we apply Lemma 11, 16, or 12, we first transform the considered derivation D into D', apply the lemma to the data-decomposed derivation D', and transform it back by replacing attacker$'$ with attacker. We obtain the same result as by transforming D directly, because the simplifications of *simplify*$'$ apply in the same way when the conclusion is $\text{attacker}(p)$ or $\text{attacker}'(p)$, since *simplify*$'$ uses *decomphyp* instead of *decomp* and does not use *elimtaut*.

Let D_0 be a derivation of F' from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. Let D'_0 be obtained from D_0 by adding a node labeled by $\{F\} \Rightarrow F$ at the subroot of D_0. By definition of $\mathsf{derivable}$, $\mathsf{deriv}(R', \emptyset, \mathcal{R}_1) \subseteq \mathsf{derivable}(F, \mathcal{R}_1)$, and $\forall R'' \in \emptyset, R'' \not\sqsupseteq R'$. Hence D'_0 is a derivation of F' in $\mathcal{D}$, so $\mathcal{D}$ is non-empty.

Now consider a derivation D_1 in $\mathcal{D}$ with the smallest number of nodes. The clause R' labeling the subroot η' of D_1 satisfies $(F_{\text{g}}, F_{\text{i}}) \in \mathcal{F}$, $\mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{derivable}(F_{\text{g}}, \mathcal{R}_1)$, and $\forall R'' \in \mathcal{R}, R'' \not\sqsupseteq R'$. In order to obtain a contradiction, we assume that $\mathsf{sel}(R') \neq \emptyset$. Let $F_0 \in \mathsf{sel}(R')$. By Lemma 11, there exists a son η of η', labeled by R, such that $R \circ_{F_0} R'$ is defined. By hypothesis on the derivation D_1, $R \in \mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. By the choice of the selection function, F_0 is not a m-event fact, so $R \notin \mathcal{F}_{\text{me}}$, so $R \in \mathcal{R}_1$. Let $R_0 = R \circ_{F_0} R'$. So, by Lemma 11, we can replace R' with R_0, obtaining a derivation D_2 of F_{i} with fewer nodes than D_1.

By Lemma 16, we can either replace R_0 with some $R'_0 \in \textit{simplify}'(R_0)$ or remove R_0, yielding a derivation D_3.

- In the latter case, D_3 is a derivation of a fact F'_{i} which is either F_{i} or an instance of a fact F'_{g} in $\mathcal{F}_{\text{not}}$. If $F'_{\text{i}} = F_{\text{i}}$, we let $F'_{\text{g}} = F_{\text{g}}$. So $(F'_{\text{g}}, F'_{\text{i}}) \in \mathcal{F}$.

  We replace R_0 with $R'_0 = F'_{\text{g}} \Rightarrow F'_{\text{g}}$ in D_2. Hence we obtain a derivation with fewer nodes than D_1 and such that $\mathsf{deriv}(R'_0, \emptyset, \mathcal{R}_1) \subseteq \mathsf{derivable}(F'_{\text{g}}, \mathcal{R}_1)$ and $\forall R_1 \in \emptyset, R_1 \not\sqsupseteq R'_0$. So we have a derivation in $\mathcal{D}$ with fewer nodes than D_1, which is a contradiction.

- In the former case, D_3 is a derivation of F_{i}, and $\mathsf{deriv}(R'_0, \{R'\} \cup \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{derivable}(F_{\text{g}}, \mathcal{R}_1)$ (third case of the definition of $\mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1)$).

  - If $\forall R_1 \in \{R'\} \cup \mathcal{R}, R_1 \not\sqsupseteq R'_0$, D_3 is a derivation of F_{i} in $\mathcal{D}$, with fewer nodes than D_1, which is a contradiction.
  - Otherwise, $\exists R_1 \in \{R'\} \cup \mathcal{R}, R_1 \sqsupseteq R'_0$. Therefore, by Lemma 12, we can build a derivation D_4 by replacing R'_0 with R_1 in D_3. There is an older call to deriv, of the form $\mathsf{deriv}(R_1, \mathcal{R}', \mathcal{R}_1)$, such that $\mathsf{deriv}(R_1, \mathcal{R}', \mathcal{R}_1) \subseteq \mathsf{derivable}(F_{\text{g}}, \mathcal{R}_1)$. Moreover, R_1 has been added to $\mathcal{R}'$ in this call, since R_1 appears in $\{R'\} \cup \mathcal{R}$. Therefore the third case of the definition of $\mathsf{deriv}(R_1, \mathcal{R}', \mathcal{R}_1)$ has been applied, and not the first case. So $\forall R_2 \in \mathcal{R}', R_2 \not\sqsupseteq R_1$, so the derivation D_4 is in $\mathcal{D}$ and has fewer nodes than D_1, which is a contradiction.

In all cases, we could find a derivation in $\mathcal{D}$ that has fewer nodes than D_1. This is a contradiction, so $\mathsf{sel}(R') = \emptyset$, hence $R' \in \mathsf{derivable}(F_{\text{g}}, \mathcal{R}_1)$. The other clauses of this derivation are in $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. By definition of a derivation, $R' \sqsupseteq H' \Rightarrow F_{\text{i}}$ where H' is the multiset of labels of the outgoing edges of the subroot of the derivation. Taking $R' = H \Rightarrow C$, there exists σ such that $\sigma C = F_{\text{i}}$ and $\sigma H \subseteq H'$, so all elements of σH are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\text{me}}$. We have the result, since $(F_{\text{g}}, F_{\text{i}}) \in \mathcal{F}$.

The proof of the converse implication is left to the reader. (Basically, the clause $R \circ_{F_0} R'$ does not generate facts that cannot be generated by applying R and R'.) □

D Termination Proof

In this section, we give the proof of Proposition 3 stated in Section 8.1. We denote by P_0 a tagged protocol and let $P'_0 = \text{instr}(P_0)$. We have the following properties:

- By Condition C2, the input and output constructs in the protocol always use a public channel c. So the facts message(c, p) are replaced with attacker(p) in all clauses. The only remaining clauses containing message are (Rl) and (Rs). Since message(x, y) is selected in these clauses, the only inference with these clauses is to combine (Rs) with (Rl), and it yields a tautology which is immediately removed. Therefore, we can ignore these clauses in our termination proof.

- By hypothesis on the queries and Remark 3, the clauses do not contain m-event facts.

In this section, we use the sort system defined at the beginning of Appendix C (Lemma 10).

The *patterns* of a fact $pred(p_1, \dots, p_n)$ are $p_1, \dots, p_n$. The *patterns* of a clause R are the patterns of all facts in R, and we denote the set of patterns of R by $patterns(R)$. A pattern is said to be *non-data* when it is not of the form $f(\dots)$ with f a data constructor. The set $sub(S)$ contains the subterms of patterns in the set S. Below, we use the word "program" for a set of clauses (that is, a logic program).

Definition 20 (Weakly tagged programs) Let S_0 be a finite set of closed patterns and *tagGen* be a set of patterns.

A pattern is *top-tagged* when it is an instance of a pattern in *tagGen*.

A pattern is *fully tagged* when all its non-variable non-data subterms are top-tagged.

Let $\mathcal{R}_{\text{ProtAdv}}$ be the set of clauses R that satisfy Lemma 10 and are of one of the following three forms:

1. $\mathcal{R}_{\text{Protocol}}$ contains clauses R of the form $F_1 \wedge \dots \wedge F_n \Rightarrow F$ where for all i, F_i is of the form attacker(p) for some p, F is of the form attacker(p) or event(p) for some p, there exists a substitution σ such that $patterns(\sigma R) \subseteq sub(S_0)$, and the patterns of R are fully-tagged.

2. $\mathcal{R}_{\text{Constr}}$ contains clauses of the form attacker$(x_1) \wedge \dots \wedge$attacker$(x_n) \Rightarrow$ attacker$(f(x_1, \dots, x_n))$ where f is a constructor.

3. $\mathcal{R}_{\text{Destr}}$ contains clauses of the form attacker$(f(p_1, \dots, p_n)) \wedge$ attacker$(x_1) \wedge \dots \wedge$ attacker$(x_k) \Rightarrow$ attacker(x) where f is a constructor, $p_1, \dots, p_n$ are fully tagged, x is one of $p_1, \dots, p_n$, and $f(p_1, \dots, p_n)$ is more general than every pattern of the form $f(\dots)$ in $sub(S_0)$.

A program $\mathcal{R}_0$ is *weakly tagged* if there exist a finite set of closed patterns S_0 and a set of patterns *tagGen* such that

W1. $\mathcal{R}_0$ is included in $\mathcal{R}_{\text{ProtAdv}}$.

W2. If two patterns p_1 and p_2 in *tagGen* unify, p'_1 is an instance of p_1 in $sub(S_0)$, and p'_2 is an instance of p_2 in $sub(S_0)$, then $p'_1 = p'_2$.

Intuitively, a pattern is top-tagged when its root function symbol is tagged (that is, it is of the form $f((ct, M_1, \dots, M_n), \dots))$. A pattern is fully tagged when all its function symbols are tagged.

We are going to show that all clauses generated by the resolution algorithm are in $\mathcal{R}_{\text{ProtAdv}}$. Basically, the clauses in $\mathcal{R}_{\text{Protocol}}$ satisfy two conditions: they can be instantiated into clauses whose patterns are in $sub(S_0)$ and they are tagged. Then, all patterns in clauses of $\mathcal{R}_{\text{Protocol}}$ are instances of *tagGen* and have instance in $sub(S_0)$. Property W2 allows us to show that this property is preserved by resolution: when unifying two patterns that satisfy the invariant, the result of the unification also satisfies the invariant, because the instances in $sub(S_0)$ of those two patterns are in fact equal. Thanks to this property, we can show that clauses obtained by resolution from clauses in $\mathcal{R}_{\text{Protocol}}$ are still in $\mathcal{R}_{\text{Protocol}}$. To prove termination, we show that the size of generated clauses decreases, for a suitable notion of size defined below. The clauses

$$E, \mathcal{P} \cup \{ 0 \}, \mathcal{M} \rightarrow E, \mathcal{P}, \mathcal{M} \quad \text{(Red Nil')}$$
$$E, \mathcal{P} \cup \{ !^i P \}, \mathcal{M} \rightarrow E[i \mapsto \mathrm{Id}_0], \mathcal{P} \cup \{ P\{\mathrm{Id}_0/i\} \}, \mathcal{M} \cup \{\mathrm{Id}_0\} \quad \text{(Red Repl')}$$
$$E, \mathcal{P} \cup \{ P \mid Q \}, \mathcal{M} \rightarrow E, \mathcal{P} \cup \{ P, Q \}, \mathcal{M} \quad \text{(Red Par')}$$
$$E, \mathcal{P} \cup \{ (\nu a : \ell) P \} \rightarrow E[a \mapsto E(\ell)], \mathcal{P} \cup \{ P \}, \mathcal{M} \cup \{M_1, \dots, M_n, a\} \quad \text{(Red Res')}$$
$$E, \mathcal{P} \cup \{ \overline{c}\langle M \rangle.Q \}, \mathcal{M} \rightarrow E, \mathcal{P} \cup \{ Q \}, \mathcal{M} \cup \{M\} \quad \text{(Red Out')}$$
$$E, \mathcal{P} \cup \{ c(x).P \}, \mathcal{M} \rightarrow E[x \mapsto E(M)], \mathcal{P} \cup \{ P\{M/x\} \}, \mathcal{M} \text{ if } M \in \mathcal{M} \quad \text{(Red In')}$$
$$\begin{aligned} &E, \mathcal{P} \cup \{ \mathit{let}\ x = g(M_1, \dots, M_n)\ \mathit{in}\ P\ \mathit{else}\ 0 \}, \mathcal{M} \rightarrow \\ &\quad E[x \mapsto E(M')], \mathcal{P} \cup \{ P\{M'/x\} \}, \mathcal{M} \cup \{M_1, \dots, M_n, M'\} \quad \text{(Red Destr 1')} \\ &\quad \text{if } g(M_1, \dots, M_n) \rightarrow M' \end{aligned}$$
$$E, \mathcal{P} \cup \{ \mathtt{event}(M).Q \}, \mathcal{M} \rightarrow E, \mathcal{P} \cup \{ Q \}, \mathcal{M} \cup \{M\} \quad \text{(Red Event')}$$

Figure 10: Special semantics for instrumented processes

of $\mathcal{R}_{\text{Constr}}$ and $\mathcal{R}_{\text{Destr}}$ are needed for constructors and destructors. Although they do not satisfy exactly the conditions for being in $\mathcal{R}_{\text{Protocol}}$, their resolution with a clause in $\mathcal{R}_{\text{Protocol}}$ yields a clause in $\mathcal{R}_{\text{Protocol}}$.

Let Params_{pk} and Params_{host} be the sets of arguments of pk resp. $host$ in the terms that occur in the trace of Condition C5. Let $\mathit{condense}(\mathcal{R}_0)$ be the set of clauses $\mathcal{R}$ obtained by $\mathcal{R} \leftarrow \emptyset$; for each $R \in \mathcal{R}_0$, $\mathcal{R} \leftarrow \mathit{elim}(\mathit{simplify}(R) \cup \mathcal{R})$. We first consider the case in which a single long-term key is used, that is, Params_{pk} and Params_{host} have at most one element. The results will be generalized to any number of keys at the end of this section. The next proposition shows that the initial clauses given to the resolution algorithm form a weakly tagged program.

Proposition 4 *If P_0 is a tagged protocol such that Params_{pk} and Params_{host} have at most one element and $P_0' = \mathrm{instr}(P_0)$, then $\mathit{condense}(\mathcal{R}_{P_0',\mathit{Init}})$ is a weakly tagged program.*

Proof sketch The fully detailed proof is very long (about 8 pages) so we give only a sketch here. A similar proof (for strong secrecy instead of secrecy and reachability) with more details can be found in the technical report [16, Appendix C].

We assume that different occurrences of restrictions and variables have different identifiers and identifiers different from free names and variables. In Figure 10, we define a special semantics for instrumented processes, which is only used as a tool in the proof. A semantic configuration consists of three components: an environment E mapping names and variables to patterns, a multiset of instrumented processes $\mathcal{P}$, and a set of terms $\mathcal{M}$. The semantics is defined as a reduction relation on semantic configurations. In this semantics, (νa) creates the name a, instead of a fresh name a'. Indeed, creating fresh names is useless, since the replication does not copy processes in this semantics, and the names are initially pairwise distinct.

Let $E_0 = \{a \mapsto a[\,] \mid a \in \mathit{fn}(P_0)\}$. We show that $E_0, \{P_0'\}, \mathit{fn}(P_0) \rightarrow^* E', \emptyset, \mathcal{M}'$, for some E' and $\mathcal{M}'$, such that the second argument of $\mathit{pencrypt}_p$ in $\mathcal{M}'$ is of the form $pk(M)$ and the arguments of pk and $host$ in $\mathcal{M}'$ are atomic constants in Params_{pk} and Params_{host} respectively. This result is obtained by simulating in the semantics of Figure 10 the trace of Condition C5. Moreover, the second argument of $\mathit{pencrypt}_p$ in $\mathcal{M}'$ is of the form $pk(M)$ by Condition C6 and the arguments of pk and $host$ in $\mathcal{M}'$ are atomic constants in Params_{pk} and Params_{host} respectively, by Condition C7 and definition of Params_{pk} and Params_{host}.

Let us define $S_0 = E'(\mathcal{M}') \cup \{b_0[\mathrm{Id}_0]\}$. If Params_{pk} is empty, we add some key k to it, so that $\mathit{Params}_{pk} = \{k\}$. Let c, c', c'', c''' be constants. If S_0 contains no instance of $\mathit{sencrypt}(x, y)$, we add $\mathit{sencrypt}((c, c'), c'')$ to S_0. If S_0 contains no instance of $\mathit{sencrypt}_p(x, y, z)$, we add $\mathit{sencrypt}_p((c, c'), c'', c''')$ to S_0. If S_0 contains no instance of $\mathit{pencrypt}_p(x, y, z)$, we

add $pencrypt_p((c,c'), pk(k), c'')$ to S_0. If S_0 contains no instance of $sign(x,y)$, we add $sign((c,c'), k)$ to S_0. If S_0 contains no instance of $nmrsign(x,y)$, we add $nmrsign((c,c'),k)$ to S_0. So S_0 is a finite set of closed patterns. Intuitively, S_0 is the set of patterns corresponding to closed terms that occur in the trace of Condition C5.

Let E_t be E in which all patterns $a[\ldots]$ are replaced with their corresponding term a. In all reductions $E_0, \{P'_0\}, fn(P_0) \to^* E, \mathcal{P}, \mathcal{M}$, all patterns of the form $a[\ldots]$ in the image of E are equal to $E(a)$, so $E \circ E_t = E$. We show the following result by induction on P:

> Let P be an instrumented process, subprocess of P'_0. Assume that $E_0, \{P'_0\}, fn(P_0) \to^* E, \mathcal{P} \cup \{E_t(P)\}, \mathcal{M} \to^* E', \emptyset, \mathcal{M}'$, and that there exists σ' such that $E'_{|dom(\rho)} = \sigma' \circ \rho$ and $patterns(\sigma' H) \subseteq sub(S_0)$. Then for all $R \in [\![P]\!]\rho H$, there exists σ'' such that $patterns(\sigma'' R) \subseteq sub(S_0)$.

Let $\rho_0 = \{a \mapsto a[\,] \mid a \in fn(P_0)\}$. By applying this result to $P = P'_0$, we obtain that for all clauses R in $[\![P'_0]\!]\rho_0\emptyset$, there exists a substitution σ such that $patterns(\sigma R) \subseteq sub(S_0)$.

Let

$$\begin{aligned} tagGen = \{ & f((ct_i, x_1, \ldots, x_n), x'_2, \ldots, x'_{n'}) \mid \\ & \quad f \in \{sencrypt, sencrypt_p, pencrypt_p, sign, nmrsign, h, mac\}\} \\ & \cup \{a[x_1, \ldots, x_n] \mid a \text{ name function symbol}\} \\ & \cup \{pk(x), host(x)\} \cup \{c \mid c \text{ atomic constant}\} \end{aligned}$$

We show the following result by induction on P:

> Assume that the patterns of the image of ρ and of H are fully tagged. Assume that P is an instrumented process, subprocess of P'_0. For all $R \in [\![P]\!]\rho H$, $patterns(R)$ are fully tagged.

This result relies on Condition C3 to show that the created terms are tagged, and on Condition C4 to show that the tags are checked. By applying this result to $P = P'_0$, we obtain that for all $R \in [\![P'_0]\!]\rho_0\emptyset$, the patterns of R are fully tagged.

By the previous results, $[\![P'_0]\!]\rho_0\emptyset \subseteq \mathcal{R}_{\text{Protocol}}$.

The clauses (Rf) are in $\mathcal{R}_{\text{Constr}}$. The clauses (Init) and (Rn) are in $\mathcal{R}_{\text{Protocol}}$ given the value of S_0. The clauses (Rg) for nth_i, $sdecrypt$, $sdecrypt_p$, $pdecrypt_p$, and $getmessage$ are:

$$\text{attacker}((x_1, \ldots, x_n)) \Rightarrow \text{attacker}(x_i) \qquad (nth_i)$$

$$\text{attacker}(sencrypt(x,y)) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(x) \qquad (sdecrypt)$$

$$\text{attacker}(sencrypt_p(x,y,z)) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(x) \qquad (sdecrypt_p)$$

$$\text{attacker}(pencrypt_p(x, pk(y), z)) \wedge \text{attacker}(y) \Rightarrow \text{attacker}(x) \qquad (pdecrypt_p)$$

$$\text{attacker}(sign(x,y)) \Rightarrow \text{attacker}(x) \qquad (getmessage)$$

and they are in $\mathcal{R}_{\text{Destr}}$ provided that all public-key encryptions in S_0 are of the form $pencrypt_p(p_1, pk(p_2), p_3)$ (that is, Condition C6). The clauses for $checksignature$ and $nmrchecksign$ are

$$\text{attacker}(sign(x,y)) \wedge \text{attacker}(pk(y)) \Rightarrow \text{attacker}(x) \qquad (checksignature)$$

$$\text{attacker}(nmrsign(x,y)) \wedge \text{attacker}(pk(y)) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(true) \qquad (nmrchecksign)$$

These two clauses are subsumed respectively by the clauses for $getmessage$ (given above) and $true$ (which is simply attacker($true$) since $true$ is a zero-ary constructor), so they are eliminated by $condense$, *i.e.*, they are not in $condense(\mathcal{R}_{P'_0,Init})$. (This is important, because they are not in $\mathcal{R}_{\text{Destr}}$.) Therefore all clauses in $condense(\mathcal{R}_{P'_0,Init})$ are in $\mathcal{R}_{\text{ProtAdv}}$, since the set of clauses $\mathcal{R}_{\text{ProtAdv}}$ is preserved by simplification, so we have Condition W1.

Different patterns in *tagGen* do not unify. Moreover, each pattern in *tagGen* has at most one instance in $\mathit{sub}(S_0)$. For $pk(x)$ and $host(x)$, this comes from the hypothesis that Params_{pk} and Params_{host} have at most one element. For atomic constants, this is obvious. (Their only instance is themselves.) For other patterns, this comes from the fact that the trace of Condition C5 executes each program point at most once, and that patterns created at different programs points are associated with different symbols (f, c) for $f((c, \ldots), \ldots)$ and a for $a[\ldots]$. (For $f((c, \ldots), \ldots)$, this comes from Condition C3. For $a[\ldots]$, this is because different restrictions use a different function symbol by construction of the clauses.) So we have Condition W2. □

The next proposition shows that saturation terminates for weakly tagged programs.

Proposition 5 *Let $\mathcal{R}_0$ be a set of clauses. If $\mathit{condense}(\mathcal{R}_0)$ is a weakly tagged program (Definition 20), then the computation of* $\mathsf{saturate}(\mathcal{R}_0)$ *terminates.*

Proof This result is very similar to [20, Proposition 8], so we give only a brief sketch and refer the reader to that paper for details.

We show by induction that all clauses R generated from $\mathcal{R}_0$ are in $\mathcal{R}_{\text{Protocol}} \cup \mathcal{R}_{\text{Constr}} \cup \mathcal{R}_{\text{Destr}}$ and the patterns of attacker facts in clauses R in $\mathcal{R}_{\text{Protocol}}$ are non-data.

First, by hypothesis, all clauses in $\mathit{condense}(\mathcal{R}_0)$ satisfy this property, by definition of weakly tagged programs and because of the decomposition of data constructors by *decomp*.

If we combine by resolution two clauses in $\mathcal{R}_{\text{Constr}} \cup \mathcal{R}_{\text{Destr}}$, we in fact combine a clause of $\mathcal{R}_{\text{Constr}}$ with a clause of $\mathcal{R}_{\text{Destr}}$. The resulting clause is a tautology by definition of $\mathcal{R}_{\text{Constr}}$ and $\mathcal{R}_{\text{Destr}}$, so it is eliminated by *elimtaut*.

Otherwise, we combine by resolution a clause R in $\mathcal{R}_{\text{Protocol}}$ with a clause R' such that $R' \in \mathcal{R}_{\text{Protocol}}$, $\mathsf{sel}(R') = \emptyset$, and $\mathsf{sel}(R) \neq \emptyset$, or $R' \in \mathcal{R}_{\text{Constr}}$, or $R' \in \mathcal{R}_{\text{Destr}}$. Let R'' be the clause obtained by resolution of R and R'. We show that the patterns of R'' are fully tagged, and for each σ such that $\mathit{patterns}(\sigma R) \subseteq \mathit{sub}(S_0)$, there exists σ'' such that $\mathit{patterns}(\sigma'' R'') \subseteq \mathit{sub}(S_0)$ and $\mathit{size}(\sigma'' R'') < \mathit{size}(\sigma R)$, where the size is defined as follows. The size of a pattern $\mathit{size}(p)$ is defined as usual, $\mathit{size}(\text{attacker}(p)) = \mathit{size}(\text{event}(p)) = \mathit{size}(p)$, and $\mathit{size}(F_1 \wedge \ldots \wedge F_n \Rightarrow F) = \mathit{size}(F_1) + \ldots + \mathit{size}(F_n) + \mathit{size}(F)$.

Let $R_s \in \mathit{simplify}(R'')$. The patterns of R_s are non-data fully tagged, $\mathit{patterns}(\sigma'' R_s) \subseteq \mathit{sub}(S_0)$, and $\mathit{size}(\sigma'' R_s) \leq \mathit{size}(\sigma'' R'') < \mathit{size}(\sigma R)$. So $R_s \in \mathcal{R}_{\text{Protocol}}$ and its patterns are non-data.

Moreover, for all generated clauses R, there exists σ such that $\mathit{size}(\sigma R)$ is smaller than the maximum initial value of $\mathit{size}(\sigma R)$ for a clause of the protocol. There is a finite number of such clauses (since $\mathit{size}(R) \leq \mathit{size}(\sigma R)$). So $\mathsf{saturate}(\mathcal{R}_0)$ terminates. □

Next, we show that $\mathsf{derivable}$ terminates when it is called on the result of the saturation of a weakly tagged program.

Proposition 6 *If F is a closed fact and $\mathcal{R}_1$ is a weakly tagged program simplified by simplify such that, for all $R \in \mathcal{R}_1$, $\mathsf{sel}_0(R) = \emptyset$, then* $\mathsf{derivable}(F, \mathcal{R}_1)$ *terminates.*

Proof We show the following property:

> For all calls $\mathsf{deriv}(R, \mathcal{R}, \mathcal{R}_1)$, $R = F \Rightarrow F$ or $R = \text{attacker}(p_1) \wedge \ldots \wedge \text{attacker}(p_n) \Rightarrow F$ where $p_1, \ldots, p_n$ are closed patterns.

This property is proved by induction. It is obviously true for the initial call to deriv, $\mathsf{deriv}(F \Rightarrow F, \emptyset, \mathcal{R}_1)$. For recursive calls to deriv, $\mathsf{deriv}(R'', \mathcal{R}, \mathcal{R}_1)$, the clause R'' is in $\mathit{simplify}'(R' \circ_{F_0} R)$, where $R' = \text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_k) \Rightarrow F'$ since $R' \in \mathcal{R}_1$ and $R = F \Rightarrow F$ or $R = \text{attacker}(p_1) \wedge \ldots \wedge \text{attacker}(p_n) \Rightarrow F$ where $p_1, \ldots, p_n$ are closed patterns, by induction hypothesis. After unification of F' and F_0, x_i is substituted by a closed pattern p'_i (subpattern

of F_0, and F_0 is closed since F_0 is a hypothesis of R), since x_i appears in F'. (If x_i did not appear in F', $\text{attacker}(x_i)$ would have been removed by *elimattx*.)

If $R = F \Rightarrow F$, $R' \circ_{F_0} R = \text{attacker}(p'_1) \wedge \ldots \wedge \text{attacker}(p'_k) \Rightarrow F$ has only closed patterns in its hypotheses, and so has the clause R'' in $\mathit{simplify}'(R' \circ_{F_0} R)$.

Otherwise, $R = \text{attacker}(p_1) \wedge \ldots \wedge \text{attacker}(p_n) \Rightarrow F$, $F_0 = \text{attacker}(p_i)$, and p_i is a closed pattern. We have $R' \circ_{F_0} R = \text{attacker}(p'_1) \wedge \ldots \wedge \text{attacker}(p'_k) \wedge \text{attacker}(p_1) \wedge \ldots \wedge \text{attacker}(p_{i-1}) \wedge \text{attacker}(p_{i+1}) \wedge \ldots \wedge \text{attacker}(p_n) \Rightarrow F$, which has only closed patterns in its hypotheses, and so has the clause R'' in $\mathit{simplify}'(R' \circ_{F_0} R)$. Moreover, $p'_1, \ldots, p'_k$ are disjoint subterms of p_i, therefore the total size of $p'_1, \ldots, p'_k$ is strictly smaller than the size of p_i. (If we had equality, F' would be a variable; this variable would occur in the hypothesis by definition of $\mathcal{R}_{\text{ProtAdv}}$, so R' would have been removed by *elimtaut*.) Therefore the total size of the patterns in the hypotheses strictly decreases. (The simplification function *simplify*$'$ cannot increase this size.) This decrease proves termination. □

From the previous results, we infer the termination of the algorithm for tagged protocols, when Params_{pk} and Params_{host} have at most one element. The general case can then be obtained as in [20]: we define a function OneKey which maps all elements of Params_{pk} and Params_{host} to a single atomic constant. When P_0 is a tagged protocol, $\text{OneKey}(P_0)$ is a tagged protocol in which Params_{pk} and Params_{host} are singletons. We consider a “less optimized algorithm” in which elimination of duplicate hypotheses and of tautologies are performed only for facts of the form $\text{attacker}(x)$, elimination of redundant hypotheses is not performed, and elimination of subsumed clauses is performed only for eliminating the destructor clauses for *checksignature* and *nmrchecksign*. We observe that the previous results still hold for the less optimized algorithm, with the same proof, so this algorithm terminates on $\text{OneKey}(P_0)$. All resolution steps possible for the less optimized algorithm applied to P_0 are possible for the less optimized algorithm applied to $\text{OneKey}(P_0)$ as well (more patterns are unifiable, and the remaining simplifications of the less optimized algorithm commute with applications of OneKey). Hence, the derivations from $\mathcal{R}_{P'_0,\mathit{Init}}$ are mapped by OneKey to derivations from $\mathcal{R}_{\text{OneKey}(P'_0),\mathit{Init}}$, which are finite, so derivations from $\mathcal{R}_{P'_0,\mathit{Init}}$ are also finite, so the less optimized algorithm terminates on P_0. We can then show that the original, fully optimized algorithm also terminates on P_0. So we finally obtain Proposition 3.

E General Correspondences

In this appendix, we prove Theorem 5. For simplicity, we assume that the function applications at the root of events are unary.

Lemma 17 *Let P_0 be a closed process and $P'_0 = \text{instr}'(P_0)$. Let Q be an Init-adversary and $Q' = \text{instrAdv}(Q)$. Assume that, in P_0, the arguments of events are function applications. Let f be a function symbol. Assume that there is a single occurrence of $\texttt{event}(f(_))$ in P_0 and this occurrence is under a replication. Consider any trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$. The multiset of session identifiers λ of events $\texttt{event}(f(_), \lambda)$ executed in $\mathcal{T}$ contains no duplicates.*

Proof Let us define the multiset $\mathit{SId}(P)$ by $\mathit{SId}(\texttt{event}(f(M), \lambda).P) = \{\lambda\} \cup \mathit{SId}(P)$ (for the given function symbol f), $\mathit{SId}(!^i P) = \emptyset$, and in all other cases, $\mathit{SId}(P)$ is the union of the $\mathit{SId}(P')$ for all immediate subprocesses P' of P. For a trace $\mathcal{T}$, let $\mathit{SId}(\mathcal{T})$ be the set of session identifiers λ of events $\texttt{event}(f(_), \lambda)$ executed in the trace $\mathcal{T}$.

We show that, for each trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$, $\mathit{SId}(\mathcal{T}) \cup \bigcup_{P \in \mathcal{P}'} \mathit{SId}(P) \cup S'$ contains no duplicates. The proof is by induction on the length of the trace.

For the empty trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S_0, E_0, \{P'_0, Q'\}$, $\mathit{SId}(\mathcal{T}) = \emptyset$ and $\mathit{SId}(P'_0) \cup \mathit{SId}(Q) = \emptyset$ by definition.

The reduction (Red Repl) moves at most one session identifier from S' to $\bigcup_{P\in\mathcal{P}'} \mathit{SId}(P)$ (without introducing duplicates since there is one occurrence of $\mathtt{event}(f(_), _)$). The reduction (Red Event) moves at most one session identifier from $\bigcup_{P\in\mathcal{P}'} \mathit{SId}(P)$ to $\mathit{SId}(\mathcal{T})$. The other reductions can only remove session identifiers from $\bigcup_{P\in\mathcal{P}'} \mathit{SId}(P)$ (by removing subprocesses). □

Lemma 18 *Let $P_0 = C[\mathtt{event}(f(M)).D[\mathtt{event}(f^{\mathrm{m-event}}(M,x).P]]$, where no replication occurs in $D[\,]$ above the hole $[\,]$, and the variables and names bound in P_0 are all pairwise distinct and distinct from free names. Assume that, in P_0, the arguments of events are function applications, and that there is a single occurrence of $\mathtt{event}(f(_))$ and of $\mathtt{event}(f^{\mathrm{m-event}}(_,_))$ in P_0.*

Let Q be an Init-adversary and $Q' = \mathrm{instrAdv}(Q)$. Let $P'_0 = \mathrm{instr}'(P_0)$. Consider a trace of P'_0: $\mathcal{T} = S_0, E_0, \mathcal{P}_0 = \{P'_0, Q'\} \rightarrow^* S_{\tau_\mathrm{f}}, E_{\tau_\mathrm{f}}, \mathcal{P}_{\tau_\mathrm{f}}$.

Then there exists a function ϕ^i such that a) if $\mathtt{event}(f^{\mathrm{m-event}}(p,p'),\lambda)$ is executed at step τ in $\mathcal{T}$ for some λ, p, p', τ, then $\mathtt{event}(f(p),\lambda)$ is executed at step $\phi^\mathrm{i}(\tau)$ in $\mathcal{T}$, b) ϕ^i is injective, and c) if $\phi^\mathrm{i}(\tau)$ is defined, then $\phi^\mathrm{i}(\tau) < \tau$.

Proof We denote by $S_\tau, E_\tau, \mathcal{P}_\tau$ the configuration at the step τ in the trace $\mathcal{T}$. Let

$$\begin{aligned}
S^1(\tau) &= \{(\lambda,p) \mid \mathtt{event}(f(p),\lambda) \text{ is executed in the first } \tau \text{ steps of } \mathcal{T}\},\\
S^2(\tau) &= \{(\lambda,p) \mid \mathtt{event}(f^{\mathrm{m-event}}(p,p'),\lambda) \text{ is executed in the first } \tau \text{ steps of } \mathcal{T}\}\\
S^3(\tau) &= \{(\lambda,p) \mid \mathtt{event}(f^{\mathrm{m-event}}(M,M'),\lambda) \text{ occurs not under } \mathtt{event}(f(M),\lambda) \text{ in}\\
&\qquad \mathcal{P}_\tau \text{ for } E_\tau(M) = p\}
\end{aligned}$$

For each τ, we show that $S^2(\tau) \cup S^3(\tau) \subseteq S^1(\tau)$.

- For $\tau = 0$, the sets $S^1(\tau)$, $S^2(\tau)$, and $S^3(\tau)$ are empty.
- If $S_\tau, E_\tau, \mathcal{P}_\tau \rightarrow S_{\tau+1}, E_{\tau+1}, \mathcal{P}_{\tau+1}$ using (Red Event) to execute $\mathtt{event}(f(M),\lambda)$, then the same $(\lambda, E_{\tau+1}(M))$ is added to $S^3(\tau+1)$ and to $S^1(\tau+1)$. Similarly, for (Red Event) executing $\mathtt{event}(f^{\mathrm{m-event}}(M,M'),\lambda)$, a pair $(\lambda, E_{\tau+1}(M))$ is moved from $S^3(\tau)$ to $S^2(\tau+1)$. These changes preserve the desired inclusion.
- Otherwise, if $S_\tau, E_\tau, \mathcal{P}_\tau \rightarrow S_{\tau+1}, E_{\tau+1}, \mathcal{P}_{\tau+1}$, then $S^1(\tau+1) = S^1(\tau)$, $S^2(\tau+1) = S^2(\tau)$, and $S^3(\tau+1) \subseteq S^3(\tau)$ (because some subprocesses may be removed by the reduction).

In particular, $S^2(\tau_\mathrm{f}) \subseteq S^1(\tau_\mathrm{f})$. By Lemma 17, there is a bijection ϕ_1 from the session labels λ of executed $\mathtt{event}(f(_),\lambda)$ events in $\mathcal{T}$ to the steps at which these events are executed in $\mathcal{T}$, and similarly ϕ_2 for $\mathtt{event}(f^{\mathrm{m-event}}(_,_),_)$ events. Let $\phi^\mathrm{i} = \phi_1 \circ \phi_2^{-1}$.

- If $\mathtt{event}(f^{\mathrm{m-event}}(p,p'),\lambda)$ is executed at step τ, $(\lambda,p) \in S^2(\tau_\mathrm{f}) \subseteq S^1(\tau_\mathrm{f})$, so $\mathtt{event}(f(p),\lambda)$ is executed at a certain step τ'. So $\phi_2(\lambda) = \tau$ and $\phi_1(\lambda) = \tau'$, so $\phi^\mathrm{i}(\tau)$ is defined and $\tau' = \phi^\mathrm{i}(\tau)$.
- Since ϕ_1 and ϕ_2^{-1} are injective, ϕ^i is injective.
- If $\phi^\mathrm{i}(\tau)$ is defined, the event $\mathtt{event}(f^{\mathrm{m-event}}(\sigma y, \sigma x),\lambda)$ is executed at step τ by (Red Event). So $(\lambda, \sigma y) \in S^3(\tau)$, where $\mathcal{P}_\tau$ corresponds to the state just before the event $\mathtt{event}(f^{\mathrm{m-event}}(\sigma y, \sigma x),\lambda)$ is executed. Hence $(\lambda,\sigma y) \in S^1(\tau)$ since $S^2(\tau) \cup S^3(\tau) \subseteq S^1(\tau)$. So $\mathtt{event}(f(\sigma y),\lambda)$ is executed at step $\tau' < \tau$. We have $\phi_2(\lambda) = \tau$ and $\phi_1(\lambda) = \tau'$, so $\phi^\mathrm{i}(\tau) = \tau' < \tau$. □

Proof (of Theorem 5) For each non-empty $\overline{jk}$, when $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, let $f_{\overline{jk}}$ be the root function symbol of $p_{\overline{jk}}$. We consider a modified process P_1 built from P_0 as follows. For each $\overline{jk}$ such that

$[\text{inj}]_{\overline{jk}} = \text{inj}$ and $\texttt{event}(f_{\overline{jk}}(M))$ occurs in P_0, we add another event $\texttt{event}(f_{\overline{jk}}^{\text{m-event}}(M, x_{\overline{jk}}))$ just under the definition of variable $x_{\overline{jk}}$ if $x_{\overline{jk}}$ is defined under $\texttt{event}(f_{\overline{jk}}(M))$ and just under $\texttt{event}(f_{\overline{jk}}(M))$ otherwise. Let $P'_1 = \text{instr}'(P_1)$. The process P'_1 is built from P'_0 as follows. For each $\overline{jk}$ such that $[\text{inj}]_{\overline{jk}} = \text{inj}$ and $\texttt{event}(f_{\overline{jk}}(M), i)$ occurs in P'_0, we add another event $\texttt{event}(f_{\overline{jk}}^{\text{m-event}}(M, x_{\overline{jk}}), i)$ just under the definition of variable $x_{\overline{jk}}$ if $x_{\overline{jk}}$ is defined under $\texttt{event}(f_{\overline{jk}}(M), i)$ and just under $\texttt{event}(f_{\overline{jk}}(M), i)$ otherwise. (When $[\text{inj}]_{\overline{jk}} = \text{inj}$, $x_{\overline{jk}} \in \mathit{dom}(\rho_{\overline{jrk}})$ where $\rho_{\overline{jrk}}$ is the environment added as argument of m-event facts in the clauses, so $x_{\overline{jk}}$ is defined either above $\texttt{event}(f_{\overline{jk}}(M), i)$ or under $\texttt{event}(f_{\overline{jk}}(M), i)$ without any replication between the event and the definition of $x_{\overline{jk}}$, since the domain of the environment given as argument to m-event is set at replications by substituting $\Box$ and not modified later.) We will show that P'_1 satisfies the desired correspondence. It is then clear that P'_0 also satisfies it.

The clauses $\mathcal{R}_{P'_1,\mathit{Init}}$ can be obtained from $\mathcal{R}'_{P'_0,\mathit{Init}}$ by replacing all facts $\text{m-event}(p, \rho)$ with

$$\text{m-event}(p, i) \wedge \bigwedge_{\overline{jk} \text{ such that } p = f_{\overline{jk}}(p') \text{ and } x_{\overline{jk}} \in \mathit{dom}(\rho)} \text{m-event}(f_{\overline{jk}}^{\text{m-event}}(p', \rho(x_{\overline{jk}})), i)$$

for some i, and adding clauses that conclude $\text{event}(f_{\overline{jk}}^{\text{m-event}}(\ldots), \ldots)$.

The clauses in $\mathsf{solve}_{P'_1,\mathit{Init}}$ can be obtained in the same way from $\mathsf{solve}'_{P'_0,\mathit{Init}}$. So we can define a function verify$'$ like verify with an additional argument $(x_{\overline{jkj'k'}})_{\overline{jkj'k'}}$ by adding $(x_{jk\overline{jkj'k'}})_{\overline{jkj'k'}}$ in the arguments of recursive call of Point V2.3 and replacing Point V2.1 with $\mathsf{solve}_{P'_1,\mathit{Init}}(\text{event}(p, i)) \subseteq \{H \wedge \bigwedge_{k=1}^{l_j} \text{m-event}(\arg_{jrk}, i_{jrk}) \Rightarrow \text{event}(\sigma_{jr}p'_j, i_{jr})$ for some H, $j \in \{1, \ldots, m\}$, r, i_{jrk}, and $(\rho_{jrk}, i_{jr}) \in \mathit{Env}_{jk}$ for all $k\}$ where $\arg_{jrk} = \sigma_{jr}p_{jk}$ if $[\text{inj}]_{jk} \neq \text{inj}$, and $\arg_{jrk} = f_{jk}^{\text{m-event}}(\sigma_{jr}p', \rho_{jrk}(x_{jk}))$ if $[\text{inj}]_{jk} = \text{inj}$ and $p_{jk} = f_{jk}(p')$. When $\text{verify}(q, (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$ is true, $\text{verify}'(q, (\mathit{Env}_{\overline{jk}})_{\overline{jk}}, (x_{\overline{jk}})_{\overline{jk}})$ is also true.

Let Q be an *Init*-adversary and $Q' = \text{instrAdv}(Q)$. Let E_0 such that $E_0(a) = a[\,]$ for all $a \in \mathit{dom}(E_0)$ and $\mathit{fn}(P'_1) \cup \mathit{Init} \subseteq \mathit{dom}(E_0)$. Let us now consider a trace of P'_1, $\mathcal{T} = S_0, E_0, \{P'_1, Q'\} \rightarrow^* S', E', \mathcal{P}'$.

By Lemma 18, for each non-empty $\overline{jk}$ such that $[\text{inj}]_{\overline{jk}} = \text{inj}$, there exists a function $\phi^{\text{i}}_{\overline{jk}}$ such that a) if $\texttt{event}(f_{\overline{jk}}^{\text{m-event}}(p, p'), \lambda)$ is executed at step τ in $\mathcal{T}$ for some λ, p, p', τ, then $\texttt{event}(f_{\overline{jk}}(p), \lambda)$ is executed at step $\phi^{\text{i}}_{\overline{jk}}(\tau)$ in $\mathcal{T}$, b) $\phi^{\text{i}}_{\overline{jk}}$ is injective, and c) if $\phi^{\text{i}}_{\overline{jk}}(\tau)$ is defined, then $\phi^{\text{i}}_{\overline{jk}}(\tau) < \tau$.

When $\psi_{\overline{jk}}$ is a family of functions from steps to steps in a trace, we define $\psi^{\circ}_{\overline{jk}}$ as follows:

- $\psi^{\circ}_{\epsilon}(\tau) = \tau$ for all τ;
- for all $\overline{jk}$, for all j and k, $\psi^{\circ}_{\overline{jk}jk} = \phi^{\text{i}}_{\overline{jk}jk} \circ \psi_{\overline{jk}jk} \circ \psi^{\circ}_{\overline{jk}}$ when $[\text{inj}]_{\overline{jk}jk} = \text{inj}$ and $\psi^{\circ}_{\overline{jk}jk} = \psi_{\overline{jk}jk} \circ \psi^{\circ}_{\overline{jk}}$ otherwise.

We show that, if $\text{verify}'(q', (\mathit{Env}_{\overline{jk}})_{\overline{jk}}, (x_{\overline{jk}})_{\overline{jk}})$ is true for

$$q' = \text{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left(\text{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\text{inj}]_{jk} q'_{jk} \right)$$

$$q'_{\overline{jk}} = \text{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\text{inj}]_{\overline{jk}jk} q'_{\overline{jk}jk}$$

then there exists a function $\psi_{\overline{jk}}$ for each $\overline{jk}$ such that

P1. For all τ, if the event $\texttt{event}(\sigma p, \lambda_\epsilon)$ is executed at step τ in $\mathcal{T}$, then there exist σ'' and $J = (j_{\overline{k}})_{\overline{k}}$ such that $\sigma'' p'_{j_\epsilon} = \sigma p$ and, for all non-empty $\overline{k}$, $\psi^\circ_{\mathrm{makejk}(\overline{k},J)}(\tau)$ is defined and $\texttt{event}(\sigma'' p_{\mathrm{makejk}(\overline{k},J)}, \lambda_{\overline{k}})$ is executed at step $\psi^\circ_{\mathrm{makejk}(\overline{k},J)}(\tau)$ in $\mathcal{T}$.

P2. For all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$ and $\psi_{\overline{jk}}(\tau)$ is defined, then $\texttt{event}(p''_1, \lambda'_1)$ is executed at step τ in $\mathcal{T}$, $\texttt{event}(f^{\mathrm{m-event}}_{\overline{jk}}(p''_2, \theta\rho(x_{\overline{jk}})), \lambda'_2)$ is executed at step $\psi_{\overline{jk}}(\tau)$ in $\mathcal{T}$, and $\theta i = \lambda'_1$ for some p''_1, p''_2, λ'_1, λ'_2, θ, and $(\rho, i) \in \mathit{Env}_{\overline{jk}}$, where $f_{\overline{jk}}$ is the root function symbol of $p_{\overline{jk}}$. (This property is used for proving injectivity and recentness.)

P3. For all non-empty $\overline{jk}$, if $\psi_{\overline{jk}}(\tau)$ is defined, then $\psi_{\overline{jk}}(\tau) \le \tau$.

The proof is by induction on q'.

- If $q' = \mathrm{event}(p)$ (that is, $m = 1$, $l_1 = 0$, and $p_1 = p$), we define $j_\epsilon = 1$ and $\sigma'' = \sigma$, so that $\sigma'' p'_{j_\epsilon} = \sigma p$. All other conditions hold trivially, since there is no non-empty $\overline{k}$.
- Otherwise, we define ψ_{jk} as follows.

  Using Point V2.1, by Theorem 3, P'_1 satisfies the correspondence

$$\mathrm{event}(p, i) \Rightarrow \bigvee_{j=1..m,r} \left(\mathrm{event}(\sigma_{jr} p'_j, i_{jr}) \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(\mathrm{arg}_{jrk}, i_{jrk}) \right) \tag{24}$$

  against *Init*-adversaries.

  Assume that $\texttt{event}(\sigma p, \lambda)$ is executed at step τ in $\mathcal{T}$ for some substitution σ. Let us consider the trace $\mathcal{T}$ cut just after step τ. By Correspondence (24), there exist σ', $j \in \{1, \dots, m\}$, and r such that $\sigma' \sigma_{jr} p'_j = \sigma p$, $\sigma' i_{jr} = \sigma\lambda = \lambda$, and for $k \in \{1, \dots, l_j\}$, there exists λ_k such that $\texttt{event}(\sigma' \mathrm{arg}_{jrk}, \lambda_k)$ is executed in the trace $\mathcal{T}$ cut after step τ. So the event $\texttt{event}(\sigma' \mathrm{arg}_{jrk}, \lambda_k)$ is executed at step $\tau_k \le \tau$ in $\mathcal{T}$. In this case, we define $\psi_{jk}(\tau) = \tau_k$ and $r(\tau) = r$.

  If $[\mathrm{inj}]_{jk} = \mathrm{inj}$, then $\texttt{event}(\sigma' \sigma_{jr} p_{jk}, \lambda_k)$ is executed as step $\phi^{\mathrm{i}}_{jk}(\psi_{jk}(\tau)) = \psi^\circ_{jk}(\tau)$.

  If $[\mathrm{inj}]_{jk} \ne \mathrm{inj}$, then $\mathrm{arg}_{jrk} = \sigma_{jr} p_{jk}$, so $\texttt{event}(\sigma' \sigma_{jr} p_{jk}, \lambda_k)$ is executed as step $\psi_{jk}(\tau) = \psi^\circ_{jk}(\tau)$.

  By construction, if $\psi_{jk}(\tau)$ is defined, then $\psi_{jk}(\tau) \le \tau$.

  When $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, we let $f_{\overline{jk}}$ be the root function symbol of $p_{\overline{jk}}$.

  By Point V2.3, for all j, r, k, $\mathrm{verify}'(\sigma_{jr} q'_{jk}, (\mathit{Env}_{jk\overline{jk}})_{\overline{jk}}, (x_{jk\overline{jk}})_{\overline{jk}})$ is true. So, by induction hypothesis, there exist functions $\psi_{jrk,\overline{jk}}$ such that

  - For all τ_k, if the event $\texttt{event}(\sigma' \sigma_{jr} p_{jk}, \lambda_k)$ is executed at step τ_k in $\mathcal{T}$, then there exist σ''_{jrk} and $J = (j_{jrk,\overline{k}})_{\overline{k}}$ such that $\sigma''_{jrk} \sigma_{jr} p_{jk} = \sigma' \sigma_{jr} p_{jk}$ and, for all non-empty $\overline{k}$, $\psi^\circ_{jrk,\mathrm{makejk}(\overline{k},J)}(\tau_k)$ is defined and $\texttt{event}(\sigma''_{jrk} \sigma_{jr} p_{jk\mathrm{makejk}(\overline{k},J)}, \lambda_{k\overline{k}})$ is executed at step $\psi^\circ_{jrk,\mathrm{makejk}(\overline{k},J)}(\tau_k)$ in $\mathcal{T}$.
  - For all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{jk\overline{jk}} = \mathrm{inj}$ and $\psi_{jrk,\overline{jk}}(\tau)$ is defined, then $\texttt{event}(p''_1, \lambda'_1)$ is executed at step τ in $\mathcal{T}$, $\texttt{event}(f^{\mathrm{m-event}}_{jk\overline{jk}}(p''_2, \theta\rho(x_{jk\overline{jk}})), \lambda'_2)$ is executed at step $\psi_{jrk,\overline{jk}}(\tau)$ in $\mathcal{T}$ and $\theta i = \lambda'_1$ for some p''_1, p''_2, λ'_1, λ'_2, θ, and $(\rho, i) \in \mathit{Env}_{jk\overline{jk}}$.
  - For all non-empty $\overline{jk}$, if $\psi_{jrk,\overline{jk}}(\tau)$ is defined, then $\psi_{jrk,\overline{jk}}(\tau) \le \tau$.

  We define $\psi_{jk\overline{jk}}(\tau) = \psi_{jrk,\overline{jk}}(\tau)$ for $r = r(\tau)$. Then we have $\psi^\circ_{jk\overline{jk}}(\tau) = \psi^\circ_{jrk,\overline{jk}}(\psi^\circ_{jk}(\tau))$ for $r = r(\tau)$.

  Therefore, for all τ, if $\texttt{event}(\sigma p, \lambda)$ is executed at step τ in $\mathcal{T}$, then

- there exist σ', $J_\epsilon = (j_{\overline{k}})_{\overline{k}}$, and r such that $j_\epsilon = j \in \{1,\dots,m\}$, $j_{\overline{k}}$ is undefined for all $\overline{k} \neq \epsilon$, $\sigma'\sigma_{jr}p'_j = \sigma p$, and, for all k, $\psi^\circ_{\mathrm{makejk}(k,J_\epsilon)}(\tau)$ is defined and $\mathtt{event}(\sigma'\sigma_{jr}p_{\mathrm{makejk}(k,J_\epsilon)},\lambda_k)$ is executed as step $\psi^\circ_{\mathrm{makejk}(k,J_\epsilon)}(\tau)$;
- for all k, there exist σ''_{jrk} and $J_k = (j_{k\overline{k}})_{k\overline{k}}$ such that $\sigma''_{jrk}\sigma_{jr}p_{jk} = \sigma'\sigma_{jr}p_{jk}$ and, for all non-empty $\overline{k}$, $\psi^\circ_{\mathrm{makejk}(k\overline{k},J_k)}(\tau)$ is defined and $\mathtt{event}(\sigma''_{jrk}\sigma_{jr}p_{\mathrm{makejk}(k\overline{k},J_k)},\lambda_{k\overline{k}})$ is executed at step $\psi^\circ_{\mathrm{makejk}(k\overline{k},J_k)}(\tau)$ in $\mathcal{T}$.

We define a family of indices J by merging J_ϵ and J_k for all k, that is, $J = (j_{\overline{k}})_{\overline{k}}$. Therefore, in order to obtain P1, it is enough to find a substitution σ'' such that $\sigma''p'_j = \sigma'\sigma_{jr}p'_j$, $\sigma''p_{jk} = \sigma'\sigma_{jr}p_{jk}$, and $\sigma''p_{jk\overline{jk}} = \sigma''_{jrk}\sigma_{jr}p_{jk\overline{jk}}$ for all non-empty $\overline{jk}$. Let us define σ_u as follows:

- For all $x \in \mathit{fv}(\sigma_{jr}p'_j) \cup \bigcup_k \mathit{fv}(\sigma_{jr}p_{jk})$, $\sigma_u x = \sigma' x$.
- For all k, for all $x \in \mathit{fv}(\sigma_{jr}q'_{jk}) \setminus \mathit{fv}(\sigma_{jr}p_{jk})$, $\sigma_u x = \sigma''_{jrk}x$.

By Point V2.2, these sets of variables are disjoint, so σ_u is well defined. Let $\sigma'' = \sigma_u\sigma_{jr}$.

We have $\sigma''p'_j = \sigma_u\sigma_{jr}p'_j = \sigma'\sigma_{jr}p'_j$ and $\sigma''p_{jk} = \sigma_u\sigma_{jr}p_{jk} = \sigma'\sigma_{jr}p_{jk}$. Since $\sigma''q'_{jk} = \sigma_u\sigma_{jr}q'_{jk}$, we just have to show that $\sigma_u\sigma_{jr}q'_{jk} = \sigma''_{jrk}\sigma_{jr}q'_{jk}$. We have $\sigma_u\sigma_{jr}p_{jk} = \sigma'\sigma_{jr}p_{jk} = \sigma''_{jrk}\sigma_{jr}p_{jk}$. Therefore, if $x \in \mathit{fv}(\sigma_{jr}p_{jk})$, then $\sigma_u x = \sigma''_{jrk}x$.[5] Hence, for all $x \in \mathit{fv}(\sigma_{jr}q'_{jk})$, $\sigma_u x = \sigma''_{jrk}x$, which proves that $\sigma_u\sigma_{jr}q'_{jk} = \sigma''_{jrk}\sigma_{jr}q'_{jk}$. Hence we obtain P1.

If $[\mathrm{inj}]_{jk} = \mathrm{inj}$ and $\psi_{jk}(\tau)$ is defined, then $\mathtt{event}(p''_1,\lambda'_1) = \mathtt{event}(\sigma p,\lambda)$ is executed at step τ in $\mathcal{T}$, $\mathtt{event}(f^{\mathrm{m-event}}_{jk}(p''_2,\theta\rho(x_{jk})),\lambda'_2) = \mathtt{event}(\sigma'\arg_{jrk},\lambda_k)$ is executed at step $\psi_{jk}(\tau)$ in $\mathcal{T}$, and $\theta i = \lambda'_1$ for some $p''_1 = \sigma p$, p''_2, $\lambda'_1 = \lambda$, $\lambda'_2 = \lambda_k$, $\theta = \sigma'$, and $(\rho,i) = (\rho_{jrk}, i_{jr}) \in \mathit{Env}_{jk}$. For all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{jk\overline{jk}} = \mathrm{inj}$ and $\psi_{jk\overline{jk}}(\tau)$ is defined, then $\mathtt{event}(p''_1,\lambda'_1)$ is executed at step τ in $\mathcal{T}$, $\mathtt{event}(f^{\mathrm{m-event}}_{jk\overline{jk}}(p''_2,\theta\rho(x_{jk\overline{jk}})),\lambda'_2)$ is executed at step $\psi_{jk\overline{jk}}(\tau)$ in $\mathcal{T}$, and $\theta i = \lambda'_1$ for some p''_1, p''_2, λ'_1, λ'_2, θ, and $(\rho,i) \in \mathit{Env}_{jk\overline{jk}}$. So we obtain P2.

If $\psi_{jk}(\tau)$ is defined, then $\psi_{jk}(\tau) \leq \tau$. For all non-empty $\overline{jk}$, if $\psi_{jk\overline{jk}}(\tau)$ is defined, then $\psi_{jk\overline{jk}}(\tau) \leq \tau$. Therefore, we have P3.

Let $q = \mathrm{event}(p) \Rightarrow \bigvee_{j=1}^{m}\left(\mathrm{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j}[\mathrm{inj}]_{jk}q_{jk}\right)$, and $q_{\overline{jk}} = \mathrm{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}}\bigwedge_{k=1}^{l_{\overline{jk}j}}[\mathrm{inj}]_{\overline{jk}jk}q_{\overline{jk}jk}$. By Hypothesis H1, $\mathrm{verify}'(q,(\mathit{Env}_{\overline{jk}})_{\overline{jk}},(x_{\overline{jk}})_{\overline{jk}})$ is true, so there exists a function $\psi_{\overline{jk}}$ for each $\overline{jk}$ such that P1, P2, and P3 are satisfied. Let $\phi_{\overline{jk}} = \psi^\circ_{\overline{jk}}$.

- By P1, for all τ, if the event $\mathtt{event}(\sigma p,\lambda_\epsilon)$ is executed at step τ in $\mathcal{T}$, then there exist σ' and $J = (j_{\overline{k}})_{\overline{k}}$ such that $\sigma'p'_{j_\epsilon} = \sigma p$ and, for all non-empty $\overline{k}$, $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$ is defined and $\mathtt{event}(\sigma'p_{\mathrm{makejk}(\overline{k},J)},\lambda_{\overline{k}})$ is executed at step $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$ in $\mathcal{T}$.

  Let us show recentness. Suppose that $[\mathrm{inj}]_{\mathrm{makejk}(\overline{k},J)} = \mathrm{inj}$. We show that the runtimes of $\mathrm{session}(\lambda_{\overline{k}\lceil})$ and $\mathrm{session}(\lambda_{\overline{k}})$ overlap. We have $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau) = \phi^{\mathrm{i}}_{\mathrm{makejk}(\overline{k},J)}(\psi_{\mathrm{makejk}(\overline{k},J)}(\phi_{\mathrm{makejk}(\overline{k}\lceil,J)}(\tau)))$. Let $\tau_1 = \phi_{\mathrm{makejk}(\overline{k}\lceil,J)}(\tau)$. Then $\psi_{\mathrm{makejk}(\overline{k},J)}(\tau_1)$ is defined. Hence, by P2, $e_1 = \mathtt{event}(p''_1,\lambda'_1)$ is executed at step τ_1 in $\mathcal{T}$, $e_2 = \mathtt{event}(f^{\mathrm{m-event}}_{\mathrm{makejk}(\overline{k},J)}(p''_2,\theta\rho(x_{\mathrm{makejk}(\overline{k},J)})),\lambda'_2)$ is executed at step $\tau_2 = \psi_{\mathrm{makejk}(\overline{k},J)}(\tau_1)$ in $\mathcal{T}$ by a reduction $S_{\tau_2}, E_{\tau_2}, \mathcal{P}_{\tau_2} \rightarrow S_{\tau_2+1}, E_{\tau_2+1}, \mathcal{P}_{\tau_2+1}$, and $\theta i = \lambda'_1$ for some p''_1, p''_2, λ'_1, λ'_2, θ, and $(\rho,i) \in \mathit{Env}_{\mathrm{makejk}(\overline{k},J)}$. Since the event $\mathtt{event}(\sigma'p_{\mathrm{makejk}(\overline{k}\lceil,J)},\lambda_{\overline{k}\lceil})$ is also executed at step $\tau_1 = \phi_{\mathrm{makejk}(\overline{k}\lceil,J)}(\tau)$, we have $\lambda'_1 = \lambda_{\overline{k}\lceil}$. By the properties of $\phi^{\mathrm{i}}_{\mathrm{makejk}(\overline{k},J)}$,

[5] This property does not hold in the presence of an equational theory (see Section 9.1). In that case, we conclude by the additional hypothesis mentioned in Section 9.1.

$\mathtt{event}(f_{\mathrm{makejk}(\overline{k},J)}(p''_2), \lambda'_2)$ is executed at step $\phi^{\mathrm{i}}_{\mathrm{makejk}(\overline{k},J)}(\tau_2) = \phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$. Moreover, $\mathtt{event}(\sigma' p_{\mathrm{makejk}(\overline{k},J)}, \lambda_{\overline{k}})$ is also executed at step $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$, so $\lambda'_2 = \lambda_{\overline{k}}$.

By Hypothesis H2, $\rho(x_{\mathrm{makejk}(\overline{k},J)})\{\lambda/i\}$ does not unify with $\rho(x_{\mathrm{makejk}(\overline{k},J)})\{\lambda'/i\}$ when $\lambda \neq \lambda'$, so i occurs in $\rho(x_{\mathrm{makejk}(\overline{k},J)})$, so $\lambda_{\overline{k}\lceil} = \lambda'_1 = \theta i$ occurs in $\theta\rho(x_{\mathrm{makejk}(\overline{k},J)})$, so $\lambda_{\overline{k}\lceil}$ occurs in e_2.

So e_2 is executed after the rule $S, E, \mathcal{P} \cup \{!^{i'} P'\} \to S \setminus \{\lambda_{\overline{k}\lceil}\}, E, \mathcal{P} \cup \{P'\{\lambda_{\overline{k}\lceil}/i'\}, !^{i'} P'\}$ in $\mathcal{T}$. Indeed, since $\lambda_{\overline{k}\lceil}$ occurs in the event e_2 executed at step τ_2, $\lambda_{\overline{k}\lceil} \in \mathit{SId}'(E_{\tau_2}) \cup \mathit{SId}'(\mathcal{P}_{\tau_2})$ where $\mathit{SId}'(\mathcal{P})$ (resp. $\mathit{SId}'(E)$) is the set of session identifiers λ that occur in $\mathcal{P}$ (resp. E). Moreover, $\mathit{SId}'(E_0) \cup \mathit{SId}'(\{P'_1, Q'\}) = \emptyset$, and the only rule that increases $\mathit{SId}'(E) \cup \mathit{SId}'(\mathcal{P})$ is $S, E, \mathcal{P} \cup \{!^i P'\} \to S \setminus \{\lambda\}, E, \mathcal{P} \cup \{P'\{\lambda/i\}, !^i P'\}$, which adds λ to $\mathit{SId}'(E) \cup \mathit{SId}'(\mathcal{P})$. Therefore, e_2 is executed after the beginning of the runtime of session$(\lambda_{\overline{k}\lceil})$.

Moreover, e_2 is executed at step $\tau_2 = \psi_{\mathrm{makejk}(\overline{k},J)}(\tau_1)$ and e_1 is executed at step τ_1 in $\mathcal{T}$, with $\psi_{\mathrm{makejk}(\overline{k},J)}(\tau_1) \leq \tau_1$, so e_2 is executed before $e_1 = \mathtt{event}(p''_1, \lambda_{\overline{k}\lceil})$.

So $e_2 = \mathtt{event}(f^{\mathrm{m-event}}_{\mathrm{makejk}(\overline{k},J)}(p''_2, \theta\rho(x_{\mathrm{makejk}(\overline{k},J)})), \lambda_{\overline{k}})$ is executed during the runtime of session$(\lambda_{\overline{k}\lceil})$, therefore the runtimes of session$(\lambda_{\overline{k}\lceil})$ and session$(\lambda_{\overline{k}})$ overlap.

- Let us show that, for all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, then $\psi_{\overline{jk}}$ is injective. Let τ_1 and τ_2 such that $\psi_{\overline{jk}}(\tau_1) = \psi_{\overline{jk}}(\tau_2)$. By P2, $\mathtt{event}(p''_1, \lambda'_1)$ is executed at step τ_1 in $\mathcal{T}$, $\mathtt{event}(f^{\mathrm{m-event}}_{\overline{jk}}(p''_3, \theta_1\rho_1(x_{\overline{jk}})), \lambda'_3)$ is executed at step $\psi_{\overline{jk}}(\tau_1)$ in $\mathcal{T}$, and $\theta_1 i_1 = \lambda'_1$ for some p''_1, p''_3, λ'_1, λ'_3, θ_1, and $(\rho_1, i_1) \in \mathit{Env}_{\overline{jk}}$. Also by P2, $\mathtt{event}(p''_2, \lambda'_2)$ is executed at step τ_2 in $\mathcal{T}$, $\mathtt{event}(f^{\mathrm{m-event}}_{\overline{jk}}(p''_4, \theta_2\rho_2(x_{\overline{jk}})), \lambda'_4)$ is executed at step $\psi_{\overline{jk}}(\tau_2)$ in $\mathcal{T}$, and $\theta_2 i_2 = \lambda'_2$ for some p''_1, p''_4, λ'_2, λ'_4, θ_2, and $(\rho_2, i_2) \in \mathit{Env}_{\overline{jk}}$. Since $\psi_{\overline{jk}}(\tau_1) = \psi_{\overline{jk}}(\tau_2)$, $\theta_1\rho_1(x_{\overline{jk}}) = \theta_2\rho_2(x_{\overline{jk}})$. By Hypothesis H2, this implies that $\theta_1 i_1 = \theta_2 i_2$, so $\lambda'_1 = \lambda'_2$. By Lemma 17, $\tau_1 = \tau_2$, which proves the injectivity of $\psi_{\overline{jk}}$.

- Let us show that, for all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, then $\phi_{\overline{jk}}$ is injective, by induction on the length of the sequence of indices $\overline{jk}$.

  For all j and k, if $[\mathrm{inj}]_{jk} = \mathrm{inj}$, then ϕ_{jk} is injective since ϕ^{i}_{jk}, ψ_{jk}, and ϕ_ϵ are injective.

  For all non-empty $\overline{jk}$, for all j and k, if $[\mathrm{inj}]_{\overline{jk}jk} = \mathrm{inj}$, then, by hypothesis, $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, so, by induction hypothesis, $\phi_{\overline{jk}}$ is injective. The functions $\phi^{\mathrm{i}}_{\overline{jk}jk}$ and $\psi_{\overline{jk}jk}$ are injective, so $\phi_{\overline{jk}jk}$ is also injective.

- For all $\overline{jk}$, for all j and k, if $\phi_{\overline{jk}jk}(\tau)$ is defined, then $\phi_{\overline{jk}}(\tau)$ is defined, and $\phi_{\overline{jk}jk}(\tau) \leq \phi_{\overline{jk}}(\tau)$, since $\phi^{\mathrm{i}}_{\overline{jk}jk}(\tau'') \leq \tau''$ and $\psi_{\overline{jk}jk}(\tau') \leq \tau'$ by P3, when they are defined.

  In particular, for all j and k, if $\phi_{jk}(\tau)$ is defined, then $\phi_{jk}(\tau) \leq \phi_\epsilon(\tau) = \tau$.

This concludes the proof of the desired recent correspondence. □

Proof (of Proposition 2) We have verify$(q, (\mathit{Env}_{\overline{jk}})_{\overline{jk}})$ with $\mathit{Env}_{jk} = \{(\rho_{jrk}, i_{jr}) \mid r \in \{1, \ldots, n_j\}\}$, because the first item implies V2.1, V2.2 holds trivially since q_{jk} reduces to event(p_{jk}), and V2.3 also holds since q_{jk} reduces to event(p_{jk}), so verify$(\sigma_{jr} q_{jk}, (\mathit{Env}_{jk\overline{jk}})_{\overline{jk}})$ holds by V1. The second item implies H2. So we have the result by Theorem 5. □

Automated Verification of Selected Equivalences for Security Protocols*

Bruno Blanchet
CNRS, École Normale Supérieure, Paris

Martín Abadi
University of California, Santa Cruz
and Microsoft Research, Silicon valley

Cédric Fournet
Microsoft Research, Cambridge

Abstract

In the analysis of security protocols, methods and tools for reasoning about protocol behaviors have been quite effective. We aim to expand the scope of those methods and tools. We focus on proving equivalences $P \approx Q$ in which P and Q are two processes that differ only in the choice of some terms. These equivalences arise often in applications. We show how to treat them as predicates on the behaviors of a process that represents P and Q at the same time. We develop our techniques in the context of the applied pi calculus and implement them in the tool ProVerif.

1 Introduction

Many security properties can be expressed as predicates on system behaviors. These properties include some kinds of secrecy properties (for instance, "the system never broadcasts the key k"). They also include correspondence properties (for instance, "if the system deletes file f, then the administrator must have requested it"). Such predicates on system behaviors are the focus of many successful methods for security analysis. In recent years, several tools have made it possible to prove many such predicates automatically or semi-automatically, even for infinite-state systems (e.g., [15, 40, 43]).

Our goal in this work is to expand the scope of those methods and tools. We aim to apply them to important security properties that have been hard to prove and that cannot be easily phrased as predicates on system behaviors. Many such properties can be written as equivalences. For instance, the secrecy of a boolean parameter x of a protocol $P(x)$ may be written as the equivalence $P(\text{true}) \approx P(\text{false})$. Similarly, as is common in theoretical cryptography, we may wish to express the correctness of a construction P by comparing it to an ideal functionality Q, writing $P \approx Q$. Here the relation $\approx$ represents observational equivalence: $P \approx Q$ means that no context (that is, no attacker) can distinguish P and Q. A priori, $P \approx Q$ is not a simple predicate on the behaviors of P or Q.

We focus on proving equivalences $P \approx Q$ in which P and Q are two variants of the same process obtained by selecting different terms on the left and on the right. In particular, $P(\text{true}) \approx P(\text{false})$ is such an equivalence, since $P(\text{true})$ and $P(\text{false})$ differ only in the choice of value for the parameter x. Both $P(\text{true})$ and $P(\text{false})$ are variants of a process that we may write $P(\mathsf{diff}[\text{true}, \text{false}])$; the two variants are obtained by giving different interpretations to $\mathsf{diff}[\text{true}, \text{false}]$, making it select either true or false.

*A preliminary version of this work was presented at the 20th IEEE Symposium on Logic in Computer Science (LICS 2005) [20].

Although the notation diff can be viewed as a simple informal abbreviation, we find that there is some value in giving it a formal status. We define a calculus that supports diff. With a careful definition of the operational semantics of this calculus, we can establish the equivalence $P(\text{true}) \approx P(\text{false})$ by reasoning about behaviors of $P(\mathsf{diff}[\text{true}, \text{false}])$.

In this operational semantics, $P(\mathsf{diff}[\text{true}, \text{false}])$ behaves like both $P(\text{true})$ and $P(\text{false})$ from the point of view of the attacker, as long as the attacker cannot distinguish $P(\text{true})$ and $P(\text{false})$. The semantics requires that the results of reducing $P(\text{true})$ and $P(\text{false})$ can be written as a process with subexpressions of the form $\mathsf{diff}[M_1, M_2]$. On the other hand, when $P(\text{true})$ and $P(\text{false})$ would do something that may differentiate them, the semantics specifies that the execution of $P(\mathsf{diff}[\text{true}, \text{false}])$ gets stuck. Hence, if no behavior of $P(\mathsf{diff}[\text{true}, \text{false}])$ ever gets stuck, then $P(\text{true}) \approx P(\text{false})$. Thus, we can prove equivalences by reasoning about behaviors, though not the behaviors of the original processes in isolation.

This technique applies not only to an equivalence $P(\text{true}) \approx P(\text{false})$ that represents the concealment of a boolean parameter, but to a much broader class of equivalences that arise in security analysis and that go beyond secrecy properties. In principle, every equivalence could be rewritten as an equivalence in our class: we might try to prove $P \approx Q$ by examining the behaviors of

$$\textit{if } \mathsf{diff}[\text{true}, \text{false}] = \text{true} \textit{ then } P \textit{ else } Q$$

This observation suggests that we should not expect completeness for an automatic technique. Indeed, the class of equivalences that we can establish automatically does not include some traditional bisimilarities. Accordingly, we aim to complement, not to replace, other proof methods. Moreover, we are primarily concerned with soundness and usefulness, and (in contrast with some related work [7, 23–25, 29, 38]) we emphasize simplicity and automation over generality. We believe, however, that the use of diff is not "just a hack", because diff is amenable to a rigorous treatment and because operators much like diff have already proved useful in other contexts—in particular, in elegant soundness proofs of information-flow type systems [44, 45]. Baudet's recent thesis includes a further study of diff and obtains a decidability result for processes without replication [12].

We implement our technique in the tool ProVerif [15]. This tool is a protocol analyzer for protocols written in the applied pi calculus [6], an extension of the pi calculus with function symbols that may represent cryptographic operations. Internally, ProVerif translates protocols to Horn clauses in classical logic, and uses resolution on these clauses. The mapping to classical logic (rather than linear logic) embodies a safe abstraction which ignores the number of repetitions of each action, and which is key to the treatment of infinite-state systems [19]. We extend the translation into Horn clauses and also the manipulation of these Horn clauses.

While the implementation in ProVerif requires a non-trivial development of theory and code, it is rather fruitful. It enables us to treat, automatically, interesting proofs of equivalences. In particular, as in previous ProVerif proofs, it does not require that all systems under consideration be finite-state. We demonstrate these points through small examples and larger applications.

Specifically, we apply our technique to an infinite-state analysis of the important Encrypted Key Exchange (EKE) protocol [13, 14]. (Password-based protocols such as EKE have attracted much attention in recent years, partly because of the difficulty of reasoning about them.) We also use our technique for checking certain equivalences that express authenticity properties in an example from the literature [8]. In other applications, automated proofs of equivalences serve as lemmas for manual proofs of other results. We illustrate this combination by revisiting proofs for the JFK protocol [9].

One of the main features of the approach presented in this paper is that it is compatible with the inclusion of equational theories on function symbols. We devote considerable attention to their proper, sound integration. Those equational theories serve in modelling properties of the underlying cryptographic operations; they are virtually necessary in many applications. For instance, an equational theory may describe a decryption function that returns "junk" when its

input is not a ciphertext under the expected key. Without equational theories, we may be able to model decryption only as a destructor that fails when there is a mismatch between ciphertext and key. Because the failure of decryption would be observable, it can result in false indications of attacks. Our approach overcomes this problem.

In contrast, a previous method for proving equivalences with ProVerif [17] does not address equivalences that depend on equational theories. Moreover, that method applies only to pairs of processes in which the terms that differ are global constants, not arbitrary terms. In these respects, the approach presented in this paper constitutes a clear advance. It enables significant proofs that were previously beyond the reach of automated techniques.

ProVerif belongs in a substantial body of work on sound, useful, but incomplete methods for protocol analysis. These methods rely on a variety of techniques from the programming-language literature, such as type systems, control-flow analyses, and abstract interpretation (e.g., [1, 22, 37, 42]). The methods are of similar power for proving predicates on behaviors [3, 21]. On the other hand, they typically do not target proofs of equivalences, or treat only specific classes of equivalences for particular equational theories.

The next section describes the process calculus that serves as setting for this work. Section 3 defines and studies observational equivalence. Section 4 contains some examples. Section 5 deals with equational theories. Section 6 explains how ProVerif maps protocols with `diff` to Horn clauses. Section 7 is concerned with proof techniques for those Horn clauses. Section 8 introduces a simple construct for breaking protocols into stages, as a convenience for applications. Section 9 describes applications. Section 10 mentions other related work and concludes. The Appendix contains proofs. The proof scripts for all examples and applications of this paper, as well as the tool ProVerif, are available at `http://www.di.ens.fr/~blanchet/obsequi/`.

2 The process calculus

This section introduces our process calculus, by giving its syntax and its operational semantics. This calculus is a combination of the original applied pi calculus [6] with one of its dialects [17]. This choice of calculus gives us the richness of the original applied pi calculus (in particular with regard to equational theories) while enabling us to leverage ProVerif.

2.1 Syntax and informal semantics

Figure 1 summarizes the syntax of our calculus. It defines a category of terms (data) and processes (programs). It assumes an infinite set of names and an infinite set of variables; a, b, c, k, s, and similar identifiers range over names, and x, y, and z range over variables. It also assumes a signature Σ (a set of function symbols, with arities and with associated definitions as explained below). We distinguish two categories of function symbols: constructors and destructors. We often write f for a constructor, g for a destructor, and h for a constructor or a destructor. Constructors are used for building terms. Thus, the terms $M, N, \dots$ are variables, names, and constructor applications of the form $f(M_1, \dots, M_n)$.

As in the applied pi calculus [6], terms are subject to an equational theory. Identifying an equational theory with its signature Σ, we write $\Sigma \vdash M = N$ for an equality modulo the equational theory, and $\Sigma \vdash M \neq N$ an inequality modulo the equational theory. (We write $M = N$ and $M \neq N$ for syntactic equality and inequality, respectively.) The equational theory is defined by a finite set of equations $\Sigma \vdash M_i = N_i$, where M_i and N_i are terms that contain only constructors and variables. The equational theory is then obtained from this set of equations by reflexive, symmetric, and transitive closure, closure by substitution (for any substitution σ, if $\Sigma \vdash M = N$ then $\Sigma \vdash \sigma M = \sigma N$), and closure by context application (if $\Sigma \vdash M = N$ then $\Sigma \vdash M'\{M/x\} = M'\{N/x\}$, where $\{M/x\}$ is the substitution that replaces x with M). We assume that there exist M and N such that $\Sigma \vdash M \neq N$.

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$D ::=$	term evaluations
M	term
$\mathrm{eval}\ h(D_1, \ldots, D_n)$	function evaluation
$P, Q, R ::=$	processes
$M(x).P$	input
$\overline{M}\langle N \rangle.P$	output
$\mathbf{0}$	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
$let\ x = D\ in\ P\ else\ Q$	term evaluation

Figure 1: Syntax for terms and processes

As previously implemented in ProVerif, destructors are partial, non-deterministic operations on terms that processes can apply. More precisely, the semantics of a destructor g of arity n is given by a finite set $\mathrm{def}_\Sigma(g)$ of rewrite rules $g(M'_1, \ldots, M'_n) \rightarrow M'$, where $M'_1, \ldots, M'_n, M'$ are terms that contain only constructors and variables, the variables of M' are bound in $M'_1, \ldots, M'_n$, and variables are subject to renaming. Then $g(M_1, \ldots, M_n)$ is defined if and only if there exists a substitution σ and a rewrite rule $g(M'_1, \ldots, M'_n) \rightarrow M'$ in $\mathrm{def}_\Sigma(g)$ such that $M_i = \sigma M'_i$ for all $i \in \{1, \ldots, n\}$, and in this case $g(M_1, \ldots, M_n) \rightarrow \sigma M'$. In order to avoid distinguishing constructors and destructors in the definition of term evaluation, we let $\mathrm{def}_\Sigma(f)$ be $\{f(x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n)\}$ when f is a constructor of arity n.

The process $let\ x = D\ in\ P\ else\ Q$ tries to evaluate D; if this succeeds, then x is bound to the result and P is executed, else Q is executed. Here the reader may ignore the prefix eval which may occur in D, since eval f and f have the same semantics when f is a constructor, and destructors are used only with eval. In Section 5, we distinguish eval f and f in order to indicate when terms are evaluated.

Using constructors, destructors, and equations, we can model various data structures (tuples, lists, ...) and cryptographic primitives (shared-key encryption, public-key encryption, signatures, ...). Typically, destructors represent primitives that can visibly succeed or fail, while equations apply to primitives that always succeed but may sometimes return "junk". For instance, suppose that one can detect whether shared-key decryption succeeds or fails; then we would use a constructor *enc*, a destructor *dec*, and the rewrite rule $dec(enc(x, y), y) \rightarrow x$. Otherwise, we would use two constructors *enc* and *dec*, and the equations $dec(enc(x, y), y) = x$ and $enc(dec(x, y), y) = x$. The second equation prevents that the equality test $enc(dec(M, N), N) = M$ reveal that M must be a ciphertext under N. (The first equation is standard; the second is not, but it holds for block ciphers.) We refer to previous work [6, 17] for additional explanations and examples.

The rest of the syntax of Figure 1 is fairly standard pi calculus. The input process $M(x).P$ inputs a message on channel M, and executes P with x bound to the input message. The output process $\overline{M}\langle N \rangle.P$ outputs the message N on channel M and then executes P. (We allow M to be an arbitrary term; we could require that M be a name, as is frequently done, and adapt other definitions accordingly.) The nil process $\mathbf{0}$ does nothing and is sometimes omitted in examples. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents an unbounded number of copies of P in parallel. The restriction $(\nu a)P$ creates a new name a, and

$$
\begin{array}{l}
M \Downarrow M \\
\text{eval } h(D_1, \ldots, D_n) \Downarrow \sigma N \\
\quad \text{if } h(N_1, \ldots, N_n) \rightarrow N \in \mathrm{def}_\Sigma(h), \\
\quad \text{and } \sigma \text{ is such that for all } i,\ D_i \Downarrow M_i \text{ and } \Sigma \vdash M_i = \sigma N_i
\end{array}
$$

$$
\begin{array}{ll}
P \mid 0 \equiv P & P \equiv P \\
P \mid Q \equiv Q \mid P & Q \equiv P \ \Rightarrow\ P \equiv Q \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & P \equiv Q,\ Q \equiv R \ \Rightarrow\ P \equiv R \\
(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P & P \equiv Q \ \Rightarrow\ P \mid R \equiv Q \mid R \\
(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q & P \equiv Q \ \Rightarrow\ (\nu a)P \equiv (\nu a)Q \\
\quad \text{if } a \notin \mathit{fn}(P) &
\end{array}
$$

$$
\begin{array}{lr}
\overline{N}\langle M\rangle.Q \mid N'(x).P \ \rightarrow\ Q \mid P\{M/x\} & \\
\quad \text{if } \Sigma \vdash N = N' & \text{(Red I/O)} \\
\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow P\{M/x\} & \\
\quad \text{if } D \Downarrow M & \text{(Red Fun 1)} \\
\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow Q & \\
\quad \text{if there is no } M \text{ such that } D \Downarrow M & \text{(Red Fun 2)} \\
!P \ \rightarrow\ P \mid !P & \text{(Red Repl)} \\
P \ \rightarrow\ Q \ \Rightarrow\ P \mid R \ \rightarrow\ Q \mid R & \text{(Red Par)} \\
P \ \rightarrow\ Q \ \Rightarrow\ (\nu a)P \ \rightarrow\ (\nu a)Q & \text{(Red Res)} \\
P' \equiv P,\ P \ \rightarrow\ Q,\ Q \equiv Q' \ \Rightarrow\ P' \ \rightarrow\ Q' & \text{(Red } \equiv\text{)}
\end{array}
$$

Figure 2: Semantics for terms and processes

then executes P. The syntax does not include the conditional *if* $M = N$ *then* P *else* Q, which can be defined as *let* $x = \mathrm{equals}(M, N)$ *in* P *else* Q where x is a fresh variable and equals is a binary destructor with the rewrite rule $\mathrm{equals}(x, x) \rightarrow x$. We always include this destructor in Σ.

We write $\mathit{fn}(P)$ and $\mathit{fv}(P)$ for the sets of names and variables free in P, respectively, which are defined as usual. A process is closed if it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables. An evaluation context C is a closed context built from $[\,]$, $C \mid P$, $P \mid C$, and $(\nu a)C$.

2.2 Formal semantics

The rules of Figure 2 axiomatize the reduction relation for processes ($\rightarrow_\Sigma$), thus defining the operational semantics of our calculus. Auxiliary rules define term evaluation ($\Downarrow_\Sigma$) and the structural congruence relation ($\equiv$); this relation is useful for transforming processes so that the reduction rules can be applied. Both $\equiv$ and $\rightarrow_\Sigma$ are defined only on closed processes. We write $\rightarrow_\Sigma^*$ for the reflexive and transitive closure of $\rightarrow_\Sigma$, and $\rightarrow_\Sigma^*\equiv$ for its union with $\equiv$. When Σ is clear from the context, we abbreviate $\rightarrow_\Sigma$ and $\Downarrow_\Sigma$ to $\rightarrow$ and $\Downarrow$, respectively.

This semantics differs in minor ways from the semantics of the applied pi calculus [6]. In particular, we do not substitute equals for equals in structural congruence, but only in a controlled way in certain rules. Thus, the rule for I/O does not require a priori that the input and output channels be equal: it explicitly uses the equational theory to compare them. We also use a reduction rule (Red Repl) for modelling replication, instead of the more standard, but essentially equivalent, structural congruence rule. This weakening of structural congruence in favor of the reduction relation is designed to simplify our proofs.

$\overline{N}\langle M\rangle.Q \mid N'(x).P \ \rightarrow\ Q \mid P\{M/x\}$ (Red I/O)
if $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N')$ and $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$

$\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow P\{\mathsf{diff}[M_1, M_2]/x\}$ (Red Fun 1)
if $\mathsf{fst}(D) \Downarrow M_1$ and $\mathsf{snd}(D) \Downarrow M_2$

$\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q \rightarrow Q$ (Red Fun 2)
if there is no M_1 such that $\mathsf{fst}(D) \Downarrow M_1$ and
there is no M_2 such that $\mathsf{snd}(D) \Downarrow M_2$

Figure 3: Generalized rules for biprocesses

3 Observational equivalence

In this section we introduce diff formally and establish a sufficient condition for observational equivalence. We first recall the standard definition of observational equivalence from the pi calculus:

Definition 1 The process P *emits on* M ($P \downarrow_M$) if and only if $P \equiv C[\overline{M'}\langle N\rangle.R]$ for some evaluation context C that does not bind $\mathit{fn}(M)$ and $\Sigma \vdash M = M'$.

(Strong) observational equivalence $\sim$ is the largest symmetric relation $\mathcal{R}$ on closed processes such that $P\ \mathcal{R}\ Q$ implies

1. if $P \downarrow_M$ then $Q \downarrow_M$;
2. if $P \rightarrow P'$ then $Q \rightarrow Q'$ and $P'\ \mathcal{R}\ Q'$ for some Q';
3. $C[P]\ \mathcal{R}\ C[Q]$ for all evaluation contexts C.

Weak observational equivalence $\approx$ is defined similarly, with $\rightarrow^* \downarrow_M$ instead of $\downarrow_M$ and $\rightarrow^*$ instead of $\rightarrow$.

Intuitively, a context may represent an adversary, and two processes are observationally equivalent when no adversary can distinguish them.

Next we introduce a new calculus that can represent pairs of processes that have the same structure and differ only by the terms and term evaluations that they contain. We call such a pair of processes a *biprocess*. The grammar for the calculus is a simple extension of the grammar of Figure 1, with additional cases so that $\mathsf{diff}[M, M']$ is a term and $\mathsf{diff}[D, D']$ is a term evaluation. We also extend the definition of contexts to permit the use of diff, and sometimes refer to contexts without diff as plain contexts.

Given a biprocess P, we define two processes $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$, as follows: $\mathsf{fst}(P)$ is obtained by replacing all occurrences of $\mathsf{diff}[M, M']$ with M and $\mathsf{diff}[D, D']$ with D in P, and similarly, $\mathsf{snd}(P)$ is obtained by replacing $\mathsf{diff}[M, M']$ with M' and $\mathsf{diff}[D, D']$ with D' in P. We define $\mathsf{fst}(D)$, $\mathsf{fst}(M)$, $\mathsf{snd}(D)$, and $\mathsf{snd}(M)$ similarly. Our goal is to show that the processes $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$ are observationally equivalent:

Definition 2 Let P be a closed biprocess. We say that P *satisfies observational equivalence* when $\mathsf{fst}(P) \sim \mathsf{snd}(P)$.

The semantics for biprocesses is defined as in Figure 2 with generalized rules (Red I/O), (Red Fun 1), and (Red Fun 2) given in Figure 3. Reductions for biprocesses bundle those for processes: if $P \rightarrow Q$ then $\mathsf{fst}(P) \rightarrow \mathsf{fst}(Q)$ and $\mathsf{snd}(P) \rightarrow \mathsf{snd}(Q)$. Conversely, however, reductions in $\mathsf{fst}(P)$ and $\mathsf{snd}(P)$ need not correspond to any biprocess reduction, in particular when they do not match up. Our first theorem shows that the processes are equivalent when this does not happen.

Definition 3 We say that the biprocess P is *uniform* when $\mathsf{fst}(P) \to Q_1$ implies that $P \to Q$ for some biprocess Q with $\mathsf{fst}(Q) \equiv Q_1$, and symmetrically for $\mathsf{snd}(P) \to Q_2$.

Theorem 1 *Let P_0 be a closed biprocess. If, for all plain evaluation contexts C and reductions $C[P_0] \to^* P$, the biprocess P is uniform, then P_0 satisfies observational equivalence.*

Proof Let P be a closed biprocess such that $C[P] \to^*\equiv Q$ always yields a uniform biprocess Q, and consider the relation

$$\mathcal{R} = \{(\mathsf{fst}(Q), \mathsf{snd}(Q)) \mid C[P] \to^*\equiv Q\}$$

In particular, we have $\mathsf{fst}(P)\ \mathcal{R}\ \mathsf{snd}(P)$, so we can show that P satisfies observational equivalence by establishing that the relation $\mathcal{R}' = \mathcal{R} \cup \mathcal{R}^{-1}$ meets the three conditions of Definition 1. By symmetry, we focus on $\mathcal{R}$. Assume $\mathsf{fst}(Q)\ \mathcal{R}\ \mathsf{snd}(Q)$.

1. Assume $\mathsf{fst}(Q) \downarrow_M$, and let $T_M = M(x).\overline{c}\langle c\rangle$ for some fresh name c. As usual in the pi calculus, the predicate $_\downarrow_M$ tests the ability to send any message on M, hence for any plain process Q_i, we have $Q_i \downarrow_M$ if and only if $Q_i \mid T_M \to R_i \mid \overline{c}\langle c\rangle$ for some R_i.

   Here, we have $\mathsf{fst}(Q) \mid T_M \to R_1 \mid \overline{c}\langle c\rangle$ for some R_1. The reductions $C[P] \to^*\equiv Q$ imply $C[P] \mid T_M \to^*\equiv Q \mid T_M$. By hypothesis (with the context $C[_] \mid T_M$), $Q \mid T_M$ is uniform, hence $Q \mid T_M \to Q'$ for some Q' with $\mathsf{fst}(Q') \equiv R_1 \mid \overline{c}\langle c\rangle$. Since c does not occur anywhere in Q, by case analysis on this reduction step with our semantics for biprocesses we obtain $Q' \equiv R \mid \overline{c}\langle c\rangle$ for some biprocess R. Thus, we obtain $\mathsf{snd}(Q) \mid T_M \to \mathsf{snd}(R) \mid \overline{c}\langle c\rangle$, and finally $\mathsf{snd}(Q) \downarrow_M$.

2. If $\mathsf{fst}(Q) \to Q'_1$ then, by uniformity, we have $Q \to Q'$ with $\mathsf{fst}(Q') = Q'_1$. Thus, $C[P] \to^*\equiv\to Q'$ and, by definition of $\mathcal{R}$, we obtain $\mathsf{fst}(Q')\ \mathcal{R}\ \mathsf{snd}(Q')$. Finally, by definition of the semantics of biprocesses, $Q \to Q'$ implies $\mathsf{snd}(Q) \to \mathsf{snd}(Q')$.

3. Let C' be a plain evaluation context. By definition of the semantics of biprocesses, $C[P] \to^*\equiv Q$ always implies $C'[C[P]] \to^*\equiv C'[Q]$, hence $C'[\mathsf{fst}(Q)] = \mathsf{fst}(C'[Q])\ \mathcal{R}\ \mathsf{snd}(C'[Q]) = C'[\mathsf{snd}(Q)]$. □

Our plan is to establish the hypothesis of Theorem 1 by automatically verifying that all the biprocesses P in question meet conditions that imply uniformity. The next corollary details those conditions, which guarantee that a communication and an evaluation, respectively, succeed in $\mathsf{fst}(P)$ if and only if they succeed in $\mathsf{snd}(P)$:

Corollary 1 *Let P_0 be a closed biprocess. Suppose that, for all plain evaluation contexts C, all evaluation contexts C', and all reductions $C[P_0] \to^* P$,*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$, then $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N')$ if and only if $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$,*

2. *if $P \equiv C'[let\ x = D\ in\ Q\ else\ R]$, then there exists M_1 such that $\mathsf{fst}(D) \Downarrow M_1$ if and only if there exists M_2 such that $\mathsf{snd}(D) \Downarrow M_2$.*

Then P_0 satisfies observational equivalence.

Proof We show that P is uniform, then we conclude by Theorem 1. Let us show that, if $\mathsf{fst}(P) \to P'_1$ then there exists a biprocess P' such that $P \to P'$ and $\mathsf{fst}(P') \equiv P'_1$. The case for $\mathsf{snd}(P) \to P'_2$ is symmetric.

By induction on the derivation of $\mathsf{fst}(P) \to P'_1$, we first show that there exist C, Q, and Q'_1 such that $P \equiv C[Q]$, $P'_1 \equiv \mathsf{fst}(C)[Q'_1]$, and $\mathsf{fst}(Q) \to Q'_1$ using one of the four process rules (Red

I/O), (Red Fun 1), (Red Fun 2), or (Red Repl): every step in this derivation trivially commutes with fst, except for structural steps that involve a parallel composition and a restriction, in case $a \in fn(P)$ but $a \notin fn(\mathsf{fst}(P))$. In that case, we use a preliminary renaming from a to some fresh $a' \notin fn(P)$.

For each of these four rules, relying on a hypothesis of Corollary 1, we find Q' such that $\mathsf{fst}(Q') = Q'_1$ and $Q \rightarrow Q'$ using the corresponding biprocess rule:

(Red I/O): We have $Q = \overline{N}\langle M \rangle.R \mid N'(x).R'$ with $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N')$ and $Q'_1 = \mathsf{fst}(R) \mid \mathsf{fst}(R')\{\mathsf{fst}(M)/x\}$. For $Q' = R \mid R'\{M/x\}$, we have $\mathsf{fst}(Q') = Q'_1$ and, by hypothesis 1, $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$, hence $Q \rightarrow Q'$.

(Red Fun 1): We have $Q = let\ x = D\ in\ R\ else\ R'$ with $\mathsf{fst}(D) \Downarrow M_1$ and $Q'_1 = \mathsf{fst}(R)\{M_1/x\}$. By hypothesis 2, $\mathsf{snd}(D) \Downarrow M_2$ for some M_2. We take $Q' = R\{\mathsf{diff}[M_1, M_2]\}$, so that $\mathsf{fst}(Q') = Q'_1$ and $Q \rightarrow Q'$.

(Red Fun 2): We have $Q = let\ x = D\ in\ R\ else\ R'$ with no M_1 such that $\mathsf{fst}(D) \Downarrow M_1$ and $Q'_1 = \mathsf{fst}(R')$. By hypothesis 2, there is no M_2 such that $\mathsf{snd}(D) \Downarrow M_2$. We obtain $Q \rightarrow Q'$ for $Q' = R'$.

(Red Repl): We have $Q = !R$ and $Q'_1 = \mathsf{fst}(R) \mid !\mathsf{fst}(R)$. We take $Q' = R \mid !R$, so that $\mathsf{fst}(Q') = Q'_1$ and $Q \rightarrow Q'$.

To conclude, we take the biprocess $P' = C[Q']$ and the reduction $P \rightarrow P'$. □

Thus, we have a sufficient condition for observational equivalence of biprocesses. This condition is essentially a reachability condition on biprocesses. Starting in Section 5, we adapt existing techniques for reasoning about processes in order to prove this condition. The condition is however not necessary: as suggested in the introduction, if $P \sim Q$, then *if* $\mathsf{diff}[\text{true}, \text{false}] = \text{true}$ *then* P *else* Q satisfies observational equivalence, but Theorem 1 and Corollary 1 will not enable us to prove this fact.

4 Examples in the applied pi calculus

This section illustrates our approach by revisiting examples of observational equivalences presented with the applied pi calculus [6]. Interestingly, all those equivalences can be formulated using biprocesses, proved via Theorem 1 and, it turns out, verified automatically by ProVerif. Section 9 presents more complex examples.

We begin with equivalences that can be expressed with biprocesses that perform a single output, of the form $(\nu a_1, \ldots, a_k)\overline{c}\langle M \rangle$ where c is a name that does not occur in $a_1, \ldots, a_k$ or in M. Intuitively, such equivalences state that no environment can differentiate $\mathsf{fst}(M)$ from $\mathsf{snd}(M)$ without knowing some name in $a_1, \ldots, a_k$. Such equivalences on terms under restrictions are called static equivalences [6]. They arise when one considers attackers that first intercept a series of messages, then attempt to differentiate two configurations of the protocol by computing on those messages without interacting with the protocol further. Here, the term M may be a tuple $\mathsf{diff}[M_1, M'_1], \ldots, \mathsf{diff}[M_n, M'_n]$ that collects all pairs of intercepted messages, and $a_1, \ldots, a_k$ may be names that represent all local secrets and fresh values used by the protocol.

Static equivalences play a central role in the extension of proof techniques from the pure pi calculus to the applied pi calculus. In particular, observational equivalence in the applied pi calculus can be reduced to standard pi calculus requirements plus static equivalences [6]. In other words, proofs of observational equivalences can be decomposed into lemmas that deal with terms and general arguments that relate processes with different structures; the former depend on the signature, while the latter come from the pure pi calculus. In our experience, a

large fraction of the proof effort is typically devoted to those lemmas on terms, and Theorem 1 is a good tool for establishing them.

Example 1 Consider a cryptographic hash function, modelled as a constructor h with neither rewrite rule nor equation. The environment should not be able to distinguish a freshly generated random value, modelled as a fresh name a, from its hash $h(a)$ [6, Section 4.2]. Formally, using the automated technique presented in this paper, we verify that the biprocess $(\nu a)\overline{c}\langle \mathsf{diff}[a, h(a)]\rangle$ satisfies equivalence. On the other hand, $P = (\nu a, a')\overline{c}\langle (a, \mathsf{diff}[a', h(a)])\rangle$ does not satisfy equivalence: although both processes emit a pair of fresh terms, the environment can distinguish one process from the other by computing a hash of the first element of the pair and comparing it to the second element of the pair, using the context

$$C[_] = c(x, y).\mathit{if}\ \ y = h(x)\ \ \mathit{then}\ \ \overline{d}\langle c\rangle\ \ \mathit{else}\ 0 \mid [_]$$

With our biprocess semantics, $C[P]$ performs a (Red I/0) step then gets stuck on the test $(\nu a, a')\mathit{if}\ \ \mathsf{diff}[a', h(a)] = h(a)\ \ \mathit{then}\ \ \overline{d}\langle c\rangle\ \ \mathit{else}\ 0.$ □

Example 2 Diffie-Hellman computations used in key agreement protocols can be expressed in terms of a constant $\mathtt{b}$, a binary constructor $\hat{}$, and the equation $(\mathtt{b}\hat{}x)\hat{}y = (\mathtt{b}\hat{}y)\hat{}x$ [4, 6]. With this signature, we verify that

$$(\nu a_1, a_2, a_3)\overline{c}\langle (\mathtt{b}\hat{}a_1, \mathtt{b}\hat{}a_2, \mathsf{diff}[(\mathtt{b}\hat{}a_1)\hat{}a_2, \mathtt{b}\hat{}a_3])\rangle$$

satisfies equivalence. This equivalence closely corresponds to the Decisional Diffie-Hellman assumption often made by cryptographers; it is also the main lemma in the proof of [6, Theorem 3]. Intuitively, even if the environment is given access to the exponentials $\mathtt{b}\hat{}a_1$ and $\mathtt{b}\hat{}a_2$, those values are (apparently) unrelated to the Diffie-Hellman secret $(\mathtt{b}\hat{}a_1)\hat{}a_2$, since the environment cannot distinguish this secret from the exponential of any fresh unrelated value a_3. □

The remaining two examples concern applications beyond proofs of static equivalences.

Example 3 Non-deterministic encryption is a variant of public-key encryption that further protects the secrecy of the plaintext by embedding some additional, fresh value in each encryption. It can be modelled using three functions for public-key decryption, public-key encryption, and public-key derivation, linked by the equation

$$\mathit{pdec}(\mathit{penc}(x, \mathit{pk}(y), z), y) = x$$

where z is the additional parameter for the encryption. A key property of non-deterministic encryption is that, without knowledge of the decryption key, ciphertexts appear to be unrelated to the plaintexts, even if the attacker knows the plaintexts and the encryption key. A strong version of this property is that the ciphertexts cannot be distinguished from freshly generated random values. Formally, we state that

$$(\nu s)(\overline{c}\langle \mathit{pk}(s)\rangle \mid !c'(x).(\nu a)\overline{c}\langle \mathsf{diff}[\mathit{penc}(x, \mathit{pk}(s), a), a]\rangle)$$

satisfies equivalence. This biprocess is more complex than those presented above; instead of a single output, it performs a first output to reveal the public key $\mathit{pk}(s)$ (but not s!), then repeatedly inputs a term x from the environment and either outputs its encryption under $\mathit{pk}(s)$ or outputs a fresh, unrelated name. Thus, a single biprocess represents the family of static equivalences that relate a series of non-deterministic encryptions for any series of plaintext to a series of fresh, independent names. (Formally, each such equivalence can be obtained as a corollary of this biprocess equivalence, by applying the congruence property of equivalence for the particular context that sends the plaintexts of values on channel c' and reads the encryption key and encryptions on channel c.) □

Example 4 Biprocesses can also be used for relating an abstract specification of a cryptographic primitive with its implementation in terms of lower-level functions. As an example, we consider the construction of message authentication codes (MACs) for messages of arbitrary length, as modelled in the applied pi calculus [6, Section 6]. MAC functions are essentially keyed hash functions; MACs should not be subject to tampering or forgery. More formally, the usage of MACs can be captured via a little protocol that generates MACs on demand and checks them:

$$P_0 = (\nu k)(!c'(x).\overline{c}\langle x, mac(k,x)\rangle \\ \mid c(x,y).if\ y = mac(k,x)\ then\ \overline{c''}\langle x\rangle)$$

The unforgeability of MACs means that the MAC checker succeeds and forwards a message x on c'' only if a MAC has been generated for x by sending it to the MAC generator on c'.

Let P be P_0 with the term $\mathsf{diff}[mac(k,x), impl(k,x)]$ instead of the two occurrences of $mac(k,x)$. For a given signature with no equation for mac, a function $impl$ may be said to implement mac correctly when P satisfies equivalence. With this formulation, we can verify the correctness of the second construction considered in [6], $impl(k,x) = f(k, f(k,x))$, with equation $f(k,(x,y)) = h(f(k,x),y)$, where f is a keyed hash function that iterates a compression function h on the message blocks. We can also confirm that the first construction considered in [6], $impl(k,x) = f(k,x)$ with the same equation $f(k,(x,y)) = h(f(k,x),y)$, is subject to a standard extension attack: anyone that obtains the MAC $impl(k,N_1)$ can produce the MAC $impl(k,(N_1,N_2))$ as $h(impl(k,N_1),N_2)$ for any message extension N_2 without knowing k. □

5 Modelling equations with rewrite rules

We handle equations by translating from a signature with equations to a signature without equations. This translation is designed to ease implementation: with it, resolution can continue to rely on ordinary syntactic unification, and remains very efficient. Although our technique is general and automatic, it does have limitations: it does not apply to some equational theories, in particular theories with associative symbols such as XOR. (It may be possible to handle some of those theories by shifting from syntactic unification to unification modulo the theory in question, at the cost of increased complexity.)

5.1 Definitions

We consider an auxiliary rewriting system on terms, $\mathcal{S}$, that defines partial normal forms. The terms manipulated by $\mathcal{S}$ do not contain diff, but they may contain variables. The rules of $\mathcal{S}$ do not contain names and do not have a single variable on the left-hand side. We say that a term is irreducible by $\mathcal{S}$ when none of the rewrite rules of $\mathcal{S}$ applies to it; we say that the set of terms $\mathcal{M}$ is in normal form relatively to $\mathcal{S}$ and Σ, and write $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$, if and only if all terms of $\mathcal{M}$ are irreducible by $\mathcal{S}$ and, for all subterms N_1 and N_2 of terms of $\mathcal{M}$, if $\Sigma \vdash N_1 = N_2$ then $N_1 = N_2$. Intuitively, we allow for the possibility that terms may have several irreducible forms (see Example 6 below), requiring that $\mathcal{M}$ use irreducible forms consistently. This requirement implies, for instance, that if the rewrite rule $f(x,x) \to x$ applies modulo the equational theory to a term $f(N_1,N_2)$ then N_1 and N_2 are identical and the rule $f(x,x) \to x$ also applies without invoking the equational theory. We extend the definition of $\mathrm{nf}_{\mathcal{S},\Sigma}(\cdot)$ to sets of processes: $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P})$ if and only if the set of terms that appear in processes in $\mathcal{P}$ is in normal form.

For a signature Σ' (without equations), we define evaluation on open terms as a relation

$D \Downarrow' (M, \sigma)$, where σ collects instantiations of D obtained by unification:

$$M \Downarrow' (M, \emptyset)$$

$$\begin{array}{l} \text{eval } h(D_1, \ldots, D_n) \Downarrow' (\sigma_u N, \sigma_u \sigma') \\ \quad \text{if } (D_1, \ldots, D_n) \Downarrow' ((M_1, \ldots, M_n), \sigma'), \\ \quad h(N_1, \ldots, N_n) \rightarrow N \text{ is in } \mathrm{def}_{\Sigma'}(h) \text{ and} \\ \quad \sigma_u \text{ is a most general unifier of } (M_1, N_1), \ldots, (M_n, N_n) \end{array}$$

$$\begin{array}{l} (D_1, \ldots, D_n) \Downarrow' ((\sigma_n M_1, \ldots, \sigma_n M_{n-1}, M_n), \sigma_n \sigma) \\ \quad \text{if } (D_1, \ldots, D_{n-1}) \Downarrow' ((M_1, \ldots, M_{n-1}), \sigma) \text{ and } \sigma D_n \Downarrow' (M_n, \sigma_n) \end{array}$$

As suggested in Section 2, we rely on eval for indicating term evaluations: while $f(M_1, \ldots, M_n) \Downarrow' (f(M_1, \ldots, M_n), \emptyset)$, deriving eval $f(M_1, \ldots, M_n) \Downarrow' (M, \sigma)$ requires an application of a rewrite rule for the constructor f.

We let addeval$(M_1, \ldots, M_n)$ be the tuple of term evaluations obtained by adding eval before each function symbol of $M_1, \ldots, M_n$. Using these definitions, we describe when a signature Σ' with rewrite rules models another signature Σ with equations:

Definition 4 Let Σ and Σ' be signatures on the same function symbols. We say that Σ' models Σ if and only if

1. The equational theory of Σ' is syntactic equality: $\Sigma' \vdash M = N$ if and only if $M = N$.

2. The constructors of Σ' are the constructors of Σ; their definition $\mathrm{def}_{\Sigma'}(f)$ contains the rule $f(x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n)$, plus perhaps other rules such that there exists a rewriting system $\mathcal{S}$ on terms that satisfies the following properties:

   S1. If $M \rightarrow N$ is in $\mathcal{S}$, then $\Sigma \vdash M = N$.

   S2. If $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$, then for any term M there exists M' such that $\Sigma \vdash M' = M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$.

   S3. If $f(N_1, \ldots, N_n) \rightarrow N$ is in $\mathrm{def}_{\Sigma'}(f)$, then $\Sigma \vdash f(N_1, \ldots, N_n) = N$.

   S4. If $\Sigma \vdash f(M_1, \ldots, M_n) = M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M_1, \ldots, M_n, M\})$, then there exist σ and $f(N_1, \ldots, N_n) \rightarrow N$ in $\mathrm{def}_{\Sigma'}(f)$ such that $M = \sigma N$ and $M_i = \sigma N_i$ for all $i \in \{1, \ldots, n\}$.

3. The destructors of Σ' are the destructors of Σ, with a rule $g(M'_1, \ldots, M'_n) \rightarrow M'$ in $\mathrm{def}_{\Sigma'}(g)$ for each $g(M_1, \ldots, M_n) \rightarrow M$ in $\mathrm{def}_{\Sigma}(g)$ and each addeval$(M_1, \ldots, M_n, M) \Downarrow' ((M'_1, \ldots, M'_n, M'), \sigma)$.

Condition 1 says that the equational theory of Σ' is trivial. In Condition 2, Properties S1 and S2 concern the relation of $\mathcal{S}$ and Σ. Property S1 guarantees that all rewrite rules of $\mathcal{S}$ are sound according to the equational theory of Σ. Property S2 requires there are "enough" normal forms: that for every term M there is an $\mathcal{S}$-irreducible Σ-equal term M', and that M' can be chosen consistently with a set $\mathcal{M}$ in normal form. Properties S3 and S4 concern the definition of constructors in Σ'. Property S3 guarantees that the rewrite rules that define the constructors are sound according to the equational theory of Σ. Property S4 requires that there are "enough" rewrite rules: basically, that when $M_1, \ldots, M_n$ are in normal form, every normal form of $f(M_1, \ldots, M_n)$ can be generated by applying a rewrite rule for f in Σ' to $f(M_1, \ldots, M_n)$. Finally, according to Condition 3, the definition of destructors in Σ' can be computed by applying the rewrite rules of constructors in Σ' to the definition of destructors in Σ.

According to this definition, we deal with any equations on f in Σ by evaluating f once in Σ'. (We use eval markers in expressions accordingly: eval f and f represent f before and

after this evaluation, respectively.) This characteristic entails a limitation of our approach. For instance, suppose that we have $f(f'(x)) = f'(f(x))$ in the equational theory of Σ, and we want a Σ' that models Σ. In Σ', we should equate $f'(f(\dots f(a)))$ and $f(\dots f(f'(a)))$ by one reduction step, so we need one rewrite rule for each length of sequence of applications of f, so $\mathrm{def}_{\Sigma'}(f')$ cannot be finite. Associative symbols like XOR pose a similar problem.

5.2 Examples

The following two examples illustrate the definitions of Section 5.1. ProVerif handles these examples automatically, using the approach of Section 5.3.

Example 5 Suppose that Σ has the constructors *enc* and *dec* with the equations

$$\mathit{dec}(\mathit{enc}(x, y), y) = x \qquad \mathit{enc}(\mathit{dec}(x, y), y) = x$$

In Σ', we adopt the rewrite rules:

$$\begin{array}{ll} \mathit{dec}(x, y) \rightarrow \mathit{dec}(x, y) & \mathit{enc}(x, y) \rightarrow \mathit{enc}(x, y) \\ \mathit{dec}(\mathit{enc}(x, y), y) \rightarrow x & \mathit{enc}(\mathit{dec}(x, y), y) \rightarrow x \end{array}$$

We have that Σ' models Σ for the rewriting system $\mathcal{S}$ with rules $\mathit{dec}(\mathit{enc}(x, y), y) \rightarrow x$ and $\mathit{enc}(\mathit{dec}(x, y), y) \rightarrow x$, and a single normal form for every term. □

Example 6 In order to model the Diffie-Hellman equation of Example 2, we define Σ' with three rewrite rules:

$$\mathtt{b} \rightarrow \mathtt{b} \qquad x\,\hat{}\,y \rightarrow x\,\hat{}\,y \qquad (\mathtt{b}\,\hat{}\,x)\,\hat{}\,y \rightarrow (\mathtt{b}\,\hat{}\,y)\,\hat{}\,x$$

and use an empty $\mathcal{S}$. Intuitively, applying ^ to $(\mathtt{b}\,\hat{}\,x)$ and y yields both possible forms of $(\mathtt{b}\,\hat{}\,x)\,\hat{}\,y$ modulo the equational theory, $(\mathtt{b}\,\hat{}\,x)\,\hat{}\,y$ and $(\mathtt{b}\,\hat{}\,y)\,\hat{}\,x$. Hence, a term M may have several irreducible forms M' that satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M'\})$ and $\Sigma \vdash M' = M$: one can choose $(\mathtt{b}\,\hat{}\,N)\,\hat{}\,N'$ or $(\mathtt{b}\,\hat{}\,N')\,\hat{}\,N$. □

5.3 Algorithms

Next we explain a method for finding, for a given signature Σ, a signature Σ' that models Σ and a corresponding rewriting system $\mathcal{S}$. This method is embodied in algorithms that, when they terminate, yield the definition of $\mathrm{def}_{\Sigma'}(f)$ for each constructor of Σ. The definition of $\mathrm{def}_{\Sigma'}(g)$ for each destructor of Σ follows from Condition 3 of Definition 4. These algorithms do not always terminate because, for some equational theories, they generate an unbounded number of rewrite rules. However, they often terminate in practice, as our examples illustrate; moreover, Lemma 7 in Appendix A.2 establishes a termination result for a significant class of theories, the convergent subterm theories [5].

Our first algorithm handles convergent (terminating and confluent) theories. It applies, for instance, to Example 5. Here and elsewhere, we write T for a term context (a term with a hole).

Algorithm 1 (Convergent theories) Let $M_i = N_i$ (for $i \in \{1, \dots, m\}$) be the equations that define the equational theory of Σ. Let $\mathcal{S}$ be defined by the rewrite rules $M_i \rightarrow N_i$. Assume that $\mathcal{S}$ is convergent, and let $M\!\downarrow$ be the normal form of M relatively to $\mathcal{S}$.

When E is a set of rewrite rules, we define *normalize*(E) by
– replacing each rule $f(M_1, \dots, M_n) \rightarrow N$ of E with $f(M_1\!\downarrow, \dots, M_n\!\downarrow) \rightarrow N\!\downarrow$;
– removing rules of the form $M \rightarrow M$ from E;
– if $M \rightarrow N$ is in E, removing all other rules of the form $T[\sigma M] \rightarrow T[\sigma N]$ from E.

Let $E = \mathit{normalize}(\mathcal{S})$.

Repeat until E reaches a fixpoint
  For each pair of rules $M \rightarrow M'$ and $N \rightarrow N'$ in E and each T
    such that $M' = T[M'']$, M'' is not a variable,
    and σ_u is the most general unifier of M'' and N,
      set $E = \mathit{normalize}(E \cup \{\sigma_u M \rightarrow \sigma_u T[\sigma_u N']\})$.
For each constructor f,
  $\mathrm{def}_{\Sigma'}(f) = \{f(M_1, \dots, M_n) \rightarrow N \in E\} \cup \{f(x_1, \dots, x_n) \rightarrow f(x_1, \dots, x_n)\}$.

In this algorithm, we add to E new rewrite rules obtained by composing two rewrite rules of E, until a fixpoint is reached. Lemma 7 in Appendix A.2 shows that a fixpoint is reached immediately for convergent subterm theories.

Before running the algorithm, we can check that $\mathcal{S}$ is convergent as follows.

- The termination of $\mathcal{S}$ can be established via a reduction ordering $>$, by showing that if $M \rightarrow M'$ is in $\mathcal{S}$, then $M > M'$. In the implementation, we use a lexicographic path ordering.
- The confluence of $\mathcal{S}$ can be established via the critical-pair theorem, by showing that all critical pairs are joinable [31].

Alternatively, one could use the Knuth-Bendix completion algorithm in order to transform a rewriting system into a convergent one.

Our next algorithm handles linear theories, such as that of Example 6.

Algorithm 2 (Linear theories) Let Σ be a signature such that all equations of Σ are linear: each variable occurs at most once in the left-hand side and at most once in the right-hand side. Let $\mathcal{S}$ be empty.

When E is a set of rewrite rules, we define *normalize*(E) by:
– removing rules of the form $M \rightarrow M$ from E;
– if $M \rightarrow N$ is in E, removing all other rules of the form $T[\sigma M] \rightarrow T[\sigma N]$ from E.

Let $E = \mathit{normalize}(\{M \rightarrow N, N \rightarrow M \mid M = N \text{ is an equation of } \Sigma\})$.
Repeat until E reaches a fixpoint
  For each pair of rules $M \rightarrow M'$ and $N \rightarrow N'$ in E and each T
    such that $M' = T[M'']$, M'' and N are not variables,
    and σ_u is the most general unifier of M'' and N,
      set $E = \mathit{normalize}(E \cup \{\sigma_u M \rightarrow \sigma_u T[\sigma_u N']\})$.
  For each pair of rules $M \rightarrow M'$ and $N \rightarrow N'$ in E and each T
    such that $N = T[N'']$, M' and N'' are not variables,
    and σ_u is the most general unifier of M' and N'',
      set $E = \mathit{normalize}(E \cup \{\sigma_u T[\sigma_u M] \rightarrow \sigma_u N'\})$.
For each constructor f,
  $\mathrm{def}_{\Sigma'}(f) = \{f(M_1, \dots, M_n) \rightarrow N \in E\} \cup \{f(x_1, \dots, x_n) \rightarrow f(x_1, \dots, x_n)\}$.

In this algorithm, when two rewrite rules of E have a critical pair with one another, we compose them and add the result to E.

Algorithms 1 and 2 are similar. The main difference is that Algorithm 1 performs additional optimizations that are sound for convergent rewriting systems but not for linear equational theories. In particular, in the initial definition of E, Algorithm 1 considers rewrite rules oriented only in the direction of $\mathcal{S}$, while Algorithm 2 considers both directions. Furthermore, in Algorithm 1, *normalize* reduces the right-hand sides and the strict subterms of the left-hand sides of rewrite rules by $\mathcal{S}$, while Algorithm 2 does not include this reduction. As a consequence, the second way of combining rules of E in Algorithm 2 is not necessary in Algorithm 1, since the rules thus created would be reduced by *normalize* into an instance of an already existing rule.

Our final algorithm combines the two previous ones:

Algorithm 3 (Union) Let Σ be a signature.
Split the set of equations of Σ into subsets E_i that use disjoint sets of constructors.
Let E_{conv} be the union of those subsets E_i that we can prove convergent.
Let E_{lin} be the union of those subsets E_i that are linear and are not in E_{conv}.
If some subsets E_i are neither in E_{conv} nor in E_{lin}, then fail.
Apply Algorithm 1 to E_{conv}, obtaining the rewriting system $\mathcal{S}_{\text{conv}}$ and the definition $\text{def}_{\Sigma'}(f)$ of the constructors of E_{conv}.
Apply Algorithm 2 to E_{lin}, obtaining the definition $\text{def}_{\Sigma'}(f)$ of the constructors of E_{lin}.
Let $\mathcal{S} = \mathcal{S}_{\text{conv}}$.

The following theorem says that these three algorithms are correct. It is proved in Appendix A.

Theorem 2 *If Algorithm 1, 2, or 3 produces a signature Σ' from a signature Σ, then Σ' models Σ.*

5.4 Reductions with equations and rewrite rules

From this point on, we assume that Σ' models Σ. We extend equality modulo Σ from terms to biprocesses and term evaluations: $\Sigma \vdash P = P'$ if and only if P' can be obtained from P by replacing some of its subterms M (not containing diff or eval) with subterms equal modulo Σ. We define $\Sigma \vdash D = D'$ similarly. Finally, we define $P \rightarrow_{\Sigma',\Sigma} P'$ as $P \rightarrow_{\Sigma} P'$ except that signature Σ' is used for reduction rules (Red I/O) and (Red Fun 1)—signature Σ is still used for (Red Fun 2).

We say that a biprocess P_0 is unevaluated when every term in P_0 is either a variable or $\mathsf{diff}[a, a]$ for some name a. Hence, every function symbol in P_0 must be in a term evaluation and prefixed by eval. For any biprocess P, we can build an unevaluated biprocess unevaluated(P) by introducing a term evaluation for every non-trivial term and a diff for every name (with $P \approx$ unevaluated(P)). For instance, the unevaluated biprocess built from the process of Example 3 is:

$$\begin{aligned}&(\nu s)(\mathit{let}\ y = \text{eval}\ pk(\mathsf{diff}[s,s])\ \mathit{in}\ \overline{\mathsf{diff}[c,c]}\langle y\rangle \mid\\ &\quad !\mathsf{diff}[c',c'](x).(\nu a)\\ &\quad\ \mathit{let}\ z = \mathsf{diff}[\text{eval}\ enc(x, \text{eval}\ pk(\mathsf{diff}[s,s]), \mathsf{diff}[a,a]), \mathsf{diff}[a,a]]\ \mathit{in}\ \overline{\mathsf{diff}[c,c]}\langle z\rangle)\end{aligned}$$

Lemma 1 *Let P_0 be a closed, unevaluated biprocess. If $P_0 \rightarrow^*_{\Sigma}\equiv P'_0$, $\Sigma \vdash P'_0 = P'$, and $\text{nf}_{\mathcal{S},\Sigma}(\{P'\})$, then $P_0 \rightarrow^*_{\Sigma',\Sigma}\equiv P'$. Conversely, if $P_0 \rightarrow^*_{\Sigma',\Sigma}\equiv P'$ then there exists P'_0 such that $\Sigma \vdash P'_0 = P'$ and $P_0 \rightarrow^*_{\Sigma}\equiv P'_0$.*

This lemma gives an operational correspondence between $\rightarrow_{\Sigma}$ and $\rightarrow_{\Sigma',\Sigma}$. A similar lemma holds for processes instead of biprocesses, and can be used for extending previous proof techniques for secrecy [3] and correspondence [16] properties, so that they apply under equational theories. These extensions are implemented in ProVerif. We do not detail them further since we focus on equivalences in this paper. Using Lemma 1, we obtain:

Lemma 2 *A closed biprocess P_0 satisfies the conditions of Corollary 1 if and only if, for all plain evaluation contexts C, all evaluation contexts C', and all reductions* unevaluated$(C[P_0]) \rightarrow^*_{\Sigma',\Sigma} P$*, we have*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\mathsf{fst}(N) = \mathsf{fst}(N')$, then $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$,*
2. *if $P \equiv C'[\mathit{let}\ x = D\ \mathit{in}\ Q\ \mathit{else}\ R]$ and $\mathsf{fst}(D) \Downarrow_{\Sigma'} M_1$ for some M_1, then $\mathsf{snd}(D) \Downarrow_{\Sigma} M_2$ for some M_2,*

as well as the symmetric properties where we swap fst *and* snd.

The lemmas above are proved in Appendix B.

6 Clause generation

Given a closed biprocess P_0, our protocol verifier builds a set of Horn clauses. This section explains the generation of the clauses, substantially extending to biprocesses previous work at the level of processes.

6.1 Patterns and facts

In the clauses, the terms of processes are represented by patterns, with the following grammar:

$p ::=$	patterns
x, y, z, i	variable
$f(p_1, \ldots, p_n)$	constructor application
$a[p_1, \ldots, p_n]$	name
g	element of $GVar$

We assign a distinct, fresh session identifier variable i to each replication of P_0. (We will use a distinct value for i for each copy of the replicated process.) We assign a pattern $a[p_1, \ldots, p_n]$ to each name a of P_0. We treat a as a function symbol, and write $a[p_1, \ldots, p_n]$ rather than $a(p_1, \ldots, p_n)$ only for clarity. We sometimes write a for $a[\,]$. If a is a free name, then its pattern is $a[\,]$. If a is bound by a restriction (νa) in P_0, then its pattern takes as arguments the terms received as inputs, the results of term evaluations, and the session identifiers of replications in the context that encloses the restriction. For example, in the process $!c'(x).(\nu a)P$, each name created by (νa) is represented by $a[i, x]$ where i is the session identifier for the replication and x is the message received as input in $c'(x)$. We assume that each restriction (νa) in P_0 has a different name a, distinct from any free name of P_0. Moreover, session identifiers enable us to distinguish names created in different copies of processes. Hence, each name created in the process calculus is represented by a different pattern in the verifier.

Patterns include an infinite set of constants $GVar$. These constants are basically universally quantified variables, and occur only in arguments of the predicate nounif, defined in Definition 5 below. We write $GVar(M)$ for the term obtained from M by replacing the variables of M with new constants in the set $GVar$.

Clauses are built from the following predicates:

$F ::=$	facts
$\mathrm{att}'(p, p')$	attacker knowledge
$\mathrm{msg}'(p_1, p_2, p'_1, p'_2)$	output message p_2 on p_1 (resp. p'_2 on p'_1)
$\mathrm{input}'(p, p')$	input on p (resp. p')
$\mathrm{nounif}(p, p')$	impossible unification
bad	bad

Informally, $\mathrm{att}'(p, p')$ means that the attacker may obtain p in $\mathsf{fst}(P)$ and p' in $\mathsf{snd}(P)$ by the same operations; $\mathrm{msg}'(p_1, p_2, p'_1, p'_2)$ means that message p_2 may appear on channel p_1 in $\mathsf{fst}(P)$ and that message p'_2 may appear on channel p'_1 in $\mathsf{snd}(P)$ after the same reductions; $\mathrm{input}'(p, p')$ means that an input may be executed on channel p in $\mathsf{fst}(P)$ and on channel p' in $\mathsf{snd}(P)$, thus enabling the attacker to infer whether p (resp. p') is equal to another channel used for output; $\mathrm{nounif}(p, p')$ means that p and p' cannot be unified modulo Σ by substituting elements of $GVar$ with patterns; finally, bad serves in detecting violations of observational equivalence: when bad is not derivable, we have observational equivalence.

An evident difference with respect to previous translations from processes to clauses is that predicates have twice as many arguments: we use the binary predicate att' instead of the unary one att and the 4-ary predicate msg' instead of the binary one msg. This extension allows us to represent information for both variants of a biprocess.

The predicate nounif is not defined by clauses, but by special simplification steps in the solver, defined in Section 7.

Definition 5 Let p and p' be closed patterns. The fact $\mathrm{nounif}(p, p')$ holds if and only if there is no closed substitution σ with domain $GVar$ such that $\Sigma \vdash \sigma p = \sigma p'$.

6.2 Clauses for the attacker

The following clauses represent the capabilities of the attacker:

$$\text{For each } a \in \mathit{fn}(P_0),\ \mathrm{att}'(a[\,], a[\,]) \tag{Rinit}$$

$$\text{For some } b \text{ that does not occur in } P_0,\ \mathrm{att}'(b[x], b[x]) \tag{Rn}$$

$$\begin{aligned}&\text{For each function } h, \text{ for each pair of rewrite rules}\\ &\quad h(M_1, \dots, M_n) \to M \text{ and } h(M'_1, \dots, M'_n) \to M' \text{ in } \mathrm{def}_{\Sigma'}(h)\\ &\qquad \text{(after renaming of variables)},\\ &\qquad\quad \mathrm{att}'(M_1, M'_1) \wedge \dots \wedge \mathrm{att}'(M_n, M'_n) \to \mathrm{att}'(M, M')\end{aligned} \tag{Rf}$$

$$\mathrm{msg}'(x, y, x', y') \wedge \mathrm{att}'(x, x') \to \mathrm{att}'(y, y') \tag{Rl}$$

$$\mathrm{att}'(x, x') \wedge \mathrm{att}'(y, y') \to \mathrm{msg}'(x, y, x', y') \tag{Rs}$$

$$\mathrm{att}'(x, x') \to \mathrm{input}'(x, x') \tag{Ri}$$

$$\mathrm{input}'(x, x') \wedge \mathrm{msg}'(x, z, y', z') \wedge \mathrm{nounif}(x', y') \to \mathrm{bad} \tag{Rcom}$$

$$\begin{aligned}&\text{For each destructor } g,\\ &\quad \text{for each rewrite rule } g(M_1, \dots, M_n) \to M \text{ in } \mathrm{def}_{\Sigma'}(g),\\ &\qquad \bigwedge_{g(M'_1,\dots,M'_n)\to M' \text{ in } \mathrm{def}_{\Sigma'}(g)} \mathrm{nounif}((x'_1, \dots, x'_n), GVar((M'_1, \dots, M'_n)))\\ &\qquad \wedge\, \mathrm{att}'(M_1, x'_1) \wedge \dots \wedge \mathrm{att}'(M_n, x'_n) \to \mathrm{bad}\end{aligned} \tag{Rt}$$

plus symmetric clauses (Rcom′) and (Rt′) obtained from (Rcom) and (Rt) by swapping the first and second arguments of input' and att' and the first and third arguments of msg'.

Clause (Ri) means that, if the attacker has x (resp. x'), then it can attempt an input on x (resp. x'), thereby testing whether it is equal to some other channel used for output. Clauses (Rcom) and (Rcom′) detect when a communication can occur in one variant of the biprocess and not in the other: the input and output channels are equal in one variant and different in the other. These clauses check that condition 1 of Lemma 2 and its symmetric are true.

Clause (Rt) checks that for all applications of a destructor g, if this application succeeds in $\mathsf{fst}(P)$, then it succeeds in $\mathsf{snd}(P)$, possibly using another rule. Clause (Rt′) checks the converse. These two clauses are essential for obtaining condition 2 of Lemma 2. Consider, for instance, the destructor equals of Section 2.2. After a minor simplification, Clauses (Rt) and (Rt′) become

$$\mathrm{att}'(x, y) \wedge \mathrm{att}'(x, y') \wedge \mathrm{nounif}(y, y') \to \mathrm{bad} \tag{1}$$

$$\mathrm{att}'(y, x) \wedge \mathrm{att}'(y', x) \wedge \mathrm{nounif}(y, y') \to \mathrm{bad} \tag{2}$$

The other clauses are adapted from previous work [3, 16] by replacing unary (resp. binary) predicates with binary (resp. 4-ary) ones. Clause (Rinit) indicates that the attacker initially has all free names of P_0. Clause (Rn) means that the attacker can generate fresh names $b[x]$. Clause (Rf) mean that the attacker can apply all functions to all terms it has. In

this clause, the rewrite rules $h(M_1, \dots, M_n) \rightarrow M$ and $h(M'_1, \dots, M'_n) \rightarrow M'$ may be different elements of $\mathrm{def}_{\Sigma'}(h)$; their variables are renamed so that $M_1, \dots, M_n, M$ on the one hand and $M'_1, \dots, M'_n, M'$ on the other hand do not share variables. Clause (Rl) means that the attacker can listen on all the channels it has, and (Rs) that it can send all the messages it has on all the channels it has.

6.3 Clauses for the protocol

The translation $[\![P]\!]\rho s s' H$ of a biprocess P is a set of clauses, where ρ is an environment that associates a pair of patterns with each name and variable, s and s' are sequences of patterns, and H is a sequence of facts. The empty sequence is written $\emptyset$; the concatenation of a pattern p to the sequence s is written s, p; the concatenation of a fact F to the sequence H is written $H \wedge F$. When ρ associates a pair of patterns with each name and variable, and f is a constructor, we extend ρ as a substitution by $\rho(f(M_1, \dots, M_n)) = (f(p_1, \dots, p_n), f(p'_1, \dots, p'_n))$ where $\rho(M_i) = (p_i, p'_i)$ for all $i \in \{1, \dots, n\}$. We denote by $\rho(M)_1$ and $\rho(M)_2$ the components of the pair $\rho(M)$. We let $\rho(\mathsf{diff}[M, M']) = (\rho(M)_1, \rho(M')_2)$. We define $[\![P]\!]\rho s s' H$ as follows:

$$[\![0]\!]\rho s s' H = \emptyset$$

$$[\![!P]\!]\rho s s' H = [\![P]\!]\rho(s, i)(s', i)H$$

where i is a fresh variable

$$[\![P \mid Q]\!]\rho s s' H = [\![P]\!]\rho s s' H \cup [\![Q]\!]\rho s s' H$$

$$[\![(\nu a)P]\!]\rho s s' H = [\![P]\!](\rho[a \mapsto (a[s], a[s'])])s s' H$$

$$[\![M(x).P]\!]\rho s s' H = [\![P]\!](\rho[x \mapsto (x', x'')])(s, x')(s', x'')(H \wedge \mathrm{msg}'(\rho(M)_1, x', \rho(M)_2, x'')) \cup \{H \rightarrow \mathrm{input}'(\rho(M)_1, \rho(M)_2)\}$$

where x' and x'' are fresh variables

$$[\![\overline{M}\langle N\rangle.P]\!]\rho s s' H = [\![P]\!]\rho s s' H \cup \{H \rightarrow \mathrm{msg}'(\rho(M)_1, \rho(N)_1, \rho(M)_2, \rho(N)_2)\}$$

$$[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho s s' H =$$

$$\bigcup\{[\![P]\!]((\sigma\rho)[x \mapsto (p, p')])(\sigma s, p)(\sigma s', p')(\sigma H) \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p, p'), \sigma)\}$$

$$\cup\ [\![Q]\!]\rho s s'(H \wedge \rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \rho(\mathrm{fails}(\mathsf{snd}(D)))_2)$$

$$\cup\ \{\sigma H \wedge \sigma\rho(\mathrm{fails}(\mathsf{snd}(D)))_2 \rightarrow \mathrm{bad} \mid \rho(D)_1 \Downarrow' (p, \sigma)\}$$

$$\cup\ \{\sigma H \wedge \sigma\rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \rightarrow \mathrm{bad} \mid \rho(D)_2 \Downarrow' (p', \sigma)\}$$

where $\mathrm{fails}(D) = \bigwedge_{\sigma \mid D \Downarrow'(p,\sigma)} \mathrm{nounif}(D, \mathit{GVar}(\sigma D))$

In the translation, the environment ρ maps names and variables to their corresponding pair of patterns—one pattern for each variant of the biprocess. The sequences s and s' contain all input messages, session identifiers, and results of term evaluations in the enclosing context—one sequence for each variant of the biprocess. They are used in the restriction case $(\nu a)P$, to build patterns $a[s]$ and $a[s']$ that correspond to the name a. The sequence H contains all facts that must be true to run the current process.

The clauses generated are similar to those of [16], but clauses are added to indicate which tests the adversary can perform, and predicates have twice as many arguments.

- Replication creates a new session identifier i, added to s and s'. Replication is otherwise ignored, since Horn clauses can be applied any number of times anyway.

- In the translation of an input, the sequence H is extended with the input in question and the environment ρ with a binding of x to a new variable x' in variant 1, x'' in variant 2. Moreover, a new clause $H \rightarrow \mathrm{input}'(\rho(M)_1, \rho(M)_2)$ is added, indicating that when all conditions in H are true, an input on channel M may be executed. This input may enable

the adversary to infer that M is equal to some channel used for output; Clauses (Rcom) or (Rcom′) derive bad when this information may break equivalence.

- The output case adds a clause stating that message N may be sent on channel M.
- Finally, the clauses for a term evaluation are the union of clauses for the cases where the term evaluation succeeds on both sides (then we execute P), where the term evaluation fails on both sides (then we execute Q), and where the term evaluation fails on one side and succeeds on the other (then we derive bad). Indeed, in the last case, the adversary may get to know whether the term evaluation succeeds or fails (when the code executed in the success case is visibly different from the code executed in the failure case).

Example 7 The biprocess of Example 3 yields the clauses:

$$\begin{aligned}&\mathrm{msg}'(c, pk(s), c, pk(s))\\&\mathrm{msg}'(c', x, c', x') \rightarrow \mathrm{msg}'(c, penc(x, pk(s), a[i, x]), c, a[i, x'])\end{aligned}$$

The first clause corresponds to the output of the public key $pk(s)$. The second clause corresponds to the other output: if a message x (resp. x') is received on channel c', then the message $penc(x, pk(s), a[i, x])$ in the first variant (resp. $a[i, x']$ in the second variant) is sent on channel c. The encoding of the fresh name a as a pattern $a[i, x]$ is explained in Section 6.1. ☐

Example 8 The process $c(x).let\ y = \mathrm{eval}\ dec(x, a)\ in\ \overline{c}\langle y\rangle$, where dec is a destructor defined by $dec(enc(x, y), y) \rightarrow x$, yields the clauses:

$$\begin{aligned}&\mathrm{msg}'(c, enc(y, a), c, x') \wedge \mathrm{nounif}(x', enc(\mathsf{g}, a)) \rightarrow \mathrm{bad}\\&\mathrm{msg}'(c, x, c, enc(y', a)) \wedge \mathrm{nounif}(x, enc(\mathsf{g}, a)) \rightarrow \mathrm{bad}\\&\mathrm{msg}'(c, enc(y, a), c, enc(y', a)) \rightarrow \mathrm{msg}'(c, y, c, y')\end{aligned}$$

In the first clause, a message received on c is of the form $enc(y, a)$ in the first variant but not in the second variant; decryption succeeds only in the first variant, so the process is not uniform and we derive bad. The second clause is the symmetric case. In the third clause, decryption succeeds in both variants, and yields an output on channel c. ☐

6.4 Proving equivalences

Let $\rho_0 = \{a \mapsto (a[\,], a[\,]) \mid a \in fn(P_0)\}$. We define the set of clauses that corresponds to biprocess P_0 as:

$$\mathcal{R}_{P_0} = [\![\mathrm{unevaluated}(P_0)]\!]\rho_0\emptyset\emptyset\emptyset \cup \{(\mathrm{Rinit}), (\mathrm{Rn}), \ldots, (\mathrm{Rt}), (\mathrm{Rt}')\}$$

The following result is proved in Appendix D. It shows the soundness of the translation.

Theorem 3 *If* bad *is not a logical consequence of* $\mathcal{R}_{P_0}$*, then* P_0 *satisfies observational equivalence.*

When bad is a logical consequence of $\mathcal{R}_{P_0}$, the derivation of bad from $\mathcal{R}_{P_0}$ can serve for reconstructing a violation of the hypothesis of Corollary 1, via an extension of recent techniques for secrecy analysis [10]. However, the translation of protocols to Horn clauses performs safe abstractions that sometimes result in false counterexamples: the Horn clauses can be applied any number of times, so the translation ignores the number of repetitions of actions. For instance, $(\nu c)(\overline{c}\langle M\rangle \mid c(x).c(x).P)$ satisfies equivalence for any P because P is never executed, and $(\nu c)(\overline{c}\langle \mathsf{diff}[M_1, M_2]\rangle \mid c(x).\overline{d}\langle c\rangle)$ satisfies equivalence for any M_1 and M_2 because its diff

disappears before the attacker obtains channel c. Our technique cannot prove these equivalences in general. The latter example illustrates that our technique typically fails for biprocesses that first keep some value secret and later reveal it. The reason for the failures on $(\nu c)(\overline{c}\langle M\rangle \mid c(x).c(x).P)$ and $(\nu c)(\overline{c}\langle \mathsf{diff}[M_1, M_2]\rangle \mid c(x).\overline{d}\langle c\rangle)$ is that the translation to classical Horn clauses basically treats these two biprocesses like variants with additional replications, namely $(\nu c)(!\overline{c}\langle M\rangle \mid c(x).c(x).P)$ and $(\nu c)(!\overline{c}\langle \mathsf{diff}[M_1, M_2]\rangle \mid !c(x).\overline{d}\langle c\rangle)$ respectively, and these variants do not necessarily satisfy equivalence. On the other hand, the safe abstractions that the translation performs are crucial for the applicability of our technique to infinite-state systems, which is illustrated by many of the examples in this paper.

We also have the following lemma, which is important for proving the soundness of some simplification steps in the solving algorithm below, enabling us to work with terms in normal form only. It is proved in Appendix C.

Lemma 3 *If* bad *is derivable from* $\mathcal{R}_{P_0}$ *then* bad *is derivable from* $\mathcal{R}_{P_0}$ *by a derivation such that* $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{F})$ *where* $\mathcal{F}$ *is the set of intermediately derived facts in this derivation, excluding* nounif *facts.*

7 Solving algorithm

In order to determine whether bad is a logical consequence of $\mathcal{R}_{P_0}$, we use an algorithm based on resolution with free selection, adapting a previous algorithm [17].

7.1 The basic resolution algorithm

The algorithm infers new clauses by resolution, as follows. From two clauses $R = H \rightarrow C$ and $R' = F \wedge H' \rightarrow C'$ (where F is any hypothesis of R') such that C and F are unifiable, with most general unifier σ, it infers $R \circ_F R' = \sigma H \wedge \sigma H' \rightarrow \sigma C'$:

$$\frac{H \rightarrow C \qquad F \wedge H' \rightarrow C'}{\sigma H \wedge \sigma H' \rightarrow \sigma C'}$$

The clause $R \circ_F R'$ is the combination of R and R', in which R proves the hypothesis F of R'. Resolution is guided by a selection function: $\mathsf{sel}(R)$ returns a subset of the hypotheses of R, and the resolution step above applies only when $\mathsf{sel}(R) = \emptyset$ and $F \in \mathsf{sel}(R')$. When $\mathsf{sel}(R) = \emptyset$, we say that the conclusion of R is selected. In this paper, we use the following selection rules:

- $\mathrm{nounif}(p_1, p_2)$ is never selected. (It is handled by special simplification steps.)
- bad is never selected, except in the clause bad, and in clauses whose hypotheses are all of the form $\mathrm{nounif}(p_1, p_2)$. (If we select bad in a clause $H \rightarrow \mathrm{bad}$, then the algorithm will fail to prove that bad is not derivable. That is why we avoid selecting bad when possible.)
- $\mathrm{att}'(x, x')$ with any variables x, x' is selected only when no other fact can be selected. (Our intent is to obtain termination, whereas facts $\mathrm{att}'(x, x')$ can be unified with all facts $\mathrm{att}'(p, p')$ to generate many additional clauses.) In this case, $\mathrm{att}'(x, x')$ is selected preferably when x (or x') occurs in a fact $\mathrm{nounif}(x, p')$ where p' is not a variable. (When we select $\mathrm{att}'(x, x')$, this fact will be unified with some other fact, thus hopefully instantiating x, so that we make progress determining whether $\mathrm{nounif}(x, p')$ is true or not.)

7.2 General simplifications

As part of the algorithm, we apply a series of simplification functions on clauses. Some of them are standard, such as the elimination of tautologies (performed by *elimtaut*) and duplicate hypotheses (performed by *elimdup*). We omit their definitions. Others are specific to our purpose:

- Elimination of $\text{att}'(x, y)$: *elimattx* removes hypotheses $\text{att}'(x, y)$ when x and y do not appear elsewhere in the clause, except possibly in nounif facts. The variables x and y may be the same variable.

- Elimination of useless variables: *elimvar* transforms clauses of the form

$$R = \text{att}'(x, y) \wedge \text{att}'(x, y') \wedge H \rightarrow C$$

into $R\{y/y'\}$, when R is not Clause (1).

The soundness of *elimvar* can be established by cases. If we can derive facts $\text{att}'(p_x, p_y)$ and $\text{att}'(p_x, p_{y'})$ such that $\Sigma \vdash p_y \neq p_{y'}$ from the other clauses, then we can derive bad by applying Clause (1), included in the clause base as Clause (Rt) for $g = \text{equals}$. Otherwise, in any derivation of bad obtained by Lemma 3, any application of R uses the same fact to match both $\text{att}'(x, y)$ and $\text{att}'(x, y')$, and the transformed clause also applies. (The clause R uses $\text{att}'(p_x, p_y)$ and $\text{att}'(p_x, p_{y'})$ with $\Sigma \vdash p_y = p_{y'}$, hence $p_y = p_{y'}$ by the conclusion of Lemma 3.)

The function *elimvar* also performs the symmetric simplification, relying on the presence of Clause (2).

- Elimination of useless forms modulo equality: *simpeq* removes clauses that contain a fact F that is not a nounif fact and is not in normal form relatively to $\mathcal{S}$. The soundness of this simplification follows from Lemma 3. A typical example concerns decryption, when it is defined by an equation (as in Example 5): we can remove any clause that contains $dec(enc(y, x), x)$.

This simplification could be extended to clauses that contain several syntactically different forms of the same term modulo the equational theory, although that would be more difficult to implement.

7.3 Simplifications for nounif

These simplifications are adapted from those for testunif (from [17]).

- Unification: *unify* transforms clauses of the form $H \wedge \text{nounif}(p_1, p_2) \rightarrow C$ as follows. For every $\text{nounif}(p_1, p_2)$ hypothesis in turn, it tries to unify p_1 and p_2 modulo the equational theory, considering elements of $GVar$ as variables. If this unification fails, then the clause becomes $H \rightarrow C$, because $\text{nounif}(p_1, p_2)$ holds when $\Sigma \vdash \sigma p_1 \neq \sigma p_2$ for all σ. Otherwise, *unify* replaces the clause with

$$H \wedge \bigwedge_{j=1}^{n} \text{nounif}((x_1^j, \ldots, x_{k_j}^j), (\sigma_j x_1^j, \ldots, \sigma_j x_{k_j}^j)) \rightarrow C$$

where $\sigma_1, \ldots, \sigma_n$ are the most general unifiers of p_1 and p_2 modulo the equational theory and $x_1^j, \ldots, x_{k_j}^j$ are all variables affected by σ_j. (These may include elements of $GVar$.) In this unification, σ_j is built so that all variables in its domain and its image are variables of p_1 and p_2, and the variables in its domain do not occur in its image. Note that an instance of $\bigwedge_{j=1}^{n} \text{nounif}((x_1^j, \ldots, x_{k_j}^j), (\sigma_j x_1^j, \ldots, \sigma_j x_{k_j}^j))$ is true if and only if the same instance of $\text{nounif}(p_1, p_2)$ is, because $\sigma p_1 = \sigma p_2$ if and only if there exists $j \in \{1, \ldots, n\}$ such that $\sigma(x_1^j, \ldots, x_{k_j}^j) = \sigma\sigma_j(x_1^j, \ldots, x_{k_j}^j)$, for all σ with domain $GVar \cup Var$ where Var is the set of variables.

In order to compute unification modulo the equational theory of p_1 and p_2, we rewrite both terms according to the rewrite rules for the function symbols that they contain (generating some bindings for variables), then syntactically unify the results. Formally,

the most general unifiers of p_1 and p_2 modulo Σ are the substitutions $\sigma_u\sigma$ such that $\mathrm{addeval}(p_1, p_2) \Downarrow' ((p'_1, p'_2), \sigma)$ and σ_u is the most general unifier of p'_1 and p'_2.

For instance, with an empty equational theory, *unify* transforms the clause

$$H \wedge \mathrm{nounif}((enc(x', y'), z'), (enc(\mathtt{g}, y), \mathtt{g})) \rightarrow C$$

into

$$H \wedge \mathrm{nounif}((x', y', z'), (\mathtt{g}, y, \mathtt{g})) \rightarrow C \tag{3}$$

Assuming the equational theory of Example 6, *unify* transforms the clause

$$H \wedge \mathrm{nounif}(x\hat{\ }y, x'\hat{\ }y') \rightarrow C$$

into

$$H \wedge \mathrm{nounif}((x, y), (x', y')) \wedge \mathrm{nounif}((x, x'), (\mathtt{b}\hat{\ }y', \mathtt{b}\hat{\ }y)) \rightarrow C$$

- Swap: *swap* transforms facts $\mathrm{nounif}((p_1, \dots, p_n), (p'_1, \dots, p'_n))$ in clauses obtained after *unify*. When p_i is a variable and $p'_i \in GVar$, it swaps p_i and p'_i everywhere in the nounif fact. Note that an instance of the new nounif fact is true if and only if the same instance of the old one is, since the unification constraints remain the same.

  For instance, *swap* transforms Clause (3) into

$$H \wedge \mathrm{nounif}((\mathtt{g}, y', z'), (x', y, x')) \rightarrow C \tag{4}$$

- Elimination of elements of $GVar$: *elimGVar* transforms facts $\mathrm{nounif}((p_1, \dots, p_n), (p'_1, \dots, p'_n))$ in clauses obtained after *unify* and *swap*: when $p_i = \mathtt{g} \in GVar$, it eliminates the pair p_i, p'_i from the nounif fact.

  An instance of the new nounif fact is true if and only if the same instance of the old one is, because $\mathtt{g} \in GVar$ cannot occur elsewhere in the nounif fact. (This property comes from the result of *unify* and is preserved by *swap*.)

  For instance, *elimGVar* transforms Clause (4) into

$$H \wedge \mathrm{nounif}((y', z'), (y, x')) \rightarrow C$$

- Detection of failed nounif: *elimnouniffalse* removes clauses that contain the hypothesis $\mathrm{nounif}((), ())$.

7.4 Combining the simplifications

We group all simplifications, as follows:

- We define the simplification function $simplify = elimtaut \circ elimattx \circ elimdup \circ \circ elimnouniffalse \circ repeat(elimGVar \circ swap \circ unify \circ elimvar \circ simpeq)$. The expression $repeat(f)$ means that the application of function f is repeated until a fixpoint is obtained, that is, $f(\mathcal{R}) = \mathcal{R}$. It is enough to repeat the simplification only when *elimvar* has modified the set of clauses. Indeed, no new simplification would be done in the other cases. The repetition never leads to an infinite loop, because the number of variables decreases at each iteration.

- We let $condense(\mathcal{R})$ apply *simplify* to $\mathcal{R}$ and then eliminate subsumed clauses. We say that $H_1 \rightarrow C_1$ subsumes $H_2 \rightarrow C_2$ (and we write $(H_1 \rightarrow C_1) \sqsupseteq (H_2 \rightarrow C_2)$) if and only if there exists a substitution σ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$ (as a multiset inclusion). If $\mathcal{R}$ contains clauses R and R' such that R subsumes R', then R' is removed. (In that case, R can do all derivations that R' can do.)

Finally, we define the algorithm $\mathsf{saturate}(\mathcal{R}_0)$. Starting from $condense(\mathcal{R}_0)$, the algorithm adds clauses inferred by resolution with the selection function sel and condenses the resulting clause set until a fixpoint is reached. When a fixpoint is reached, $\mathsf{saturate}(\mathcal{R}_0)$ is the set of clauses R in the clause set such that $\mathsf{sel}(R) = \emptyset$.

We have the following soundness result:

Theorem 4 *If* $\mathsf{saturate}(\mathcal{R}_{P_0})$ *terminates and its result contains no clause of the form* $H \rightarrow \mathrm{bad}$*, then* bad *is not derivable from* $\mathcal{R}_{P_0}$*.*

This result is proved in Appendix E.

8 Extension to scenarios with several stages

Many protocols can be broken into stages, and their security properties can be formulated in terms of these stages. Typically, for instance, if a protocol discloses a session key after the conclusion of a session, then the secrecy of the data exchanged during the session may be compromised but not its authenticity. In some cases, the disclosure of keys and other keying material is harmless and even useful at certain points in protocol executions (e.g., [2]). In this section we extend our technique to protocols with several successive stages. This extension consists in the following changes:

- The syntax of processes is supplemented with a stage prefix, $t : P$, where t is a nonnegative integer. Intuitively, t represents a global clock, and the process $t : P$ is active only during stage t.

- The semantics of processes (and biprocesses) is extended by adding the rules of Figure 4 to those of Figures 2 and 3. This new semantics is a refinement, since $P \rightarrow Q$ in the simple semantics if and only if $t : P \rightarrow_t t : Q$ in the refined semantics. Conversely, if $P' \rightarrow_t Q'$ for staged processes, then $P \rightarrow Q$ in the simple semantics, where P and Q are obtained from P' and Q' by erasing all stage prefixes.

- Instead of att', msg', and input', the clause generation uses distinct predicates att'_t, msg'_t, and input'_t for each stage t used in the protocol. The clauses for the protocol use the predicates indexed by t when translating the process P in $t : P$. The clauses for the attacker are replicated for each att'_t. In addition, new clauses carry over the attacker's knowledge from one stage to the next:

  $$\mathrm{att}'_t(x, x') \rightarrow \mathrm{att}'_{t+1}(x, x') \qquad \text{(Rp)}$$

  As an optimization, when the protocol uses only plain processes for the initial stages $t \leq i$ (that is, diff occurs only at later stages), we translate these processes using the more efficient clause generation of [3], with predicates that keep track of a single process, rather than the two variants of a biprocess.

- Our main theorems hold for staged biprocesses, with minor adaptations and extra optimizations in algorithms. In particular, all definitions and theorems now use $\rightarrow_{_} = \cup_{t \geq 0} \rightarrow_t$ instead of $\rightarrow$.

9 Applications

This section surveys some of the applications of our proof method. The total runtime of all proof scripts for the experiments described below is 45 s on a Pentium M 1.8 GHz. None of these applications could be handled by ProVerif without the extensions presented in this paper.

$$(\nu a)t : P \equiv t : (\nu a)P$$
$$t : (P \mid Q) \equiv t : P \mid t : Q$$
$$t : t' : P \equiv t' : P \quad \text{if } t < t'$$

$$P \rightarrow Q \Rightarrow t : P \rightarrow_t t : Q \qquad \text{(Red Stage)}$$
$$P \rightarrow_t Q \Rightarrow P \mid R \rightarrow_t Q \mid R \qquad \text{(Red Par)}$$
$$P \rightarrow_t Q \Rightarrow (\nu a)P \rightarrow_t (\nu a)Q \qquad \text{(Red Res)}$$
$$P' \equiv P,\ P \rightarrow_t Q,\ Q \equiv Q' \Rightarrow P' \rightarrow_t Q' \qquad \text{(Red} \equiv\text{)}$$

Figure 4: Semantics for stages

9.1 Weak secrets

A *weak secret* represents a secret value with low entropy, such as a human-memorizable password. Protocols that rely on weak secrets are often subject to guessing attacks, whereby an attacker guesses a weak secret, perhaps using a dictionary, and verifies its guess. The guess verification may rely on interaction with protocol participants or on computations on intercepted messages (e.g., [13, 35, 36]). With some care in protocol design, however, those attacks can be prevented:

- On-line guessing attacks can be mitigated by limiting the number of retries that participants allow. An attacker that repeatedly attempts to guess the weak secret should be eventually detected and stopped if it tries to verify its guesses by interacting with other participants.

- Off-line guessing attacks can be prevented by making sure that, even if the attacker (systematically) guesses the weak secret, it cannot verify whether its guess is correct by computing on intercepted traffic.

Off-line guessing attacks can be explained and modelled in terms of a 2-stage scenario. In stage 0, on-line attacks are possible, but the weak secret is otherwise unguessable. In stage 1, the attacker obtains a possible value for the weak secret (intuitively, by guessing it). The absence of off-line attacks is characterized by an equivalence: the attacker cannot distinguish the weak secret used in stage 0 from an unrelated fresh value.

In our calculus, we arrive at the following definition:

Definition 6 (Weak secrecy) Let P be a closed process with no stage prefix. We say that P prevents off-line attacks against w when $(\nu w)(0 : P \mid 1 : (\nu w')\bar{c}\langle \mathsf{diff}[w, w']\rangle)$ satisfies observational equivalence.

This definition is in line with the work of Cohen, Corin et al., Delaune and Jacquemard, Drielsma et al., and Lowe [26–28, 30, 32, 41]. Lowe uses the model-checker FDR to handle a bounded number of sessions, while Delaune and Jacquemard give a decision procedure in this case. Corin et al. give a definition based on equivalence like ours, but do not consider the first, active stage; they analyze only one session.

As a first example, assume that a principal attempts to prove knowledge of a shared password w to a trusted server by sending a hash of this password encrypted under the server's public key. (For simplicity, the protocol does not aim to provide freshness guarantees, so anyone may replay this proof.) Omitting the code for the server, a first protocol may be written:

$$P = (\nu s)\bar{c}\langle pk(s)\rangle.\bar{c}\langle penc(h(w), pk(s))\rangle$$

The first output reveals the public key of the server; the second output communicates the proof of knowledge of w. This protocol does not prevent off-line attacks against w. ProVerif finds an

attack that corresponds to the following adversary:

$$A = 0 : c(pk).c(e).\\ \quad 1 : c(w).\mathit{if}\ e = \mathit{penc}(h(w), pk)\ \mathit{then}\ \overline{\text{Guessed}}\langle\rangle$$

A corrected protocol uses non-deterministic encryption (see Example 3):

$$P = (\nu s, a)\overline{c}\langle pk(s)\rangle.\overline{c}\langle \mathit{penc}(h(w), pk(s), a)\rangle$$

ProVerif automatically produces a proof for this corrected protocol.

As a second example, we consider a simplified version of EKE [13]:

$$\begin{aligned} P_A &= (\nu d_A)\overline{c}\langle \mathit{enc}(\mathtt{b}\hat{\ } d_A, w)\rangle \\ P_B &= c(x).(\nu d_B)\mathit{let}\ k = \mathit{dec}(x, w)\hat{\ } d_B\ \mathit{in}\ \overline{c}\langle \mathit{enc}(\mathtt{b}\hat{\ } d_B, w), k\rangle \\ P &= !P_A \mid !P_B \end{aligned}$$

Here, two parties obtain a shared session key $k = (\mathtt{b}\hat{\ } d_A)\hat{\ } d_B$ via a Diffie-Hellman exchange, in which $\mathtt{b}\hat{\ } d_A$ and $\mathtt{b}\hat{\ } d_B$ are exchanged protected by a weak secret w. The EKE protocol has several rounds of key confirmation; here, instead, we immediately give the session key k to the attacker. Still, relying on the contextual property of equivalence, we can define a context that performs these key confirmations. Since that context does not use the weak secret, the resulting protocol prevents off-line attacks against w as long as the original protocol does.

We have proved security properties of several versions of EKE: the public-key and the Diffie-Hellman versions for EKE [13], and the version with hashed passwords and the one with signatures for Augmented EKE [14]. Unlike the protocol displayed above, our models include an unbounded number of possibly dishonest principals that run parallel sessions.

For the analysis of such protocols, we define encryption under a weak secret by the equational theory of Example 5. The use of this equational theory is important, as it entails that the adversary cannot check whether a decryption is successful and thereby check a guess. In contrast, a straightforward presentation with constructors and destructors but without the equational theory (see Section 2.1) would not be adequate in this respect: with that presentation, an attacker could verify a guess w' of w by testing whether the decryption of the first message of the protocol with w' succeeds.

9.2 Authenticity

Abadi and Gordon [8] use equivalences for expressing authenticity properties, and treat a variant of the Wide-Mouth-Frog protocol as an example. In this protocol, two participants A and B share secret keys k_{AS} and k_{SB} with a server S, respectively. Participant A generates a key k_{AB}, sends it encrypted to S, which forwards it reencrypted to B. Then A sends the payload x to B encrypted under k_{AB}. Finally, B forwards the payload that it receives, possibly for further processing. Essentially, authenticity is defined as an equivalence between the protocol and a specification. The specification is an idealized variant of the protocol, obtained by modifying B so that, independently of what it receives, it forwards A's payload x.

For the one-session version [8, Section 3.2.2], the protocol and the specification can be combined into the following biprocess P_0:

$$\begin{aligned} &P_A = (\nu k_{AB})\overline{c}\langle \mathit{enc}(k_{AB}, k_{AS})\rangle.\overline{c}\langle \mathit{enc}(x, k_{AB})\rangle \\ &P_S = c(y_1).\mathit{let}\ y_2 = \mathit{dec}(y_1, k_{AS})\ \mathit{in}\ \overline{c}\langle \mathit{enc}(y_2, k_{SB})\rangle \\ &P_B = c(y_3).\mathit{let}\ y_4 = \mathit{dec}(y_3, k_{SB})\ \mathit{in}\ c(y_5).\mathit{let}\ x' = \mathit{dec}(y_5, y_4)\ \mathit{in}\ \overline{e}\langle \mathsf{diff}[x, x']\rangle \\ &P_0 = c(x).(\nu k_{AS})(\nu k_{SB})(P_A \mid P_S \mid P_B) \end{aligned}$$

with the rewrite rule $\mathit{dec}(\mathit{enc}(x, y), y) \rightarrow x$ for the destructor dec.

The technique presented in this paper automatically proves that P_0 satisfies observational equivalence, and hence establishes the desired authenticity property. Thus, it eliminates the need for a laborious manual proof. The technique can also be used for simplifying the proof of authenticity for the multi-session version.

Authenticity properties are sometimes formulated as correspondence assertions on behaviors, rather than as equivalences. Previous work shows how to check those assertions with ProVerif [16]. However, that previous work does not apply to equivalences.

9.3 Complete sessions in JFK

Finally, we show other ways in which automated proofs of equivalences can contribute to protocol analyses, specifically studying JFK, a modern session-establishment protocol for IP security [9].

In recent work [4], we modelled JFK in the applied pi calculus. We used processes for representing the reachable states of JFK, for any number of principals and sessions, and stated security properties as equivalences. Although we relied on ProVerif for reasoning about behaviors, our main proofs of equivalences were manual. Applying the techniques of this paper, we can revise and largely automate those proofs. The resulting proofs rely on equivalences on biprocesses, verified by ProVerif, composed with standard pi calculus equivalences that do not depend on the signature for terms.

In particular, a core property of JFK is that, once a session completes, its session key is (apparently) unrelated to the cryptographic material exchanged during the session, and all those values can be replaced by distinct fresh names [4, Theorem 2]. This property can be stated and proved in terms of a biprocess S that outputs either the actual results of JFK computations (in $\mathsf{fst}(S)$) or distinct fresh names (in $\mathsf{snd}(S)$), in parallel with the rest of the JFK system to account for any other sessions. The proof of this property goes as follows. The JFK system is split into $S \approx C[S']$, where S' is similar to S but omits unimportant parts of JFK, collected in the evaluation context $C[_]$. The proof that $S \approx C[S']$ is straightforward; it relies on pi calculus equivalences that eliminate communications on private channels introduced in the split. ProVerif shows that S' satisfies equivalence. Using the contextual property of equivalence, $C[S']$ satisfies equivalence, hence $\mathsf{fst}(S) \approx \mathsf{snd}(S)$.

10 Conclusion

In the last decade, there has been substantial research on proof methods for security protocols. While many of those proof methods have focused on predicates on behaviors, others have addressed equivalences between systems (e.g., [1, 6–8, 23–25, 29, 33, 34, 38, 39, 46]). Much of this research is concerned with obtaining sound and complete proof systems, often via sophisticated bisimulations, and eventually decision algorithms for restricted cases. In our opinion, these are important goals, and the results to date are significant.

In the present paper, we aim to contribute to this body of research with a different approach. We do not emphasize the development of bisimulation techniques. Rather, we leverage behavior-oriented techniques and tools (ProVerif, in particular) for equivalence proofs. We show how to derive equivalences by reasoning about behaviors—specifically, by reasoning about behaviors of applied pi calculus biprocesses. We also show how to translate those biprocesses to Horn clauses and how to reason about their behaviors by resolution. The resulting proof method is sound, although that is not simple to establish. We demonstrate the usefulness of the method through automated analyses of interesting, infinite-state systems.

Acknowledgments Bruno Blanchet's work was partly done at Max-Planck-Institut für Informatik, Saarbrücken. Martín Abadi's work was partly supported by the National Science

Foundation under Grants CCR-0204162, CCR-0208800, and CCF-0524078. We are grateful to Harald Ganzinger for helpful discussions on the treatment of equational theories.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, Sept. 1999.

[2] M. Abadi, A. Birrell, M. Burrows, F. Dabek, and T. Wobber. Bankable postage for network services. In V. Saraswat, editor, *Advances in Computing Science – ASIAN 2003, Programming Languages and Distributed Computation, 8th Asian Computing Science Conference*, volume 2896 of *Lecture Notes on Computer Science*, pages 72–90, Mumbai, India, Dec. 2003. Springer.

[3] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[4] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*. To appear. An extended abstract appears in Programming Languages and Systems, 13th European Symposium on Programming (ESOP 2004).

[5] M. Abadi and V. Cortier. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science*, 367(1–2):2–32, Nov. 2006.

[6] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, London, United Kingdom, Jan. 2001. ACM Press.

[7] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5(4):267–303, Winter 1998.

[8] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, Jan. 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[9] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, K. Keromytis, and O. Reingold. Just Fast Keying: Key agreement in a hostile Internet. *ACM Transactions on Information and System Security*, 7(2):242–273, May 2004.

[10] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005. IEEE.

[11] F. Baader and C. Tinelli. Deciding the word problem in the union of equational theories. Technical Report UIUCDCS-R-98-2073, UILU-ENG-98-1724, University of Illinois at Urbana-Champaign, Oct. 1998.

[12] M. Baudet. *Sécurité des protocoles cryptographiques: aspects logiques et calculatoires.* PhD thesis, Ecole Normale Supérieure de Cachan, 2007.

[13] S. M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, May 1992.

[14] S. M. Bellovin and M. Merritt. Augmented Encrypted Key Exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 244–250, Nov. 1993.

[15] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[16] B. Blanchet. From secrecy to authenticity in security protocols. In M. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, Sept. 2002. Springer.

[17] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[18] B. Blanchet. Automatic proof of strong secrecy for security protocols. Technical Report MPI-I-2004-NWG1-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany, July 2004.

[19] B. Blanchet. Security protocols: From linear to classical logic by abstract interpretation. *Information Processing Letters*, 95(5):473–479, Sept. 2005.

[20] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 331–340, Chicago, IL, June 2005. IEEE Computer Society.

[21] C. Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, Jan. 2000.

[22] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis for the π-calculus. In *International Conference on Concurrency Theory (Concur'98)*, volume 1466 of *Lecture Notes on Computer Science*, pages 84–98. Springer, Sept. 1998.

[23] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2002.

[24] J. Borgström, S. Briais, and U. Nestmann. Symbolic bisimulation in the spi calculus. In P. Gardner and N. Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes on Computer Science*, pages 161–176. Springer, Aug. 2004.

[25] J. Borgström and U. Nestmann. On bisimulations for the spi calculus. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology: 9th International Conference, AMAST 2002*, volume 2422 of *Lecture Notes on Computer Science*, pages 287–303, Saint-Gilles-les- Bains, Reunion Island, France, Sept. 2002. Springer.

[26] E. Cohen. Proving protocols safe from guessing. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.

[27] R. Corin, J. M. Doumen, and S. Etalle. Analysing password protocol security against off-line dictionary attacks. In *2nd Int. Workshop on Security Issues with Petri Nets and other Computational Models (WISP)*, Electronic Notes in Theoretical Computer Science, June 2004.

[28] R. Corin, S. Malladi, J. Alves-Foss, and S. Etalle. Guess what? here is a new tool that finds some new guessing attacks. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security (WITS'03)*, Warsaw, Poland, Apr. 2003.

[29] V. Cortier. *Vérification automatique des protocoles cryptographiques.* PhD thesis, ENS de Cachan, Mar. 2003.

[30] S. Delaune and F. Jacquemard. A theory of dictionary attacks and its complexity. In *17th IEEE Computer Security Foundations Workshop*, pages 2–15, Pacific Grove, CA, June 2004. IEEE.

[31] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.

[32] P. H. Drielsma, S. Mödersheim, and L. Viganò. A formalization of off-line guessing for security protocol analysis. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning: 11th International Conference, LPAR 2004*, volume 3452 of *Lecture Notes on Computer Science*, pages 363–379, Montevideo, Uruguay, Mar. 2005. Springer.

[33] L. Durante, R. Sisto, and A. Valenzano. Automatic testing equivalence verification of spi calculus specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):222–284, Apr. 2003.

[34] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, Sept. 1997.

[35] L. Gong. Verifiable-text attacks in cryptographic protocols. In *INFOCOM '90, The Conference on Computer Communications*, pages 686–693, San Francisco, CA, June 1990. IEEE.

[36] L. Gong, T. M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, June 1993.

[37] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[38] H. Hüttel. Deciding framed bisimilarity. In *4th International Workshop on Verification of Infinite-State Systems (INFINITY'02)*, pages 1–20, Brno, Czech Republic, Aug. 2002.

[39] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 World Congress On Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes on Computer Science*, pages 776–793, Toulouse, France, Sept. 1999. Springer.

[40] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes on Computer Science*, pages 147–166. Springer, 1996.

[41] G. Lowe. Analyzing protocols subject to guessing attacks. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, Jan. 2002.

[42] D. Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, 2003.

[43] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[44] F. Pottier. A simple view of type-secure information flow in the π-calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, Cape Breton, Nova Scotia, June 2002.

[45] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 319–330, Portland, Oregon, Jan. 2002.

[46] A. Ramanathan, J. Mitchell, A. Scedrov, and V. Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In I. Walukiewicz, editor, *FOSSACS 2004 - Foundations of Software Science and Computation Structures*, volume 2987 of *Lecture Notes on Computer Science*, pages 468–483, Barcelona, Spain, Mar. 2004. Springer.

Appendix

The Appendix contains proofs of the main results of this paper. Proof scripts for all examples and applications, as well as the tool ProVerif, are available at `http://www.di.ens.fr/~blanchet/obsequi/`.

A Proof of Theorem 2

In this section, we prove the correctness of Algorithms 1, 2, and 3 given in Section 5.3. We begin with preliminary lemmas on modelling equational theories.

A.1 Preliminary lemmas

Lemma 4 *Let N be either a name or a variable. If $\Sigma \vdash M = N$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M\})$, then $M = N$. For any set of terms $\mathcal{M}$, if $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$, then $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{N\})$.*

Proof We detail the proof for $N = a$.

If we had $M \neq a$, then either M contains a, so M does not satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M\})$, or M does not contain a. In this latter case, since Σ is invariant by substitution of terms for names, for all M', we have $\Sigma \vdash a\{M'/a\} = M\{M'/a\}$ so $\Sigma \vdash M' = M$, then all terms are equated by Σ, which contradicts the hypothesis.

Let M'' be any subterm of an element of $\mathcal{M}$. We have $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M''\})$, so if $\Sigma \vdash M'' = a$, then $M'' = a$, by the previous property. Moreover, a is irreducible by $\mathcal{S}$. So we have $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{a\})$. □

Lemma 5 *Assume $\mathcal{S} = \emptyset$ and Σ is any equational theory. Then Property S2 is true.*

Proof We show the following property: If $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$, then for any term M there exists M' such that $\Sigma \vdash M' = M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$.

The proof is by induction on M.

- Cases $M = a$ and $M = x$: Let $M' = M$; by Lemma 4, $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$.
- Case $M = f(N_1, \ldots, N_n)$: By induction hypothesis, there exist $N'_1, \ldots, N'_n$ such that $\Sigma \vdash N_i = N'_i$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{N'_1, \ldots, N'_n\})$. (For N_i, we apply the induction hypothesis with $\mathcal{M} \cup \{N'_1, \ldots, N'_{i-1}\}$ instead of $\mathcal{M}$.)

  If there exists a subterm M' of $\mathcal{M} \cup \{N'_1, \ldots, N'_n\}$ such that $\Sigma \vdash f(N_1, \ldots, N_n) = M'$, then we have $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$.

  Otherwise, let $M' = f(N'_1, \ldots, N'_n)$. We have $\Sigma \vdash M' = f(N_1, \ldots, N_n)$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$ since the subterms of $\mathcal{M} \cup \{M'\}$ are the subterms of $\mathcal{M} \cup \{N'_1, \ldots, N'_n\}$ and the

term M', $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M}\cup\{N'_1,\dots,N'_n\})$ and the new subterm M' is different from any subterm of $\mathcal{M}\cup\{N'_1,\dots,N'_n\}$ modulo the equational theory of Σ. □

A.2 Convergent theories

Lemma 6 *The signature Σ' built by Algorithm 1 models Σ.*

Proof Properties S1 and S3 are obvious.

Let us prove Property S2. Assume that $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M})$. Let $M' = M\!\downarrow$. Then $\Sigma \vdash M = M'$ and M' is irreducible by $\mathcal{S}$. Let N_1 and N_2 be two subterms of $\mathcal{M}\cup\{M'\}$ such that $\Sigma \vdash N_1 = N_2$, that is, $N_1\!\downarrow = N_2\!\downarrow$. Moreover, N_1 and N_2 are in normal form relatively to $\mathcal{S}$, so $N_1 = N_2$. Hence $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M}\cup\{M'\})$.

Finally, we prove Property S4. When $M = f(M_1,\dots,M_n)$, we let $M\!\downarrow^s = f(M_1\!\downarrow,\dots,M_n\!\downarrow)$ be the term obtained by reducing to normal form the strict subterms of M. We first note a few elementary properties of the algorithm:

P1. If $N \to N'$ is in $\mathcal{S}$, then there is $N_1 \to N'_1$ in E such that $T[\sigma N_1] = N\!\downarrow^s$ and $T[\sigma N'_1] = N'\!\downarrow$ for some σ and T. (This is true at the beginning of an execution of the algorithm, and remains true during the execution, since a rule $N_1 \to N'_1$ is removed from E only when there is another rule $N_2 \to N'_2$ such that $N_1 = T[\sigma N_2]$ and $N'_1 = T[\sigma N'_2]$ for some σ and T.)

P2. If N is reducible by a rule in E, then it is also reducible by a rule in $\mathcal{S}$. (This is true at the beginning of an execution of the algorithm and remains true during the execution.)

P3. If $N \to N'$ is in E, then N is not a variable, and all variables of N' occur in N. (This is true at the beginning of an execution of the algorithm and remains true during the execution.)

P4. At the end of the algorithm, if $N_1 \to N'_1$ and $N_2 \to N'_2$ in E are such that $N'_1 = T[N''_1]$, N''_1 is not a variable, and σ_u is the most general unifier of N''_1 and N_2, then there exist $N_3 \to N'_3$ in E, T', and σ such that $T'[\sigma N_3] = (\sigma_u N_1)\!\downarrow^s$ and $T'[\sigma N'_3] = \sigma_u T[\sigma_u N'_2]\!\downarrow$. (This simply expresses that the fixpoint is reached: the rule $(\sigma_u N_1)\!\downarrow^s \to \sigma_u T[\sigma_u N'_2]\!\downarrow$ has been added to E.)

We show the following two properties, P5(n) for $n > 0$ and P6(n) for $n \geq 0$:

P5(n). If the longest reduction of M by $\mathcal{S}$ is of length n and $M = M\!\downarrow^s$, then there exist $N \to N'$ in E and σ such that $M = \sigma N$ and $M\!\downarrow = \sigma N'$.

P6(n). If the longest reduction of $\sigma N'_1$ by $\mathcal{S}$ is of length n, $M = \sigma N_1 \to^*_{\mathcal{S}} \sigma N'_1$, $M = M\!\downarrow^s$, and $N_1 \to N'_1$ is in E, then there exist $N_2 \to N'_2$ in E and σ' such that $M = \sigma' N_2 \to^*_{\mathcal{S}} \sigma' N'_2 = M\!\downarrow$.

The proof is by induction on n.

- Proof of P5(n) ($n > 0$).

  Since $M = M\!\downarrow^s$, the strict subterms of M are irreducible, so the first application of a rewrite rule in any reduction of M much touch the root function symbol of M. Let $N \to N'$ be this rewrite rule. There exists σ_1 such that $M = \sigma_1 N$. Since $N \to N'$ is in $\mathcal{S}$, by P1, there is $N_1 \to N'_1$ in E such that $T[\sigma_2 N_1] = N\!\downarrow^s$ and $T[\sigma_2 N'_1] = N'\!\downarrow$ for some σ_2 and T. Since the strict subterms of $\sigma_1 N$ are irreducible by $\mathcal{S}$, the strict subterms of N are also irreducible by $\mathcal{S}$, hence $N\!\downarrow^s = N$. Furthermore, $T = [\,]$, since otherwise a strict subterm of N would be reducible by $N_1 \to N'_1$ in E, so using P2, it would also be

reducible by $\mathcal{S}$. Hence $\sigma_1\sigma_2 N_1 = \sigma_1 N = M$ and $\sigma_1\sigma_2 N'_1 = \sigma_1(N'\downarrow)$. Let $\sigma = \sigma_1\sigma_2$. Then $M = \sigma N_1 \rightarrow^+_{\mathcal{S}} \sigma N'_1$.

By P6(n') where n' is the length of the longest reduction of $\sigma N'_1$ ($n' < n$), there exist $N_2 \rightarrow N'_2$ in E and σ' such that $M = \sigma' N_2 \rightarrow^*_{\mathcal{S}} \sigma' N'_2 = M\downarrow$, which is P5($n$).

- Proof of P6(n), $n = 0$, $\sigma N'_1$ is irreducible by $\mathcal{S}$. Then $\sigma N'_1 = M\downarrow$ and we have P6(0) by taking $\sigma' = \sigma$, $N_2 = N_1$, and $N'_2 = N'_1$.

- Proof of P6(n), $n > 0$, $\sigma N'_1$ is reducible by $\mathcal{S}$.

  Let us consider a minimal subterm of $\sigma N'_1$ which is reducible by $\mathcal{S}$, that is, a subterm of $\sigma N'_1$ reducible by $\mathcal{S}$ but such that all its strict subterms are irreducible by $\mathcal{S}$. Such a term is of the form $\sigma N'_3$, where N'_3 is a non-variable subterm of N'_1. (Indeed, all terms σx and their subterms are irreducible by $\mathcal{S}$, since they are strict subterms of M and $M = M\downarrow^s$.)

  The longest reduction of $\sigma N'_3$ is at most as long as the one of $\sigma N'_1$, so by P5, there exist $N_4 \rightarrow N'_4$ in E and σ'' such that $\sigma N'_3 = \sigma'' N_4$ and $(\sigma N'_3)\downarrow = \sigma'' N'_4$. Thus we have $M = \sigma N_1 \rightarrow^*_{\mathcal{S}} \sigma N'_1 = \sigma T[\sigma N'_3] = \sigma T[\sigma'' N_4] \rightarrow^+_{\mathcal{S}} \sigma T[\sigma'' N'_4] = \sigma T[(\sigma N'_3)\downarrow]$.

  The rewrite rules $N_1 \rightarrow N'_1$ and $N_4 \rightarrow N'_4$ have a critical pair, that is, $N'_1 = T[N'_3]$, N'_3 is not a variable, and N'_3 and N_4 unify, with most general unifier σ_u. By P4, there is $N_5 \rightarrow N'_5$ in E such that $T'[\sigma_5 N_5] = (\sigma_u N_1)\downarrow^s$ and $T'[\sigma_5 N'_5] = \sigma_u T[\sigma_u N'_4]\downarrow$ for some T' and σ_5. Moreover, $\sigma_u N_1$ is more general than σN_1, so the strict subterms of $\sigma_u N_1$ are irreducible, since the strict subterms of σN_1 are. So $(\sigma_u N_1)\downarrow^s = \sigma_u N_1$. Furthermore $T' = [\,]$, since otherwise a strict subterm of $\sigma_u N_1$ would be reducible by E, so using P2, it would also be reducible by $\mathcal{S}$. Hence $\sigma_5 N_5 = \sigma_u N_1$ and $\sigma_5 N'_5 = \sigma_u T[\sigma_u N'_4]\downarrow$.

  Then $M = \sigma_1 N_5 \rightarrow^*_{\mathcal{S}} \sigma_1 N'_5$ for some σ_1. Moreover, $\sigma N'_1 = \sigma T[\sigma'' N_4] \rightarrow^+_{\mathcal{S}} \sigma T[\sigma'' N'_4] \rightarrow^*_{\mathcal{S}} \sigma_1 N'_5$, so the longest reduction of $\sigma_1 N'_5$ is strictly shorter than the longest reduction of $\sigma N'_1$, hence by P6 applied to $\sigma_1 N'_5$, there exist σ' and $N_2 \rightarrow N'_2$ in E such that $M = \sigma' N_2 \rightarrow^*_{\mathcal{S}} \sigma' N'_2 = M\downarrow$. This yields P6 for $\sigma N'_1$.

We now turn to the proof of Property S4 itself. Assume $\Sigma \vdash f(M_1, \ldots, M_n) = M$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M, M_1, \ldots, M_n\})$. We show that there exist $f(N_1, \ldots, N_n) \rightarrow N$ in $\mathrm{def}_{\Sigma'}(f)$ and σ such that $M = \sigma N$ and $M_i = \sigma N_i$. Since M is irreducible, we have $M = f(M_1, \ldots, M_n)\downarrow$.

- If $f(M_1, \ldots, M_n)$ is irreducible by $\mathcal{S}$, then $f(M_1, \ldots, M_n) = M$ and we have the result using the rule $f(x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n)$ always in $\mathrm{def}_{\Sigma'}(f)$.

- Otherwise, we have $f(M_1, \ldots, M_n)\downarrow^s = f(M_1, \ldots, M_n)$ since M_i is irreducible for all $i \in \{1, \ldots, n\}$. Property P5 for the term $f(M_1, \ldots, M_n)$ yields the desired result, since every rule $f(N_1, \ldots, N_n) \rightarrow N'$ of E is in $\mathrm{def}_{\Sigma'}(f)$. □

We say that $\mathcal{S}$ is a convergent subterm system when $\mathcal{S}$ is convergent and all its rewrite rules are of the form $M \rightarrow N$ where N is either a strict subterm of M or a closed term in normal form with respect to $\mathcal{S}$ [5, 12].

Lemma 7 *When $\mathcal{S}$ is a convergent subterm rewriting system, Algorithm 1 terminates and the final value of E is normalize($\mathcal{S}$).*

Proof Let $\mathcal{S}$ be a convergent subterm system, with Σ the associated equational theory. Let E_1 be obtained by replacing each rule $f(M_1, \ldots, M_n) \rightarrow N$ of $\mathcal{S}$ with $f(M_1\downarrow, \ldots, M_n\downarrow) \rightarrow N\downarrow$ and removing rules of the form $M \rightarrow M$. Let $E_2 = \mathit{normalize}(\mathcal{S})$. We first show:

P1. if $M \neq N$, $\Sigma \vdash M = N$, and N and the strict subterms of M are in normal form, then there exist $M_1 \rightarrow N_1$ in E_2 and σ such that $M = \sigma M_1$ and $N = \sigma N_1$.

Since $\Sigma \vdash M = N$, we have $M\!\downarrow = N\!\downarrow$. The term N is in normal form, so $M\!\downarrow = N$, so $M \rightarrow^*_{\mathcal{S}} N$. Since $M \neq N$, $M \rightarrow_{\mathcal{S}} M' \rightarrow^*_{\mathcal{S}} N$. Since the strict subterms of M are in normal form, there are a rewrite rule $M_1 \rightarrow M'_1$ of $\mathcal{S}$ and a substitution σ such that $M = \sigma M_1$ and $M' = \sigma M'_1$. If M'_1 is a strict subterm of M_1, M' is a strict subterm of M, so M' is in normal form, hence $M' = N$. If M'_1 is a closed term in normal form, $M' = M'_1$ is in normal form, so we also have $M' = N$.

Moreover, M'_1 and the strict subterms of M_1 are in normal form since M' and the strict subterms of M are. So the rewrite rule $M_1 \rightarrow N_1$ is preserved by the transformation of $\mathcal{S}$ into E_1, so $M_1 \rightarrow N_1$ is in E_1. Finally, if $M_1 \rightarrow N_1$ is removed when transforming E_1 into E_2, there are another rule $M'_1 \rightarrow N'_1$ in E_2 and a substitution σ' such that $M_1 = \sigma' M'_1$ and $N_1 = \sigma' N'_1$, so Property P1 holds in any case.

Let $M_0 \rightarrow N_0$ be a rewrite rule added by Algorithm 1. We show that $E_2 = \mathit{normalize}(E_2 \cup \{M_0 \rightarrow N_0\})$. Let E_3 be obtained by replacing each rule $f(M_1, \dots, M_n) \rightarrow N$ of $E_2 \cup \{M_0 \rightarrow N_0\}$ with $f(M_1\!\downarrow, \dots, M_n\!\downarrow) \rightarrow N\!\downarrow$ and removing rules of the form $M \rightarrow M$. Since E_2 has already been normalized, when we transform $E_2 \cup \{M_0 \rightarrow N_0\}$ into E_3, only $M_0 \rightarrow N_0$ is transformed, into a rule $M \rightarrow N$. If $M = N$, the rule $M \rightarrow N$ is removed, so we immediately have $E_2 = \mathit{normalize}(E_2 \cup \{M_0 \rightarrow N_0\})$. Otherwise, by Property P1, there exist $M_1 \rightarrow N_1$ in E_2 (so in E_3) and σ such that $M = \sigma M_1$ and $N = \sigma N_1$. Hence $M_0 \rightarrow N_0$ is removed by the last step of *normalize*, so $E_2 = \mathit{normalize}(E_2 \cup \{M_0 \rightarrow N_0\})$. We conclude that the fixpoint is reached before iterating, and it is E_2. □

A.3 Linear theories

Lemma 8 *The signature Σ' built by Algorithm 2 models Σ.*

Proof Property S1 is obvious. Property S2 follows from Lemma 5. Property S3 follows from the invariant that, if $M \rightarrow M'$ is in E, then $\Sigma \vdash M = M'$. Next, we prove Property S4. We first note a few elementary properties of the algorithm:

P1. If $N = N'$ or $N' = N$ is an equation of Σ, then there is $N_1 \rightarrow N'_1$ in E such that $T[\sigma N_1] = N$ and $T[\sigma N'_1] = N'$ for some σ and T. (This is true at the beginning of an execution of the algorithm, and remains true during the execution, since a rule $N_1 \rightarrow N'_1$ is removed from E only when there is another rule $N_2 \rightarrow N'_2$ in E such that $N_1 = T[\sigma N_2]$ and $N'_1 = T[\sigma N'_2]$ for some σ and T.)

P2. At the end of the algorithm, if $N_1 \rightarrow N'_1$ and $N_2 \rightarrow N'_2$ in E are such that $N'_1 = T[N''_1]$, N''_1 and N_2 are not variables, and σ_u is the most general unifier of N''_1 and N_2, then there exist $N_3 \rightarrow N'_3$ in E, T', and σ such that $T'[\sigma N_3] = \sigma_u N_1$ and $T'[\sigma N'_3] = \sigma_u T[\sigma_u N'_2]$. (This simply expresses that the fixpoint is reached: the rule $(\sigma_u N_1) \rightarrow \sigma_u T[\sigma_u N'_2]$ has been added to E.)

    Similarly, if $N_1 \rightarrow N'_1$ and $N_2 \rightarrow N'_2$ in E are such that $N_2 = T[N''_2]$, N'_1 and N''_2 are not variables, and σ_u is the most general unifier of N'_1 and N''_2, then there exist $N_3 \rightarrow N'_3$ in E, T', and σ such that $T'[\sigma N_3] = \sigma_u T[\sigma_u N_1]$ and $T'[\sigma N'_3] = \sigma_u N'_2$.

Let us now prove a few more properties:

P3. For all M, M', if $\Sigma \vdash M = M'$ then $M \rightarrow^*_E M'$.

    Assume that $\Sigma \vdash M = M'$ comes from one equation of Σ. Then there are $N = N'$ in Σ, T, and σ such that $M = T[\sigma N]$ and $M' = T[\sigma N']$. Hence, by P1, there are $N_1 \rightarrow N'_1$ in E, T', and σ' such that $N = T'[\sigma' N_1]$ and $N' = T'[\sigma' N'_1]$. So $M = T[(\sigma T')[\sigma\sigma' N_1]] \rightarrow_E T[(\sigma T')[\sigma\sigma' N'_1]] = M'$. The property stated above follows immediately.

P4. If $M_1 \rightarrow_E M_2 \rightarrow_E M_3$ using two rules $M \rightarrow N$ and $M' \rightarrow N'$ of E such that neither N nor M' are variables, $M_1 = T_1[\sigma_1 M]$, $M_2 = T_1[\sigma_1 N] = T_2[\sigma_2 M']$, and $M_3 = T_2[\sigma_2 N']$ for some contexts T_1 and T_2 and substitutions σ_1 and σ_2, then

- either $M_1 \to_E M_3$ in a single step;
- or the rules commute: $M_1 \to_E M_2' \to_E M_3$ where $M_1 \to_E M_2'$ comes from $M' \to N'$ and $M_2' \to_E M_3$ comes from $M \to N$.

We prove the property by case analysis on T_1 and T_2:

(1) The occurrences of the holes of T_1 and T_2 are not nested: there exists T'' such that $T_1 = T''[[\,], \sigma_2 M']$ and $T_2 = T''[\sigma_1 N, [\,]]$. So $M_1 = T''[\sigma_1 M, \sigma_2 M']$, $M_2 = T''[\sigma_1 N, \sigma_2 M']$, and $M_3 = T''[\sigma_1 N, \sigma_2 N']$. Then the rules commute: $M_1 = T''[\sigma_1 M, \sigma_2 M'] \to_E M_2' = T''[\sigma_1 M, \sigma_2 N'] \to_E M_3 = T''[\sigma_1 N, \sigma_2 N']$.

(2) The occurrence of the hole of T_1 is inside the one of T_2: $T_1 = T_2[T']$. We distinguish two subcases:

(2a) T' is an instance of M': $T' = \sigma_3 M'$. So we have $M_1 = T_2[(\sigma_3 M')[\sigma_1 M]]$, $M_2 = T_2[(\sigma_3 M')[\sigma_1 N]]$, and $M_3 = T_2[(\sigma_3 N')[\sigma_1 N]]$. The linearity of N' guarantees that $\sigma_3 N'$ contains at most one hole, since $\sigma_3 M'$ contains one hole.

If $\sigma_3 N'$ contains no hole (that is, the variable x of M' such that $\sigma_3 x$ contains a hole does not occur in N'), then $M_1 = T_2[(\sigma_3 M')[\sigma_1 M]] \to_E M_3 = T_2[(\sigma_3 N')]$ by $M' \to_E N'$.

If $\sigma_3 N'$ contains exactly one hole, the rules commute: $M_1 = T_2[(\sigma_3 M')[\sigma_1 M]] \to_E M_2' = T_2[(\sigma_3 N')[\sigma_1 M]] \to_E M_3 = T_2[(\sigma_3 N')[\sigma_1 N]]$.

(2b) T' is not an instance of M'. Since $T'[\sigma_1 N] = \sigma_2 M'$ and M' is linear, the hole of T' occurs at a non-variable position in M', so N and M' form a critical pair and, by Property P2, E contains a rule that corresponds to the application of both rewrite rules $M \to N$ and $M' \to N'$.

(3) The occurrence of the hole of T_2 is inside the one of T_1: $T_2 = T_1[T']$. The proof is similar to the one for case (2).

Let $\Sigma \vdash f(M_1, \dots, M_n) = M$ with $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M, M_1, \dots, M_n\})$. We show that there exist $f(N_1, \dots, N_n) \to N$ in $\mathrm{def}_{\Sigma'}(f)$ and σ such that $M = \sigma N$ and $M_i = \sigma N_i$ for all $i \in \{1, \dots, n\}$.

Since $\Sigma \vdash f(M_1, \dots, M_n) = M$, we have $f(M_1, \dots, M_n) \to_E^* M$ by P3. Consider a shortest sequence such that $f(M_1, \dots, M_n) \to_E^* M$.

- In this sequence, consecutive rewrite rules always commute, because otherwise we would obtain a shorter sequence by P4.
- If this sequence uses a rule $x \to M'$ in E, consider the last such rule. It commutes with the rule that immediately follows. So we obtain a sequence in which $x \to M'$ is applied last. This is impossible since $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M\})$. From now on, we consider a sequence that does not use any rewrite rule of the form $x \to M'$.
- If this sequence uses no rewrite rule applied with empty context, then $M = f(M_1', \dots, M_n')$ and $M_i \to_E^* M_i'$, so $\Sigma \vdash M_i = M_i'$. Since $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M_1, \dots, M_n, M\})$, $M_i = M_i'$, so $M = f(M_1, \dots, M_n)$. Then $f(x_1, \dots, x_n) \to f(x_1, \dots, x_n)$ in $\mathrm{def}_{\Sigma'}(f)$ and $\sigma x_i = M_i$ yields the desired result.
- If this sequence uses at least one rewrite rule applied with empty context, let $f(N_1, \dots, N_n) \to N$ be the first such rule.

  If the sequence uses a rule $M' \to x$ in E before $f(N_1, \dots, N_n) \to N$, then this rule is applied with non-empty context (because otherwise $f(N_1, \dots, N_n) \to N$ would not be the first rule with empty context). Consider the first such rule. This rule commutes with the rule just before it. Moreover, after commutation, $M' \to x$ is still applied with non-empty context. (The only case that would make the context disappear is when the rewrite rule before was $y \to M''$, but this case cannot occur as shown above.) So we obtain a sequence

in which $M' \to x$ is applied first, and with non-empty context. This is impossible since $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M_1,\dots,M_n\})$. So we consider a sequence not using rules of the form $M' \to x$ before $f(N_1,\dots,N_n) \to N$.

The rule $f(N_1,\dots,N_n) \to N$ commutes with the rule just before it. The rule $f(N_1,\dots,N_n) \to N$ is still applied with an empty context after commutation. So we can obtain a sequence in which $f(N_1,\dots,N_n) \to N$ is applied first. All rewrite rules after the first one are applied with a context that is an instance of N (because otherwise N is not a variable and the first rule applied with a context that is not an instance of N can be commuted with other rewrite rules so that it occurs just after $f(N_1,\dots,N_n) \to N$, so it has a critical pair with $f(N_1,\dots,N_n) \to N$, so we could obtain a shorter sequence by P2). So $M = \sigma' N$ for some σ'. Furthermore $f(M_1,\dots,M_n) = f(\sigma N_1,\dots,\sigma N_n) \to_E \sigma N \to_E^* M = \sigma' N$ for some σ. Then for all $x \in fv(N)$, $\sigma x \to_E^* \sigma' x$, so for all $x \in fv(N)$, $\Sigma \vdash \sigma x = \sigma' x$. Moreover, $\mathrm{nf}_{\mathcal{S},\Sigma}(M_1,\dots,M_n,M)$, so $\sigma x = \sigma' x$, and $M = \sigma N$, which yields the result. □

A.4 Union of disjoint equational theories

Let Σ be a signature such that its set of function symbols can be partitioned into $F_1 \cup F_2$ and its set of equations can be partitioned into $E'_1 \cup E'_2$, where E'_1 contains only function symbols in F_1 and E'_2 in F_2. Let Σ_1 be the signature obtained by considering only the equations E'_1, and Σ_2 only the equations E'_2.

Lemma 9 *If $\Sigma \vdash f(M_1,\dots,M_n) = M$, $\mathrm{nf}_{\emptyset,\Sigma}(\{M,M_1,\dots,M_n\})$, and $f \in F_i$ ($i = 1$ or 2) then $\Sigma_i \vdash f(M_1,\dots,M_n) = M$.*

Proof To prove this result, we use the decision algorithm for the word problem in a union of disjoint equational theories, by Baader and Tinelli [11, Section 4]. We use the notations of [11], and we refer the reader to that paper for details.

Assume $i = 1$. (The case $i = 2$ is symmetric.) Let us start with $S_0 = \{x_0 \not\equiv y_0, x_0 \equiv f(M_1,\dots,M_n), y_0 \equiv M\}$. Since $\Sigma \vdash f(M_1,\dots,M_n) = M$, by completeness of their algorithm, their algorithm terminates with $S = \{v \not\equiv v\} \cup T$.

Let S be a set of equations and disequations, such that all equations of S are of the form $v \equiv M$, if $v \equiv M$ and $v \equiv N$ are in S then $M = N$, and $\prec$ is acyclic on S. Let us define a substitution σ by $\sigma v = M$ when $(v \equiv M) \in S$. Since $\prec$ is acyclic on S, we can define σ^* as the substitution obtained by composing σ with itself as many times as needed so that terms do not change any more. Let $\mathrm{rec}_S(v) = \sigma^* v$.

If we apply the rules of the algorithm according to a suitable strategy (made explicit below), we can show that the algorithm preserves the following invariant:

P1. There is no equation $v \equiv M'$ in S_j such that $(v \equiv M') \prec (x_0 \equiv M'')$.

P2. If $j > 0$, then for all $v \neq x_0$ such that v occurs in S_{j-1} and S_j, we have $\mathrm{rec}_{S_{j-1}}(v) = \mathrm{rec}_{S_j}(v)$.

P3. For all $v \neq x_0$ such that v occurs in S_j, $\mathrm{rec}_{S_j}(v)$ is a subterm of $M_1,\dots,M_n,M$ (so if $v, v' \neq x_0$ occur in S_j and $\Sigma \vdash \mathrm{rec}_{S_j}(v) = \mathrm{rec}_{S_j}(v')$, then $\mathrm{rec}_{S_j}(v) = \mathrm{rec}_{S_j}(v')$, since $\mathrm{nf}_{\emptyset,\Sigma}(\{M,M_1,\dots,M_n\})$).

P4. If $j > 0$ and $x_0 \equiv M'' \in S_j$, then $\Sigma_1 \vdash \mathrm{rec}_{S_{j-1}}(x_0) = \mathrm{rec}_{S_j}(x_0)$.

P5. When $x_0 \equiv M'' \in S_j$, M'' is a non-variable 1-term.

P6. If $j > 0$, $u \not\equiv u' \in S_{j-1}$, and $v \not\equiv v' \in S_j$, then $\Sigma_1 \vdash \mathrm{rec}_{S_{j-1}}(u) = \mathrm{rec}_{S_j}(v)$ and $\Sigma_1 \vdash \mathrm{rec}_{S_{j-1}}(u') = \mathrm{rec}_{S_j}(v')$.

During the first stage (construction of the abstraction system), these properties are obvious. We even have $\mathrm{rec}_{S_{j-1}}(v) = \mathrm{rec}_{S_j}(v)$ for all v that occur in S_{j-1}, and the disequation $x_0 \not\equiv y_0$ is not changed.

During the second stage (application of Coll1, Coll2, Ident, Simpl), we do not apply Simpl since the authors remark that it is not necessary. We show that if S_{j-1} is transformed into S_j by Coll1, Coll2, or Ident, and S_{j-1} satisfies the invariant, then so does S_j.

- For Coll1 and Coll2 with $x \neq x_0$, $\Sigma_i \vdash y = t$, so $\Sigma \vdash \mathrm{rec}_{S_{j-1}}(x) = \mathrm{rec}_{S_{j-1}}(y)$, so by P3, $\mathrm{rec}_{S_{j-1}}(x) = \mathrm{rec}_{S_{j-1}}(y)$, so $y = t$. Then for all v that occur in S_j, $\mathrm{rec}_{S_{j-1}}(v) = \mathrm{rec}_{S_j}(v)$, so we have P2 and P4 for S_j. P3 holds for S_j since P2 holds for S_j and P3 holds for S_{j-1}. We have $\mathrm{rec}_{S_{j-1}}(x) = \mathrm{rec}_{S_{j-1}}(y) = \mathrm{rec}_{S_j}(y)$, so P6 follows. P1 and P5 are easy to show.

- For Coll1 with $x = x_0$, $\Sigma_1 \vdash y = t$, and $T\{r/x_0\} = T$ since x_0 does not occur in the right-hand side of equalities by P1. So for all v that occur in S_j (that is, all v that occur in S_{j-1} except x_0), $\mathrm{rec}_{S_{j-1}}(v) = \mathrm{rec}_{S_j}(v)$, so we have P2 for S_j; P3 follows. P4 and P5 hold since S_j contains no equation of the form $x_0 \equiv M''$. Since $\Sigma \vdash y = t$, we have $\Sigma_1 \vdash \mathrm{rec}_{S_{j-1}}(x_0) = \mathrm{rec}_{S_{j-1}}(y) = \mathrm{rec}_{S_j}(y)$, so P6 follows. P1 is easy to show.

- For Coll2 with $x = x_0$, $\Sigma_1 \vdash y = t$, and T is replaced with $T\{y/x_0\}$, which modifies only the disequation, since x_0 does not occur in the right-hand side of equalities by P1. We conclude as in the previous case.

- For Ident, we never apply Ident with $y = x_0$; when Ident would be applicable with $y = x_0$, we apply instead Ident with $x = x_0$ (which is possible by P1).

  If we apply Ident with $x, y \neq x_0$, then $\Sigma_i \vdash s = t$, so $\Sigma \vdash \mathrm{rec}_{S_{j-1}}(x) = \mathrm{rec}_{S_{j-1}}(y)$, so by P3, $\mathrm{rec}_{S_{j-1}}(x) = \mathrm{rec}_{S_{j-1}}(y)$, so $s = t$. Then, for all v that occur in S_j, $\mathrm{rec}_{S_{j-1}}(v) = \mathrm{rec}_{S_j}(v)$, so we have P2 and P4; P3 follows. We have $\mathrm{rec}_{S_{j-1}}(x) = \mathrm{rec}_{S_{j-1}}(y) = \mathrm{rec}_{S_j}(y)$, so P6 follows. P1 and P5 are easy to show.

  If we apply Ident with $x = x_0, y \neq x_0$, then $\Sigma_1 \vdash s = t$. x_0 does not occur in the right-hand side of equalities by P1. So replacing x_0 with y in T changes only the disequation. Then for all v that occur in S_j, $\mathrm{rec}_{S_{j-1}}(v) = \mathrm{rec}_{S_j}(v)$, so we have P2 and P4; P3 follows. Since $\Sigma_1 \vdash s = t$, we have $\Sigma \vdash \mathrm{rec}_{S_{j-1}}(x_0) = \mathrm{rec}_{S_{j-1}}(y) = \mathrm{rec}_{S_j}(y)$, so P6 follows. P1 and P5 are easy to show.

Since, in the end, S contains $v \not\equiv v$, by P6, we have $\Sigma_1 \vdash \mathrm{rec}_{S_0}(x_0) = \mathrm{rec}_S(v)$ and $\Sigma_1 \vdash \mathrm{rec}_{S_0}(y_0) = \mathrm{rec}_S(v)$ which implies $\Sigma_1 \vdash f(M_1, \dots, M_n) = M$. □

This result can be used to prove the correctness of Algorithm 3.

Lemma 10 *The signature Σ' built by Algorithm 3 models Σ.*

Proof The set of function symbols of Σ can be partitioned into $F_1 \cup F_2$, where E_{conv} contains only function symbols in F_1 and E_{lin} in F_2. Let Σ_1 be the signature obtained by considering only equations E_{conv}, and Σ_2 only E_{lin}.

Because of the particular way in which we prove that subsets E_i are convergent, we have that their union E_{conv} is also convergent, so we can apply Algorithm 1 to E_{conv}. (When we prove termination of each E_i via a lexicographic path ordering, we order the function symbols of E_i. We order the function symbols of E_{conv} by the union of these orderings. Then the corresponding lexicographic path ordering shows the termination of E_{conv}. The confluence of E_{conv} follows from the confluence of every E_i by the critical-pair theorem.)

Properties S1 and S3 are obvious. We prove Property S2 by induction on M:

- Cases $M = a$ and $M = x$: Let $M' = M$; by Lemma 4, $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M\})$.

- Case $M = f(M_1, \ldots, M_n)$: By induction hypothesis, there exist $M'_1, \ldots, M'_n$ such that $\Sigma \vdash M_i = M'_i$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'_1, \ldots, M'_n\})$. (For M_i, we apply the induction hypothesis with $\mathcal{M} \cup \{M'_1, \ldots, M'_{i-1}\}$ instead of $\mathcal{M}$.)

  Case 1: there exists a subterm M' of $\mathcal{M} \cup \{M'_1, \ldots, M'_n\}$ such that $\Sigma \vdash f(M_1, \ldots, M_n) = M'$. Then M' is irreducible by $\mathcal{S}$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$, so we have the result.

  Case 2: there exists no subterm M'' of $\mathcal{M} \cup \{M'_1, \ldots, M'_n\}$ such that $\Sigma \vdash f(M_1, \ldots, M_n) = M''$.

  Case 2.1: Assume $f \in F_2$. Let $M' = f(M'_1, \ldots, M'_n)$. We have $\Sigma \vdash f(M_1, \ldots, M_n) = M'$. Moreover, M' is irreducible by $\mathcal{S}$ since $M'_1, \ldots, M'_n$ are, $f \in F_2$, and no rewrite rule of $\mathcal{S}$ contains a function symbol in F_2 or a variable in the left-hand side. Then $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$ since the subterms of $\mathcal{M} \cup \{M'\}$ are the subterms of $\mathcal{M} \cup \{M'_1, \ldots, M'_n\}$ and the term M', $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'_1, \ldots, M'_n\})$, and the new subterm M' is different from any subterms of $\mathcal{M} \cup \{M'_1, \ldots, M'_n\}$ modulo the equational theory of Σ.

  Case 2.2: Assume $f \in F_1$. Let $M' = f(M'_1, \ldots, M'_n){\downarrow}$. We have $\Sigma \vdash f(M_1, \ldots, M_n) = M'$. Moreover, M' is irreducible by $\mathcal{S}$ by definition. If $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$ was wrong, there would exist N and N' subterms of $\mathcal{M} \cup \{M'\}$ such that $\Sigma \vdash N = N'$ and $N \neq N'$. Let us choose such terms N and N' such that the pair $(\max(\mathit{size}(N), \mathit{size}(N')), \min(\mathit{size}(N), \mathit{size}(N')))$ ordered lexicographically is minimal. When $\mathit{size}(N) < \mathit{size}(N')$, we swap N and N', so that we always have $\mathit{size}(N) \geq \mathit{size}(N')$. Let $N = f'(N_1, \ldots, N_{n'})$. We have $\mathrm{nf}_{\mathcal{S},\Sigma}(N_1, \ldots, N_{n'}, N')$. (If $\mathrm{nf}_{\mathcal{S},\Sigma}(N_1, \ldots, N_{n'}, N')$ was not true, considering subterms of $N_1, \ldots, N_{n'}, N'$ that falsify $\mathrm{nf}_{\mathcal{S},\Sigma}(N_1, \ldots, N_{n'}, N')$ would yield a smaller counterexample.)

  Notice that $\mathrm{nf}_{\mathcal{S},\Sigma}(N_1, \ldots, N_{n'}, N')$ implies $\mathrm{nf}_{\emptyset,\Sigma}(N_1, \ldots, N_{n'}, N')$, so we can apply Lemma 9.

  If $f' \in F_1$, then $\Sigma_1 \vdash f'(N_1, \ldots, N_{n'}) = N'$ by Lemma 9. Hence $f'(N_1, \ldots, N_{n'}){\downarrow} = N'{\downarrow}$. The terms N' and $f'(N_1, \ldots, N_{n'})$ are subterms of $\mathcal{M} \cup \{M'\}$, so they are irreducible by $\mathcal{S}$, so $f'(N_1, \ldots, N_{n'}) = N'$. Hence, we have a contradiction.

  If $f' \in F_2$, then $\Sigma_2 \vdash f'(N_1, \ldots, N_{n'}) = N'$ by Lemma 9. Since the reduction of $f(M'_1, \ldots, M'_n)$ into M' modifies only the top-level context of M' within F_1, all subterms of $\mathcal{M} \cup \{M'\}$ with root symbol in F_2 are also subterms of $\mathcal{M} \cup \{M'_1, \ldots, M'_n\}$, so they satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}$. So the root symbol of N' is in F_1. Let $N' = f''(N'_1, \ldots, N'_{n''})$, $f'' \in F_1$. If $\mathrm{nf}_{\mathcal{S},\Sigma}(N'_1, \ldots, N'_{n''}, N)$, we can apply the case $f' \in F_1$ above to $\Sigma \vdash f''(N'_1, \ldots, N'_{n''}) = N$. Otherwise, the counterexample to $\mathrm{nf}_{\mathcal{S},\Sigma}(N'_1, \ldots, N'_{n''}, N)$ is not smaller than $\Sigma \vdash N = N'$ since it is minimal, and $\mathit{size}(N) \geq \mathit{size}(N')$, so the counterexample to $\mathrm{nf}_{\mathcal{S},\Sigma}(N'_1, \ldots, N'_{n''}, N)$ consists of two subterms of N; this situation is impossible since $N = f'(N_1, \ldots, N_{n'})$ is a subterm of $\mathcal{M} \cup \{M'_1, \ldots, M'_n\}$, so all its subterms satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}$.

  Hence we have $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M} \cup \{M'\})$.

Finally, we prove Property S4. Let $\Sigma \vdash f(M_1, \ldots, M_n) = M$ with $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M, M_1, \ldots, M_n\})$. If $f \in F_i$ $(i = 1, 2)$, then $\Sigma_i \vdash f(M_1, \ldots, M_n) = M$ by Lemma 9 (since $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{M}')$ implies $\mathrm{nf}_{\emptyset,\Sigma}(\mathcal{M}')$). If $i = 1$, we conclude by Property S4 for Algorithm 1. If $i = 2$, we conclude by Property S4 for Algorithm 2. □

B Proofs of Lemmas 1 and 2

From this point on, we assume that Σ' models Σ. We say that a term or a term evaluation is plain when it does not contain diff.

B.1 Preliminary lemmas

The following lemma shows the soundness of $D' \Downarrow' (M', \sigma')$ with respect to $D \Downarrow_{\Sigma'} M$.

Lemma 11 *Let σ be a closed substitution.*

Let D be a plain term evaluation. If $\sigma D \Downarrow_{\Sigma'} M$, then there exist M', σ_1, and σ'_1 such that $D \Downarrow' (M', \sigma_1)$, $M = \sigma'_1 M'$, and $\sigma = \sigma'_1 \sigma_1$ except on fresh variables introduced in the computation of $D \Downarrow' (M', \sigma_1)$.

Let $D_1, \dots, D_n$ be plain term evaluations. If for all $i \in \{1, \dots, n\}$, $\sigma D_i \Downarrow_{\Sigma'} M_i$, then there exist $M'_1, \dots, M'_n$, σ_1, and σ'_1 such that $(D_1, \dots, D_n) \Downarrow' ((M'_1, \dots, M'_n), \sigma_1)$, $M_i = \sigma'_1 M'_i$ for all $i \in \{1, \dots, n\}$, and $\sigma = \sigma'_1 \sigma_1$ except on fresh variables introduced in the computation of $(D_1, \dots, D_n) \Downarrow' ((M'_1, \dots, M'_n), \sigma_1)$.

Proof The proof is by mutual induction following the definition of $\Downarrow'$.

- Case $D = M'$: Take $\sigma_1 = \emptyset$, $\sigma'_1 = \sigma$. Since $M = \sigma M'$, we have the result.
- Case $D = \text{eval } h(D_1, \dots, D_n)$: Since $\text{eval } h(\sigma D_1, \dots, \sigma D_n) \Downarrow_{\Sigma'} M$, there exist $h(N_1, \dots, N_n) \to N$ in $\text{def}_{\Sigma'}(h)$ and σ_m such that $\sigma D_i \Downarrow_{\Sigma'} \sigma_m N_i$ and $M = \sigma_m N$.

  By induction hypothesis, there exist M'_i, σ_1, and σ'_1 such that $(D_1, \dots, D_n) \Downarrow' ((M'_1, \dots, M'_n), \sigma_1)$, $\sigma_m N_i = \sigma'_1 M'_i$ for all $i \in \{1, \dots, n\}$, and $\sigma = \sigma'_1 \sigma_1$ except on fresh variables introduced in the computation of $(D_1, \dots, D_n) \Downarrow' ((M'_1, \dots, M'_n), \sigma_1)$.

  Let σ_u be the most general unifier of M'_i and N_i for $i \in \{1, \dots, n\}$. (The substitution σ_u exists since $\sigma_m N_i = \sigma'_1 M'_i$.) Then $\text{eval } h(D_1, \dots, D_n) \Downarrow' (\sigma_u N, \sigma_u \sigma_1)$. The substitution that maps variables of N_i, N as σ_m and other variables as σ'_1 is a unifier of M'_i and N_i, so there exists σ''_1 such that $\sigma_m = \sigma''_1 \sigma_u$ on variables of N_i, N, and $\sigma'_1 = \sigma''_1 \sigma_u$ on other variables.

  Then $\sigma''_1 \sigma_u N = \sigma_m N = M$ and $\sigma''_1 \sigma_u \sigma_1 = \sigma'_1 \sigma_1 = \sigma$ except on fresh variables introduced in the computation of $(D_1, \dots, D_n) \Downarrow' ((M'_1, \dots, M'_n), \sigma_1)$ and variables of $N_1, \dots, N_n, N$, that is, fresh variables introduced in the computation of $D \Downarrow' (\sigma_u N, \sigma_u \sigma_1)$.
- Case $(D_1, \dots, D_n)$: We have, for all i in $\{1, \dots, n\}$, $\sigma D_i \Downarrow_{\Sigma'} M_i$.

  By induction hypothesis, there exist M'_i, σ_1, and σ'_1 such that $(D_1, \dots, D_{n-1}) \Downarrow' ((M'_1, \dots, M'_{n-1}), \sigma_1)$, $M_i = \sigma'_1 M'_i$ for all $i \in \{1, \dots, n-1\}$, and $\sigma = \sigma'_1 \sigma_1$ except on fresh variables introduced in the computation of $(D_1, \dots, D_{n-1}) \Downarrow' ((M'_1, \dots, M'_{n-1}), \sigma_1)$.

  Then $\sigma D_n = \sigma'_1 \sigma_1 D_n$, so $\sigma'_1(\sigma_1 D_n) \Downarrow_{\Sigma'} M_n$. So by induction hypothesis, there exist M'_n, σ_2, and σ'_2 such that $\sigma_1 D_n \Downarrow' (M'_n, \sigma_2)$, $M_n = \sigma'_2 M'_n$, and $\sigma'_1 = \sigma'_2 \sigma_2$ except on fresh variables introduced in the computation of $\sigma_1 D_n \Downarrow' (M'_n, \sigma_2)$.

  Hence $(D_1, \dots, D_n) \Downarrow' ((\sigma_2 M'_1, \dots, \sigma_2 M'_{n-1}, M'_n), \sigma_2 \sigma_1)$, $M_i = \sigma'_1 M'_i = \sigma'_2(\sigma_2 M'_i)$ for all $i \in \{1, \dots, n-1\}$, $M_n = \sigma'_2 M'_n$, and $\sigma = \sigma'_1 \sigma_1 = \sigma'_2 \sigma_2 \sigma_1$ except on fresh variables introduced in the computation of $(D_1, \dots, D_n) \Downarrow' ((\sigma_2 M'_1, \dots, \sigma_2 M'_{n-1}, M'_n), \sigma_2 \sigma_1)$. □

Lemma 12 *Let σ be a closed substitution and M a plain term. If $\Sigma \vdash M' = \sigma M$ and $\text{nf}_{\mathcal{S},\Sigma}(\{M'\} \cup \{\sigma x \mid x \in fv(M)\})$ then $\sigma \text{addeval}(M) \Downarrow_{\Sigma'} M'$.*

Proof The proof is by induction on M.

- Case $M = x$: We have $\Sigma \vdash \sigma x = \sigma M = M'$. Since $\text{nf}_{\mathcal{S},\Sigma}(\{\sigma x, M'\})$, $\sigma x = M'$. Moreover, $\sigma \text{addeval}(M) = \sigma x \Downarrow_{\Sigma'} \sigma x = M'$.
- Case $M = a$: Since $\Sigma \vdash M' = \sigma M$ and $\text{nf}_{\mathcal{S},\Sigma}(\{M'\})$, we have $M' = a$ by Lemma 4, so $\sigma \text{addeval}(M) = a \Downarrow_{\Sigma'} a = M'$.

- Case $M = f(M_1, \dots, M_n)$: We have $\Sigma \vdash M' = \sigma M = f(\sigma M_1, \dots, \sigma M_n)$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M'\} \cup \{\sigma x \mid x \in \mathit{fv}(M)\})$. By Property S2, there exist $M'_1, \dots, M'_n$ such that $\Sigma \vdash \sigma M_i = M'_i$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M', M'_1, \dots, M'_n\} \cup \{\sigma x \mid x \in \mathit{fv}(M)\})$. By Property S4, there exist $f(N_1, \dots, N_n) \to N$ in $\mathrm{def}_{\Sigma'}(f)$ and σ' such that $M' = \sigma' N$ and $\sigma' N_i = M'_i$ for all $i \in \{1, \dots, n\}$. By induction hypothesis, $\sigma \mathrm{addeval}(M_i) \Downarrow_{\Sigma'} M'_i = \sigma' N_i$ for all $i \in \{1, \dots, n\}$. By definition of $\Downarrow_{\Sigma'}$, $\sigma \mathrm{addeval}(M) = \mathrm{eval}\ f(\sigma \mathrm{addeval}(M_1), \dots, \sigma \mathrm{addeval}(M_n)) \Downarrow_{\Sigma'} \sigma' N = M'$. □

The following lemma shows the soundness of the rewrite rules of h in Σ' with respect to these rewrite rules in Σ. When h is a destructor, this is proved using the previous two lemmas, and when h is a constructor, this follows from the definition of "Σ' models Σ". Lemma 14 extends this result to a term evaluation D by induction on D.

Lemma 13 *If $h(N_1, \dots, N_n) \to N$ is in $\mathrm{def}_\Sigma(h)$, $\Sigma \vdash M_i = \sigma N_i$ for all $i \in \{1, \dots, n\}$, $\Sigma \vdash M = \sigma N$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M_1, \dots, M_n, M\})$, then there exist $h(N'_1, \dots, N'_n) \to N'$ in $\mathrm{def}_{\Sigma'}(h)$ and σ' such that $M_i = \sigma' N'_i$ for all $i \in \{1, \dots, n\}$ and $M = \sigma' N'$.*

Proof Case 1: h is a constructor in Σ. We have $\Sigma \vdash M = h(M_1, \dots, M_n)$. The result follows from Property S4.

Case 2: h is a destructor in Σ. By Property S2, there exists σ_0 such that $\Sigma \vdash \sigma_0 x = \sigma x$ for all $x \in \mathit{fv}(N_1, \dots, N_n, N)$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M_1, \dots, M_n, M\} \cup \{\sigma_0 x \mid x \in \mathit{fv}(N_1, \dots, N_n, N)\})$. So $\Sigma \vdash M = \sigma_0 N$ and $\Sigma \vdash M_i = \sigma_0 N_i$ for all $i \in \{1, \dots, n\}$. By Lemma 12, $\sigma_0 \mathrm{addeval}(N) \Downarrow_{\Sigma'} M$ and $\sigma_0 \mathrm{addeval}(N_i) \Downarrow_{\Sigma'} M_i$ for all $i \in \{1, \dots, n\}$. By Lemma 11, there exist $N'_1, \dots, N'_n$, N', σ_1, and σ' such that $\mathrm{addeval}(N_1, \dots, N_n, N) \Downarrow' ((N'_1, \dots, N'_n, N'), \sigma_1)$, $\sigma' N'_i = M_i$ for all $i \in \{1, \dots, n\}$, and $\sigma' N' = M$. Then $h(N'_1, \dots, N'_n) \to N'$ is in $\mathrm{def}_{\Sigma'}(h)$, $\sigma' N'_i = M_i$ for all $i \in \{1, \dots, n\}$, and $\sigma' N' = M$. □

Lemma 14 *Let D be a plain term evaluation. If $D \Downarrow_\Sigma M$, $\Sigma \vdash M' = M$, $\Sigma \vdash D' = D$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M', D'\})$, then $D' \Downarrow_{\Sigma'} M'$.*

Proof The proof is by induction on D.

- Case $D = M$: We have $M \Downarrow_\Sigma M$, so $\Sigma \vdash D' = D = M = M'$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M', D'\})$ so $D' = M'$, and $D' \Downarrow_{\Sigma'} M'$.

- Case $D = \mathrm{eval}\ h(D_1, \dots, D_n)$: Since $D \Downarrow_\Sigma M$, we have that $h(N_1, \dots, N_n) \to N$ is in $\mathrm{def}_\Sigma(h)$, $D_i \Downarrow_\Sigma M_i$ and $\Sigma \vdash \sigma N_i = M_i$ for all $i \in \{1, \dots, n\}$, and $\sigma N = M$. So $\Sigma \vdash \sigma N = M'$. Since $\Sigma \vdash D' = D$, we have $D' = \mathrm{eval}\ h(D'_1, \dots, D'_n)$, with $\Sigma \vdash D'_i = D_i$ for all $i \in \{1, \dots, n\}$. By Property S2, there exist $M'_1, \dots, M'_n$ such that $\Sigma \vdash M_i = M'_i$ for all $i \in \{1, \dots, n\}$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M', D', M'_1, \dots, M'_n\})$. By induction hypothesis, $D'_i \Downarrow_{\Sigma'} M'_i$ for all $i \in \{1, \dots, n\}$. By Lemma 13, there exist $h(N'_1, \dots, N'_n) \to N'$ in $\mathrm{def}_{\Sigma'}(h)$ and σ' such that $M' = \sigma' N'$ and $\sigma' N'_i = M'_i$ for all $i \in \{1, \dots, n\}$. Then $D' \Downarrow_{\Sigma'} \sigma' N' = M'$. □

We define the function removeeval such that $\mathrm{removeeval}(D) = M$ where D is a term evaluation that contains no destructor, and M is the term obtained by removing any eval before the function symbols of D.

Lemma 15 *Assume that D is a plain term evaluation that contains no destructor. If $D \Downarrow' (M, \sigma)$ then $\Sigma \vdash \sigma \mathrm{removeeval}(D) = M$.*

Assume that $D_1, \dots, D_n$ are plain term evaluations that contain no destructor. If $(D_1, \dots, D_n) \Downarrow' ((M_1, \dots, M_n), \sigma)$ then $\Sigma \vdash \sigma \mathrm{removeeval}(D_i) = M_i$ for all $i \in \{1, \dots, n\}$.

Proof The proof is by mutual induction following the definition of $\Downarrow'$.

- Case $D = M$: We have $\sigma = \emptyset$, so $\Sigma \vdash \sigma M = M$.

- Case $D = \text{eval } f(D_1, \ldots, D_n)$: We have $\text{eval } f(D_1, \ldots, D_n) \Downarrow' (\sigma_u N, \sigma_u \sigma)$ where $(D_1, \ldots, D_n) \Downarrow' ((M_1, \ldots, M_n), \sigma)$, f is a constructor in Σ, $f(N_1, \ldots, N_n) \rightarrow N$ is in $\text{def}_{\Sigma'}(f)$ (with new variables), and σ_u is the most general unifier of (M_1, N_1), $\ldots$, (M_n, N_n). Then by Property S3, $\Sigma \vdash f(N_1, \ldots, N_n) = N$. By induction hypothesis, $\Sigma \vdash \sigma\text{removeeval}(D_i) = M_i$. Moreover we have $\sigma_u M_i = \sigma_u N_i$. Hence we obtain $\Sigma \vdash \sigma_u \sigma\text{removeeval}(\text{eval } f(D_1, \ldots, D_n)) = f(\sigma_u \sigma\text{removeeval}(D_1), \ldots, \sigma_u \sigma\text{removeeval}(D_n)) = f(\sigma_u M_1, \ldots, \sigma_u M_n) = f(\sigma_u N_1, \ldots, \sigma_u N_n) = \sigma_u N$.

- Case $(D_1, \ldots, D_n)$: We have $(D_1, \ldots, D_n) \Downarrow' ((\sigma' M_1, \ldots, \sigma' M_{n-1}, M_n), \sigma' \sigma)$ where $(D_1, \ldots, D_{n-1}) \Downarrow' ((M_1, \ldots, M_{n-1}), \sigma)$ and $\sigma D_n \Downarrow' (M_n, \sigma')$. Then by induction hypothesis, $\Sigma \vdash \sigma\text{removeeval}(D_i) = M_i$ for $i \in \{1, \ldots, n-1\}$ and $\Sigma \vdash \sigma'\text{removeeval}(\sigma D_n) = M_n$. Hence, $\Sigma \vdash \sigma' \sigma\text{removeeval}(D_i) = \sigma' M_i$ for $i \in \{1, \ldots, n-1\}$ and $\Sigma \vdash \sigma' \sigma\text{removeeval}(D_n) = M_n$. □

The following two lemmas show a completeness property: we do not lose precision by translating computation in Σ into computations in Σ'. The proof of Lemma 16 relies on Lemma 15 for destructor applications.

Lemma 16 *If $h(N_1, \ldots, N_n) \rightarrow N$ is in $\text{def}_{\Sigma'}(h)$ then there exists $h(N'_1, \ldots, N'_n) \rightarrow N'$ in $\text{def}_\Sigma(h)$ and σ such that $\Sigma \vdash N_i = \sigma N'_i$ for all $i \in \{1, \ldots, n\}$ and $\Sigma \vdash N = \sigma N'$.*

Proof Case 1: h is a constructor in Σ. By Property S3, $\Sigma \vdash h(N_1, \ldots, N_n) = N$. Let σ be defined by $\sigma x_i = N_i$ for all $i \in \{1, \ldots, n\}$, $N'_i = x_i$ for all $i \in \{1, \ldots, n\}$, and $N' = h(x_1, \ldots, x_n)$. We have $h(N'_1, \ldots, N'_n) \rightarrow N'$ in $\text{def}_\Sigma(h)$ because $h(x_1, \ldots, x_n) \rightarrow h(x_1, \ldots, x_n)$ is in $\text{def}_\Sigma(h)$. We also have $\Sigma \vdash N_i = \sigma N'_i$ for all $i \in \{1, \ldots, n\}$ and $\Sigma \vdash N = h(N_1, \ldots, N_n) = \sigma N'$.

Case 2: h is a destructor in Σ. Then there exists $h(N'_1, \ldots, N'_n) \rightarrow N'$ in $\text{def}_\Sigma(h)$, such that $\text{addeval}(N'_1, \ldots, N'_n, N') \Downarrow' ((N_1, \ldots, N_n, N), \sigma)$. By Lemma 15, $\Sigma \vdash N = \sigma N'$ and for all $i \in \{1, \ldots, n\}$, $\Sigma \vdash N_i = \sigma N'_i$. □

Lemma 17 *Let D be a plain term evaluation. If $\Sigma \vdash D' = D$ and $D' \Downarrow_{\Sigma'} M'$ then $D \Downarrow_\Sigma M$ for some M such that $\Sigma \vdash M = M'$.*

Proof The proof is by induction on D.

- Case $D = M$: We have $D \Downarrow_\Sigma M$. Moreover $\Sigma \vdash D' = D$, so D' is also a term, and $M' = D'$. Finally, $D = M$, $D' = M'$, and $\Sigma \vdash D' = D$, so $\Sigma \vdash M = M'$.

- Case $D = \text{eval } h(D_1, \ldots, D_n)$: Since $\Sigma \vdash D' = D$, we have $D' = \text{eval } h(D'_1, \ldots, D'_n)$ with $\Sigma \vdash D'_i = D_i$. Since $D' \Downarrow_{\Sigma'} M'$, there exist $h(N_1, \ldots, N_n) \rightarrow N$ in $\text{def}_{\Sigma'}(h)$ and σ such that $M' = \sigma N$, and for all $i \in \{1, \ldots, n\}$, $D'_i \Downarrow_{\Sigma'} \sigma N_i$. By induction hypothesis, $D_i \Downarrow_\Sigma M_i$ with $\Sigma \vdash M_i = \sigma N_i$.

  By Lemma 16, there exist $h(N'_1, \ldots, N'_n) \rightarrow N'$ in $\text{def}_\Sigma(h)$ and σ', such that $\Sigma \vdash N = \sigma' N'$ and for all $i \in \{1, \ldots, n\}$, $\Sigma \vdash N_i = \sigma' N'_i$. Then $D_i \Downarrow_\Sigma M_i$, $\Sigma \vdash M_i = \sigma N_i = \sigma\sigma' N'_i$, and $h(N'_1, \ldots, N'_n) \rightarrow N'$ is in $\text{def}_\Sigma(h)$, so $D \Downarrow_\Sigma \sigma\sigma' N'$. Moreover, $\Sigma \vdash M' = \sigma N = \sigma\sigma' N'$. □

The following lemma is useful to deal with rule (Red Fun 2): when D fails to evaluate, the lemma ensures that D' also fails to evaluate, even with the equational theory of Σ. To this end, Lemma 18 requires $D' \Downarrow_\Sigma M'$, whereas Lemma 17 requires $D' \Downarrow_{\Sigma'} M'$.

Lemma 18 *Let D be a plain term evaluation. If $\Sigma \vdash D' = D$ and $D' \Downarrow_\Sigma M'$ then $D \Downarrow_\Sigma M$ for some M such that $\Sigma \vdash M = M'$.*

Proof The proof is by induction on D.

- Case $D = M$: We have $D \Downarrow_\Sigma M$. Moreover $\Sigma \vdash D' = D$, so D' is also a term, and $M' = D'$. Finally, $D = M$, $D' = M'$, and $\Sigma \vdash D' = D$, so $\Sigma \vdash M = M'$.
- Case $D = \mathrm{eval}\ h(D_1, \ldots, D_n)$: Since $\Sigma \vdash D' = D$, we have $D' = \mathrm{eval}\ h(D'_1, \ldots, D'_n)$ with $\Sigma \vdash D'_i = D_i$. Since $D' \Downarrow_\Sigma M'$, there exist $h(N_1, \ldots, N_n) \to N$ in $\mathrm{def}_\Sigma(h)$ and σ such that $M' = \sigma N$, and for all $i \in \{1, \ldots, n\}$, $D'_i \Downarrow_\Sigma M'_i$ with $\Sigma \vdash M'_i = \sigma N_i$. By induction hypothesis, $D_i \Downarrow_\Sigma M_i$ with $\Sigma \vdash M_i = \sigma N_i$. Then $D = \mathrm{eval}\ h(D_1, \ldots, D_n) \Downarrow_\Sigma \sigma N$ and $\Sigma \vdash \sigma N = M'$. □

B.2 Proof of Lemma 1

Lemma 1 is an obvious consequence of the following lemma.

Lemma 19 *Let P_0 be a closed, unevaluated biprocess.*

If $P_0 \to^*_\Sigma\equiv P'_0$, $\Sigma \vdash Q'_0 = P'_0$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{Q'_0\})$, then $P_0 \to^*_{\Sigma',\Sigma}\equiv Q'_0$ by a reduction whose intermediate biprocesses Q all satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}(\{Q\})$.

Conversely, if $P_0 \to^*_{\Sigma',\Sigma}\equiv Q'_0$ then there exists P'_0 such that $\Sigma \vdash Q'_0 = P'_0$ and $P_0 \to^*_\Sigma\equiv P'_0$.

Proof We write $VC(P)$ when P is a closed process whose terms M are either variables or terms of the form $\mathsf{diff}[M_1, M_2]$ where M_1 and M_2 are closed terms that do not contain diff. (Function symbols prefixed by eval are not constrained.) We have the following properties:

P1. If $VC(P)$ and $P \equiv P'$ then $VC(P')$. The proof is by induction on the derivation of $P \equiv P'$. All cases are easy, since $\equiv$ cannot change terms.

P2. If $VC(P)$ and $P \to_\Sigma P'$ then $VC(P')$. The proof is by induction on the derivation of $P \to_\Sigma P'$. The only change of terms is done by the substitution $\{M/x\}$ in the rules (Red I/O) and (Red Fun 1). This substitution replaces a variable with a closed term $M = \mathsf{diff}[M_1, M_2]$, hence the result. (For (Red I/O), M is of the form $\mathsf{diff}[M_1, M_2]$ because of $VC(P)$.)

P3. If $VC(P\{\mathsf{diff}[M_1, M_2]/x\})$, $\Sigma \vdash P\{\mathsf{diff}[M_1, M_2]/x\} = P''$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{P''\})$, then there exist P', M'_1, and M'_2 such that $\Sigma \vdash P = P'$, $\Sigma \vdash M_1 = M'_1$, $\Sigma \vdash M_2 = M'_2$, $P'' = P'\{\mathsf{diff}[M'_1, M'_2]/x\}$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{P', M'_1, M'_2\})$.

Since P_0 is closed and unevaluated, $VC(P_0)$. Therefore, by P1 and P2, if $P_0 \to^*_\Sigma\equiv P$, then $VC(P)$. Moreover, the only process P such that $\Sigma \vdash P_0 = P$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{P\})$ is P_0 by Lemma 4.

Let us show that, if $P \equiv P'$, $\Sigma \vdash Q' = P'$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q'\})$, then there exists Q such that $\Sigma \vdash Q = P$, $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q\})$, and $Q \equiv Q'$. The proof is by induction on the derivation of $P \equiv P'$. All cases are easy, since $\equiv$ does not depend on terms.

Let us show that, if $VC(P)$, $P \to_\Sigma P'$, $\Sigma \vdash Q' = P'$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q'\})$, then there exists Q such that $\Sigma \vdash Q = P$, $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q\})$, and $Q \to_{\Sigma',\Sigma} Q'$. The proof is by induction on the derivation of $P \to_\Sigma P'$.

- Case (Red I/O): Since $VC(P)$, we have $P = \overline{\mathsf{diff}[M_1, M_2]}\langle N\rangle.R \mid \mathsf{diff}[M'_1, M'_2](x).R' \to_\Sigma R \mid R'\{N/x\} = P'$ with $\Sigma \vdash M_1 = M'_1$ and $\Sigma \vdash M_2 = M'_2$. Since $\Sigma \vdash Q' = P'$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q'\})$, we have $Q' = R_1 \mid R'_1\{N_1/x\}$ for some R_1, R'_1, N_1 such that $\Sigma \vdash R_1 = R$, $\Sigma \vdash R'_1 = R'$, $\Sigma \vdash N_1 = N$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{R_1, R'_1, N_1\})$ by P3.

  By Property S2, there exist M''_1 and M''_2 such that $\Sigma \vdash M''_1 = M_1 = M'_1$, $\Sigma \vdash M''_2 = M_2 = M'_2$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{R_1, R'_1, N_1, M''_1, M''_2\})$.

  We let $Q = \overline{\mathsf{diff}[M''_1, M''_2]}\langle N_1\rangle.R_1 \mid \mathsf{diff}[M''_1, M''_2](x).R'_1$. Then $\Sigma \vdash Q = P$. Moreover $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q\})$ since $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{R_1, R'_1, N_1, M''_1, M''_2\})$, and $Q \to_{\Sigma',\Sigma} Q'$, hence the result.

- Case (Red Fun 1): We have $P = let\ x = D\ in\ R\ else\ R' \rightarrow_\Sigma R\{\mathsf{diff}[M,M']/x\} = P'$ with $\mathsf{fst}(D) \Downarrow_\Sigma M$ and $\mathsf{snd}(D) \Downarrow_\Sigma M'$. Since $\Sigma \vdash Q' = P'$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q'\})$, we have $Q' = R_1\{\mathsf{diff}[M_1, M_1']/x\}$ for some R_1, M_1, M_1' such that $\Sigma \vdash R_1 = R$, $\Sigma \vdash M_1 = M$, $\Sigma \vdash M_1' = M'$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{R_1, M_1, M_1'\})$ by P3.

  By Property S2, there exist D_1 and R_1' such that $\Sigma \vdash D_1 = D$, $\Sigma \vdash R_1' = R'$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{D_1, R_1', R_1, M_1, M_1'\})$. By Lemma 14, $\mathsf{fst}(D_1) \Downarrow_{\Sigma'} M_1$ and $\mathsf{snd}(D_1) \Downarrow_{\Sigma'} M_1'$. Let $Q = let\ x = D_1\ in\ R_1\ else\ R_1'$. Then $\Sigma \vdash Q = P$, $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q\})$, and $Q \rightarrow_{\Sigma',\Sigma} Q'$.

- Case (Red Fun 2): We have $P = let\ x = D\ in\ R\ else\ P' \rightarrow_\Sigma P'$, there exists no M such that $\mathsf{fst}(D) \Downarrow_\Sigma M$, and there exists no M' such that $\mathsf{snd}(D) \Downarrow_\Sigma M'$. We have $\Sigma \vdash Q' = P'$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q'\})$.

  By Property S2, there exist D_1 and R_1 such that $\Sigma \vdash D_1 = D$, $\Sigma \vdash R_1 = R$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{R_1, D_1, Q'\})$. Then, there exists no M such that $\mathsf{fst}(D_1) \Downarrow_\Sigma M$, and there exists no M' such that $\mathsf{snd}(D_1) \Downarrow_\Sigma M'$. (Otherwise, by Lemma 18, there would exist M such that $\mathsf{fst}(D) \Downarrow_\Sigma M$, and M' such that $\mathsf{snd}(D) \Downarrow_\Sigma M'$.) Let $Q = let\ x = D_1\ in\ R_1\ else\ Q'$. Then $\Sigma \vdash Q = P$, $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q\})$, and $Q \rightarrow_{\Sigma',\Sigma} Q'$.

- Case (Red Repl): We have $P = !R \rightarrow_\Sigma R \mid !R = P'$. Since $\Sigma \vdash Q' = P'$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q'\})$, we have $Q' = R_1 \mid !R_1$ for some R_1 such that $\Sigma \vdash R_1 = R$. Let $Q = !R_1$. Then we have $\Sigma \vdash Q = P$, $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{P} \cup \{Q\})$, and $Q \rightarrow_{\Sigma',\Sigma} Q'$.

- Cases (Red Par) and (Red Res): Easy by induction hypothesis.

- Case (Red $\equiv$): Easy using the corresponding property for $\equiv$ and the induction hypothesis.

Therefore, if $P_0 \rightarrow_\Sigma^* \equiv P_0'$, $\Sigma \vdash Q_0' = P_0'$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{Q_0'\})$, then there exists Q_0 such that $\mathrm{nf}_{\mathcal{S},\Sigma}(\{Q_0\})$, $\Sigma \vdash Q_0 = P_0$, and $Q_0 \rightarrow_{\Sigma',\Sigma}^* \equiv Q_0'$ by a reduction whose intermediate biprocesses Q all satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}(\{Q\})$, simply by applying several times the results shown above. Since the only process P such that $\Sigma \vdash P_0 = P$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{P\})$ is P_0, we have $Q_0 = P_0$, so we conclude that if $P_0 \rightarrow_\Sigma^* \equiv P_0'$, $\Sigma \vdash Q_0' = P_0'$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{Q_0'\})$, then $P_0 \rightarrow_{\Sigma',\Sigma}^* \equiv Q_0'$ by a reduction whose intermediate biprocesses Q all satisfy $\mathrm{nf}_{\mathcal{S},\Sigma}(\{Q\})$.

For the converse, we show that, if $P \equiv P'$ and $\Sigma \vdash Q = P$, then there exists Q' such that $\Sigma \vdash Q' = P'$ and $Q \equiv Q'$. The proof is by induction on the derivation of $P \equiv P'$. All cases are easy, since $\equiv$ does not depend on terms.

We also show that, if $VC(P)$, $P \rightarrow_{\Sigma',\Sigma} P'$ and $\Sigma \vdash Q = P$, then there exists Q' such that $\Sigma \vdash Q' = P'$, and $Q \rightarrow_\Sigma Q'$. The proof is by induction on the derivation of $P \rightarrow_{\Sigma',\Sigma} P'$.

- Case (Red I/O): Since $VC(P)$, we have $P = \overline{\mathsf{diff}[M_1, M_2]}\langle N\rangle.R \mid \mathsf{diff}[M_1, M_2](x).R' \rightarrow_{\Sigma',\Sigma} R \mid R'\{N/x\} = P'$. Since $\Sigma \vdash Q = P$, we have $Q = \overline{\mathsf{diff}[M_1', M_2']}\langle N'\rangle.R_1 \mid \mathsf{diff}[M_1'', M_2''](x).R_1'$ with $\Sigma \vdash M_1 = M_1' = M_1''$, $\Sigma \vdash M_2 = M_2' = M_2''$, $\Sigma \vdash N' = N$, $\Sigma \vdash R = R_1$, and $\Sigma \vdash R' = R_1'$. Then $Q \rightarrow_\Sigma Q' = R_1 \mid R_1'\{N'/x\}$ with $\Sigma \vdash Q' = P'$.

- Case (Red Fun 1): We have $P = let\ x = D\ in\ R\ else\ R' \rightarrow_{\Sigma',\Sigma} R\{\mathsf{diff}[M_1, M_2]/x\} = P'$ with $\mathsf{fst}(D) \Downarrow_{\Sigma'} M_1$ and $\mathsf{snd}(D) \Downarrow_{\Sigma'} M_2$. Since $\Sigma \vdash Q = P$, we have $Q = let\ x = D'\ in\ R_1\ else\ R_1'$ with $\Sigma \vdash D' = D$, $\Sigma \vdash R_1 = R$, and $\Sigma \vdash R_1' = R'$. By Lemma 17, $\mathsf{fst}(D') \Downarrow_\Sigma M_1'$ with $\Sigma \vdash M_1 = M_1'$ and $\mathsf{snd}(D') \Downarrow_\Sigma M_2'$ with $\Sigma \vdash M_2 = M_2'$. Hence $Q \rightarrow_\Sigma Q' = C'[R_1\{\mathsf{diff}[M_1', M_2']/x\}]$ with $\Sigma \vdash Q' = P'$.

- Case (Red Fun 2): We have $P = let\ x = D\ in\ R\ else\ P' \rightarrow_{\Sigma',\Sigma} P'$, there exists no M_1 such that $\mathsf{fst}(D) \Downarrow_\Sigma M_1$, and there exists no M_2 such that $\mathsf{snd}(D) \Downarrow_\Sigma M_2$. Since $\Sigma \vdash Q = P$, we have $Q = let\ x = D'\ in\ R_1\ else\ Q'$ with $\Sigma \vdash D' = D$, $\Sigma \vdash R_1 = R$, and $\Sigma \vdash Q' = P'$. Then, there exists no M_1' such that $\mathsf{fst}(D') \Downarrow_\Sigma M_1'$, and there exists no M_2' such that $\mathsf{snd}(D') \Downarrow_\Sigma M_2'$. (Otherwise, by Lemma 18, there would exist M_1 such that $\mathsf{fst}(D) \Downarrow_\Sigma M_1$ and M_2 such that $\mathsf{snd}(D) \Downarrow_\Sigma M_2$.) Hence $Q \rightarrow_\Sigma Q'$ and $\Sigma \vdash Q' = P'$.

- Case (Red Repl): We have $P = !R \rightarrow_{\Sigma',\Sigma} R \mid !R = P'$. Since $\Sigma \vdash Q = P$, we have $Q = !R_1$ with $\Sigma \vdash R_1 = R$. Let $Q' = R_1 \mid !R_1$. So $\Sigma \vdash Q' = P'$ and $Q \rightarrow_\Sigma Q'$.
- Cases (Red Par) and (Red Res): Easy by induction hypothesis.
- Case (Red $\equiv$): Easy using the corresponding property for $\equiv$ and the induction hypothesis.

We conclude that, if $P_0 \rightarrow^*_{\Sigma',\Sigma}\equiv Q'_0$ then there exists P'_0 such that $\Sigma \vdash Q'_0 = P'_0$ and $P_0 \rightarrow^*_\Sigma\equiv P'_0$, simply by applying several times the results shown above, with $Q = P$ in the first application. □

B.3 Proof of Lemma 2

We first show that it is enough to consider unevaluated processes as initial configurations (Lemma 22), then prove Lemma 2 itself.

Let $P \mathrel{\mathcal{R}} P'$ if and only if P' is obtained from P by adding some lets on terms with constructors that occur in inputs or outputs (for instance transforming $\overline{M}\langle N\rangle.P$ into *let* $x = M$ *in let* $y = N$ *in* $\overline{x}\langle y\rangle.P$ where x and y are fresh variables), prefixing some constructors in lets with eval, and replacing some terms M with $\mathsf{diff}[\mathsf{fst}(M), \mathsf{snd}(M)]$.

For the next two proofs, we consider an alternative, equivalent definition of $\equiv$, in which a symmetric rule $Q \equiv P$ is added for each rule $P \equiv Q$ in the definition of $\equiv$ and the implication $P \equiv Q \Rightarrow Q \equiv P$ is removed from the definition of $\equiv$.

Lemma 20 *If $P \mathrel{\mathcal{R}} Q$ and $P \equiv P'$ then there exists Q' such that $P' \mathrel{\mathcal{R}} Q'$ and $Q \equiv Q'$.*
If $P \mathrel{\mathcal{R}} Q$ and $P \rightarrow_\Sigma P'$ then there exists Q' such that $P' \mathrel{\mathcal{R}} Q'$ and $Q \rightarrow^+_\Sigma Q'$.

Proof Obvious, by induction on the derivation of $P \equiv P'$ and $P \rightarrow_\Sigma P'$ respectively. □

Lemma 21 *If $\Sigma \vdash P = Q$, $Q \mathrel{\mathcal{R}} R$, and $R \equiv R'$ then there exists P' and Q' such that $\Sigma \vdash P' = Q'$, $Q' \mathrel{\mathcal{R}} R'$, and $P \equiv P'$.*
If $\Sigma \vdash P = Q$, $Q \mathrel{\mathcal{R}} R$, and $R \rightarrow_\Sigma R'$ then there exists P' and Q' such that $\Sigma \vdash P' = Q'$, $Q' \mathrel{\mathcal{R}} R'$, and $P \rightarrow_\Sigma P'$ or $P = P'$.

Proof Obvious, by induction on the derivation of $R \equiv R'$ and $R \rightarrow_\Sigma R'$ respectively. □

Lemma 22 *Let P_0 be a closed biprocess. The hypotheses of Corollary 1 are true if and only if they are true with* unevaluated$(C[P_0])$ *instead of* $C[P_0]$.

Proof We have $C[P_0] \mathrel{\mathcal{R}}$ unevaluated$(C[P_0])$. We first show that if the hypotheses of Corollary 1 are true for unevaluated$(C[P_0])$, then they are true for $C[P_0]$.

- If $C[P_0] \rightarrow^*_\Sigma\equiv C'_1[\overline{N_1}\langle M_1\rangle.Q_1 \mid N'_1(x).R_1]$, then by Lemma 20, we have unevaluated$(C[P_0]) \rightarrow^*_\Sigma\equiv P'$ with $C'_1[\overline{N_1}\langle M_1\rangle.Q_1 \mid N'_1(x).R_1] \mathrel{\mathcal{R}} P'$. Then we have $P' \rightarrow^*_\Sigma C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ with $C'_1 \mathrel{\mathcal{R}} C'$, $\mathsf{fst}(N) = \mathsf{fst}(N_1)$, $\mathsf{snd}(N) = \mathsf{snd}(N_1)$, $\mathsf{fst}(N') = \mathsf{fst}(N'_1)$, $\mathsf{snd}(N') = \mathsf{snd}(N'_1)$, $\mathsf{fst}(M) = \mathsf{fst}(M_1)$, $\mathsf{snd}(M) = \mathsf{snd}(M_1)$, $Q_1 \mathrel{\mathcal{R}} Q$, and $R_1 \mathrel{\mathcal{R}} R$, by reducing the term evaluations of constructors that may occur above inputs and outputs in P'. So unevaluated$(C[P_0]) \rightarrow^*_\Sigma\equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$, with $\mathsf{fst}(N) = \mathsf{fst}(N_1)$, $\mathsf{snd}(N) = \mathsf{snd}(N_1)$, $\mathsf{fst}(N') = \mathsf{fst}(N'_1)$, and $\mathsf{snd}(N') = \mathsf{snd}(N'_1)$. Hence, if the first hypothesis of Corollary 1 is true with unevaluated$(C[P_0])$, then it is true with $C[P_0]$.
- If $C[P_0] \rightarrow^*_\Sigma\equiv C'_1[$*let* $y_1 = D_1$ *in* Q_1 *else* $R_1]$, then by the same reasoning as above, unevaluated$(C[P_0]) \rightarrow^*_\Sigma\equiv C'[P]$ where *let* $y_1 = D_1$ *in* Q_1 *else* $R_1 \mathrel{\mathcal{R}} P$. Hence, we have $P =$ *let* $y_1 = D'_1$ *in* Q'_1 *else* R'_1 where D'_1 is obtained by prefixing some constructors of D_1

with eval and reorganizing diffs. We have $\mathsf{fst}(D_1) \Downarrow_\Sigma M_1$ if and only if $\mathsf{fst}(D'_1) \Downarrow_\Sigma M_1$, if and only if $\mathsf{snd}(D'_1) \Downarrow_\Sigma M_2$ (by the second hypothesis of Corollary 1 for $\mathrm{unevaluated}(C[P_0])$), if and only if $\mathsf{snd}(D_1) \Downarrow_\Sigma M_2$. This yields the second hypothesis of Corollary 1 for $C[P_0]$.

We now show the converse: if the hypotheses of Corollary 1 are true for $C[P_0]$, then they are true for $\mathrm{unevaluated}(C[P_0])$.

- Assume that $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_\Sigma\equiv C'_1[\overline{N_1}\langle M_1\rangle.Q_1 \mid N'_1(x).R_1]$. By Lemma 21, $C[P_0] \rightarrow^*_\Sigma\equiv P$ with $\Sigma \vdash P = P'$ and $P' \mathrel{\mathcal{R}} C'_1[\overline{N_1}\langle M_1\rangle.Q_1 \mid N'_1(x).R_1]$. Then $P = C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ with $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N_1)$, $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N_1)$, $\Sigma \vdash \mathsf{fst}(N') = \mathsf{fst}(N'_1)$, $\Sigma \vdash \mathsf{snd}(N') = \mathsf{snd}(N'_1)$, $\Sigma \vdash \mathsf{fst}(M) = \mathsf{fst}(M_1)$, and $\Sigma \vdash \mathsf{snd}(M) = \mathsf{snd}(M_1)$. So, if the first hypothesis of Corollary 1 is true with $C[P_0]$, then it is true with $\mathrm{unevaluated}(C[P_0])$.
- Assume that $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_\Sigma\equiv C'_1[\mathit{let}\ y_1 = D_1\ \mathit{in}\ Q_1\ \mathit{else}\ R_1]$. By Lemma 21, $C[P_0] \rightarrow^*_\Sigma\equiv P$ with $\Sigma \vdash P = P'$ and $P' \mathrel{\mathcal{R}} C'_1[\mathit{let}\ y_1 = D_1\ \mathit{in}\ Q_1\ \mathit{else}\ R_1]$. We have two cases:
  - Case 1: *let* y_1 is introduced by $\mathcal{R}$. Then $R_1 = 0$ and D_1 does not contain destructors. Hence there exists M_1 such that $\mathsf{fst}(D_1) \Downarrow_\Sigma M_1$ and there exists M_2 such that $\mathsf{snd}(D_1) \Downarrow_\Sigma M_2$.
  - Case 2: *let* y_1 comes from P'. Hence $P = C''[\mathit{let}\ y_1 = D'_1\ \mathit{in}\ Q'_1\ \mathit{else}\ R'_1]$ where D'_1 is obtained by removing some eval prefixes of D_1, reorganizing diffs, and replacing terms with equal terms modulo Σ. We have $\mathsf{fst}(D_1) \Downarrow_\Sigma M_1$ for some M_1 if and only if $\mathsf{fst}(D'_1) \Downarrow_\Sigma M_1$ for some M_1, if and only if $\mathsf{snd}(D'_1) \Downarrow_\Sigma M_2$ for some M_2 (by the second hypothesis of Corollary 1 for $C[P_0]$), if and only if $\mathsf{snd}(D_1) \Downarrow_\Sigma M_2$ for some M_2.

  This yields the second hypothesis of Corollary 1 for $\mathrm{unevaluated}(C[P_0])$. □

Lemma 2 is an obvious consequence of the following lemma.

Lemma 23 *Let P_0 be a closed biprocess. Suppose that, for all plain evaluation contexts C, all evaluation contexts C', and all reductions* $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_{\Sigma',\Sigma} P$ *whose intermediate biprocesses P' all satisfy* $\mathrm{nf}_{\mathcal{S},\Sigma}(\{P'\})$*,*

1. *if $P \equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\mathsf{fst}(N) = \mathsf{fst}(N')$, then $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$,*
2. *if $P \equiv C'[\mathit{let}\ x = D\ \mathit{in}\ Q\ \mathit{else}\ R]$ and $\mathsf{fst}(D) \Downarrow_{\Sigma'} M_1$ for some M_1, then $\mathsf{snd}(D) \Downarrow_\Sigma M_2$ for some M_2,*

as well as the symmetric properties where we swap fst *and* snd*. Then P_0 satisfies the hypotheses of Corollary 1.*

Conversely, if P_0 satisfies the hypotheses of Corollary 1, then for all plain evaluation contexts C, evaluation contexts C', and reductions $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_{\Sigma'} P$*, we have properties 1 and 2 above, as well as the symmetric properties where we swap* fst *and* snd*.*

Proof By Lemma 22, we can work with $\mathrm{unevaluated}(C[P_0])$ instead of $C[P_0]$. We show the two hypotheses of Corollary 1.

- Assume that $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_\Sigma\equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\Sigma \vdash \mathsf{fst}(N) = \mathsf{fst}(N')$. By Property S2, there exists P' such that $\Sigma \vdash P' = C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{P'\})$. By Lemma 19, $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv P'$. Moreover, $P' = C''[\overline{\mathsf{diff}[N_1, N_2]}\langle M'\rangle.Q_1 \mid \mathsf{diff}[N'_1, N'_2](x).R_1]$, where $\Sigma \vdash N_1 = \mathsf{fst}(N)$, $\Sigma \vdash N_2 = \mathsf{snd}(N)$, $\Sigma \vdash N'_1 = \mathsf{fst}(N')$, and $\Sigma \vdash N'_2 = \mathsf{snd}(N')$. Since $\mathrm{nf}_{\mathcal{S},\Sigma}(\{P'\})$, $N_1 = N'_1$. Hence, by hypothesis 1, $\Sigma \vdash N_2 = N'_2$. So $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$.

  We obtain the case $\mathrm{unevaluated}(C[P_0]) \rightarrow^*\equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$ by symmetry.

- Assume that unevaluated$(C[P_0]) \rightarrow^*_{\Sigma}\equiv C'[\mathit{let}\ y = D\ \mathit{in}\ Q\ \mathit{else}\ R]$ and there exists M_1 such that $\mathsf{fst}(D) \Downarrow_{\Sigma} M_1$. By Property S2, there exist P', M'_1, and D' such that $\Sigma \vdash P' = C'[\mathit{let}\ y = D\ \mathit{in}\ Q\ \mathit{else}\ R]$, $\Sigma \vdash M'_1 = M_1$, $\Sigma \vdash D' = D$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{P', M'_1, D'\})$. Then $P' = C''[\mathit{let}\ y = D'\ \mathit{in}\ Q'\ \mathit{else}\ R']$. By Lemma 19, unevaluated$(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv P'$. By Lemma 14, $\mathsf{fst}(D') \Downarrow_{\Sigma'} M'_1$. By hypothesis 2, $\mathsf{snd}(D') \Downarrow_{\Sigma} M'_2$. By Lemma 18, since $\Sigma \vdash \mathsf{snd}(D') = \mathsf{snd}(D)$ and $\mathsf{snd}(D') \Downarrow_{\Sigma} M'_2$, we have $\mathsf{snd}(D) \Downarrow_{\Sigma} M_2$.

  We obtain the case unevaluated$(C[P_0]) \rightarrow^*\equiv C'[\mathit{let}\ y = D\ \mathit{in}\ Q\ \mathit{else}\ R]$ and there exists M_2 such that $\mathsf{snd}(D) \Downarrow_{\Sigma} M_2$ by symmetry.

Next, we prove the converse property.

- Assume that unevaluated$(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\mathsf{fst}(N) = \mathsf{fst}(N')$. By Lemma 19, we have unevaluated$(C[P_0]) \rightarrow^*_{\Sigma}\equiv C_1[\overline{N_1}\langle M_1\rangle.Q_1 \mid N_1(x).R_1]$ with $\Sigma \vdash C'[\overline{N}\langle M\rangle.Q \mid N'(x).R] = C_1[\overline{N_1}\langle M_1\rangle.Q_1 \mid N_1(x).R_1]$ so $\Sigma \vdash N = N_1$ and $\Sigma \vdash N' = N'_1$. Using the first hypothesis of Corollary 1, since $\Sigma \vdash \mathsf{fst}(N_1) = \mathsf{fst}(N'_1)$, we have $\Sigma \vdash \mathsf{snd}(N_1) = \mathsf{snd}(N'_1)$, hence $\Sigma \vdash \mathsf{snd}(N) = \mathsf{snd}(N')$.

  We obtain the case unevaluated$(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv C'[\overline{N}\langle M\rangle.Q \mid N'(x).R]$ and $\mathsf{snd}(N) = \mathsf{snd}(N')$ by symmetry.

- Assume that unevaluated$(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv C'[\mathit{let}\ y = D\ \mathit{in}\ Q\ \mathit{else}\ R]$ and there exists M_1 such that $\mathsf{fst}(D) \Downarrow_{\Sigma'} M_1$. As above, unevaluated$(C[P_0]) \rightarrow^*_{\Sigma}\equiv C_1[\mathit{let}\ y = D_1\ \mathit{in}\ Q_1\ \mathit{else}\ R_1]$ with $\Sigma \vdash D_1 = D$. By Lemma 17, $\mathsf{fst}(D_1) \Downarrow_{\Sigma} M'_1$ for some M'_1. Using the second hypothesis of Corollary 1, $\mathsf{snd}(D_1) \Downarrow_{\Sigma} M'_2$, hence by Lemma 18, $\mathsf{snd}(D) \Downarrow_{\Sigma} M_2$.

  We obtain the case unevaluated$(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv C'[\mathit{let}\ y = D\ \mathit{in}\ Q\ \mathit{else}\ R]$ and there exists M_2 such that $\mathsf{snd}(D) \Downarrow_{\Sigma'} M_2$ by symmetry. □

C Proof of Lemma 3

When $\mathcal{F}$ is a set that contains patterns, facts, sequences of patterns or facts, clauses, environments that map variables and names to pairs of patterns, ..., we say that $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{F})$ if and only if all patterns that appear in $\mathcal{F}$ are irreducible by $\mathcal{S}$ and for all p_1, p_2 subpatterns of elements of $\mathcal{F}$, if $\Sigma \vdash p_1 = p_2$ then $p_1 = p_2$.

We say that $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{F})$ if and only if $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{F}')$ where $\mathcal{F}'$ is obtained from $\mathcal{F}$ by removing nounif facts. When $\mathcal{D}$ is a derivation, we say that $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$ when $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{F})$ where $\mathcal{F}$ is the set of intermediately derived facts of $\mathcal{D}$.

We say that $F_1 \wedge \cdots \wedge F_n \sim F'_1 \wedge \cdots \wedge F'_n$ when, for all $i \in \{1, \ldots, n\}$, either $F_i = F'_i$ or F_i and F'_i are nounif facts and $\Sigma \vdash F_i = F'_i$. We say that $\Sigma \vdash F_1 \wedge \cdots \wedge F_n \sim F'_1 \wedge \cdots \wedge F'_n$ when for all $i \in \{1, \ldots, n\}$, $\Sigma \vdash F_i = F'_i$. This definition is naturally extended to clauses.

The special treatment of nounif facts in the definition of $\sim$ and in Lemma 3 is necessary so that the following results hold. In particular, Lemma 28 would be wrong for Clauses (Rt) and (Rt′), which contain nounif facts.

Lemma 24 *If $h(N_1, \ldots, N_n) \rightarrow N$ is in $\mathrm{def}_{\Sigma'}(h)$, $\Sigma \vdash N'' = \sigma N$, $\Sigma \vdash N''_i = \sigma N_i$ for all $i \in \{1, \ldots, n\}$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{N''_1, \ldots, N''_n, N''\})$, then there exist a closed substitution σ' and $h(N'_1, \ldots, N'_n) \rightarrow N'$ in $\mathrm{def}_{\Sigma'}(h)$ such that $N'' = \sigma' N'$ and $N''_i = \sigma' N'_i$ for all $i \in \{1, \ldots, n\}$.*

Proof The result follows from Lemmas 16 and 13. □

The following lemma generalizes Lemma 15 to the case in which D may contain destructors. It is used in the proof of Lemma 26 below.

Lemma 25 *Let D be a plain term evaluation. If $D \Downarrow' (p,\sigma)$ and σ' is a closed substitution, then there exists p' such that $\sigma'\sigma D \Downarrow_\Sigma p'$ and $\Sigma \vdash p' = \sigma' p$.*

Let $D_1,\dots,D_n$ be plain term evaluations. If $(D_1,\dots,D_n) \Downarrow' ((p_1,\dots,p_n),\sigma)$ and σ' is a closed substitution then there exist $p'_1,\dots,p'_n$ such that for all $i \in \{1,\dots,n\}$, $\sigma'\sigma D_i \Downarrow_\Sigma p'_i$ and $\Sigma \vdash p'_i = \sigma' p_i$.

Proof The proof is by mutual induction following the definition of $\Downarrow'$.

- Case $D = p$: We have $p \Downarrow' (p,\emptyset)$, $\sigma = \emptyset$, so $\sigma'\sigma D = \sigma' p \Downarrow_\Sigma \sigma' p$, so we have the result with $p' = \sigma' p$.

- Case $D = \text{eval } h(D_1,\dots,D_n)$: Since $\text{eval } h(D_1,\dots,D_n) \Downarrow' (p,\sigma)$, there exist $h(N_1,\dots,N_n) \to N$ in $\text{def}_{\Sigma'}(h)$, $p_1,\dots,p_n$, σ'', and σ_u such that $(D_1,\dots,D_n) \Downarrow' ((p_1,\dots,p_n),\sigma'')$, σ_u is a most general unifier of $(p_1,\dots,p_n)$ and $(N_1,\dots,N_n)$, $p = \sigma_u N$, and $\sigma = \sigma_u\sigma''$. By induction hypothesis, there exist $p'_1,\dots,p'_n$ such that for all $i \in \{1,\dots,n\}$, $\sigma'\sigma_u\sigma'' D_i \Downarrow_\Sigma p'_i$ and $\Sigma \vdash p'_i = \sigma'\sigma_u p_i$, so $\sigma'\sigma D_i \Downarrow_\Sigma p'_i$. By Lemma 16, there exist $h(N'_1,\dots,N'_n) \to N'$ in $\text{def}_\Sigma(h)$ and σ_1 such that $\Sigma \vdash N_i = \sigma_1 N'_i$ for all $i \in \{1,\dots,n\}$ and $\Sigma \vdash N = \sigma_1 N'$. So $\Sigma \vdash p'_i = \sigma'\sigma_u p_i = \sigma'\sigma_u N_i = \sigma'\sigma_u\sigma_1 N'_i$ and $\Sigma \vdash \sigma' p = \sigma'\sigma_u N = \sigma'\sigma_u\sigma_1 N'$. Let $p' = \sigma'\sigma_u\sigma_1 N'$. We have $\sigma'\sigma D \Downarrow_\Sigma p'$ and $\Sigma \vdash p' = \sigma' p$.

- Case $(D_1,\dots,D_n)$: Since $(D_1,\dots,D_n) \Downarrow' ((p_1,\dots,p_n),\sigma)$, we have $(D_1,\dots,D_{n-1}) \Downarrow' ((p''_1,\dots,p''_{n-1}),\sigma_1)$, $\sigma_1 D_n \Downarrow' (p_n,\sigma_2)$, $p_i = \sigma_2 p''_i$ for all $i \in \{1,\dots,n-1\}$, and $\sigma = \sigma_2\sigma_1$. By induction hypothesis, there exist $p'_1,\dots,p'_{n-1}$ such that for all $i \in \{1,\dots,n-1\}$, $\sigma'\sigma_2\sigma_1 D_i \Downarrow_{\Sigma'} p'_i$ and $\Sigma \vdash p'_i = \sigma'\sigma_2 p''_i$, so $\sigma'\sigma D_i \Downarrow_{\Sigma'} p'_i$ and $\Sigma \vdash p'_i = \sigma' p_i$. Also by induction hypothesis, there exists p'_n such that $\sigma'\sigma_2\sigma_1 D_n \Downarrow_{\Sigma'} p'_n$ and $\Sigma \vdash p'_n = \sigma' p_n$, so $\sigma'\sigma D_n \Downarrow_{\Sigma'} p'_n$ and $\Sigma \vdash p'_n = \sigma' p_n$. □

Lemma 26 *Let D be a plain term evaluation such that the subterms M of D are variables or names. If $\rho(D) \Downarrow' (p',\sigma')$, σ is a closed substitution, $\Sigma \vdash p = \sigma p'$, $\Sigma \vdash \sigma'_0\rho' = \sigma\sigma'\rho$, and $\text{nf}_{\mathcal{S},\Sigma}(\{p,\sigma'_0\rho'\})$, then there exist σ'', p'', σ''_0 such that $\rho'(D) \Downarrow' (p'',\sigma'')$, $\sigma'_0 = \sigma''_0\sigma''$ except on fresh variables introduced in the computation of $\rho'(D) \Downarrow' (p'',\sigma'')$, and $p = \sigma''_0 p''$.*

Let D_i ($i \in \{1,\dots,n\}$) be plain term evaluations such that the subterms M of D_i are variables or names. If $(\rho(D_1),\dots,\rho(D_n)) \Downarrow' ((p'_1,\dots,p'_n),\sigma')$, σ is a closed substitution, $\Sigma \vdash p_i = \sigma p'_i$ for all $i \in \{1,\dots,n\}$, $\Sigma \vdash \sigma'_0\rho' = \sigma\sigma'\rho$, and $\text{nf}_{\mathcal{S},\Sigma}(\{p_1,\dots,p_n,\sigma'_0\rho'\})$, then there exist σ'', $p''_1,\dots,p''_n$, σ''_0 such that $(\rho'(D_1),\dots,\rho'(D_n)) \Downarrow' ((p''_1,\dots,p''_n),\sigma'')$, $\sigma'_0 = \sigma''_0\sigma''$ except on fresh variables introduced in the computation of $(\rho'(D_1),\dots,\rho'(D_n)) \Downarrow' ((p''_1,\dots,p''_n),\sigma'')$, and $p_i = \sigma''_0 p''_i$ for all $i \in \{1,\dots,n\}$.

Proof We prove the first property. (The second one follows in a similar way.) By Lemma 25, there exists p_1 such that $\sigma\sigma'\rho(D) \Downarrow_\Sigma p_1$ and $\Sigma \vdash p_1 = \sigma p'$. Then $\Sigma \vdash p = p_1$, $\Sigma \vdash \sigma'_0\rho'(D) = \sigma\sigma'\rho(D)$, and $\text{nf}_{\mathcal{S},\Sigma}(\{p,\sigma'_0\rho'(D)\})$. So by a variant of Lemma 14 for patterns instead of terms, $\sigma'_0\rho'(D) \Downarrow_{\Sigma'} p$. By a variant of Lemma 11 for patterns instead of terms, we obtain the desired result. □

Lemma 27 *Let P_0 be a closed, unevaluated process. If $[\![P]\!]\rho s s' H$ is called during the generation of $[\![P_0]\!]\rho_0\emptyset\emptyset\emptyset$, σ is a closed substitution, $\Sigma \vdash \rho_2 = \sigma\rho$, $\Sigma \vdash s_2 = \sigma s$, $\Sigma \vdash s'_2 = \sigma s'$, $\Sigma \vdash H_2 \sim \sigma H$, and $\text{nf}'_{\mathcal{S},\Sigma}(\{\rho_2, s_2, s'_2, H_2\})$, then there exist σ_1, ρ_1, H_1, s_1, s'_1 such that $\rho_2 = \sigma_1\rho_1$, $s_2 = \sigma_1 s_1$, $s'_2 = \sigma_1 s'_1$, $H_2 \sim \sigma_1 H_1$, and $[\![P]\!]\rho_1 s_1 s'_1 H_1$ is called during the generation of $[\![P_0]\!]\rho_0\emptyset\emptyset\emptyset$.*

Proof The process P is a subprocess of P_0. We proceed by induction on P: we show the result for P_0 itself, and we show that if the result is true for some occurrence of P, then it is also true for the occurrences of the direct subprocesses of P.

- Case P_0: We have $\rho_2 = \rho_0$, $s_2 = s'_2 = \emptyset$, and $H_2 = \emptyset$. Then we obtain the result by letting σ_1 be any substitution, $\rho_1 = \rho_0$, $s_1 = s'_1 = \emptyset$, and $H_1 = \emptyset$.

- Case 0: Void, since it has no subprocesses.

- Case $P \mid Q$: Obvious by induction hypothesis.

- Case $!P$: Assume $[\![P]\!]\rho s s' H$ is called. Then $\rho = \rho_3$, $s = (s_3, i)$, $s' = (s'_3, i)$, $H = H_3$, and $[\![!P]\!]\rho_3 s_3 s'_3 H_3$ has been called. Let ρ_2, s_2, s'_2, H_2 such that $\Sigma \vdash \rho_2 = \sigma\rho$, $\Sigma \vdash s_2 = \sigma s$, $\Sigma \vdash s'_2 = \sigma s'$, $\Sigma \vdash H_2 \sim \sigma H$, and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_2, s_2, s'_2, H_2\})$.

  Then $\rho_2 = \rho_4$, $s_2 = (s_4, p)$, $s'_2 = (s'_4, p)$, $H_2 = H_4$ where $\Sigma \vdash \rho_4 = \sigma\rho_3$, $\Sigma \vdash s_4 = \sigma s_3$, $\Sigma \vdash s'_4 = \sigma s'_3$, $\Sigma \vdash H_4 \sim \sigma H_3$, and $\Sigma \vdash p = \sigma i$.

  By induction hypothesis, there exist σ_1, ρ_5, s_5, s'_5, H_5 such that $\rho_4 = \sigma_1\rho_5$, $s_4 = \sigma_1 s_5$, $s'_4 = \sigma_1 s'_5$, $H_4 \sim \sigma_1 H_5$, and $[\![!P]\!]\rho_5 s_5 s'_5 H_5$ has been called. Since i is a fresh variable, we can define $\sigma_1 i = p$.

  Then $[\![P]\!]\rho_5(s_5, i)(s'_5, i)H_5$ has been called, $\rho_2 = \sigma_1\rho_5$, $s_2 = \sigma_1(s_5, i)$, $s'_2 = \sigma_1(s'_5, i)$, and $H_2 \sim \sigma_1 H_5$.

- Case $(\nu a)P$: Assume $[\![P]\!]\rho s s' H$ is called. Then $\rho = \rho_3[a \mapsto (a[s], a[s'])]$ and $[\![(\nu a)P]\!]\rho_3 s s' H$ has been called. Let ρ_2, s_2, s'_2, H_2 such that $\Sigma \vdash \rho_2 = \sigma\rho$, $\Sigma \vdash s_2 = \sigma s$, $\Sigma \vdash s'_2 = \sigma s'$, $\Sigma \vdash H_2 \sim \sigma H$, and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_2, s_2, s'_2, H_2\})$.

  Then $\rho_2 = \rho_4[a \mapsto (a[s_2], a[s'_2])]$ where $\Sigma \vdash \rho_4 = \sigma\rho_3$.

  By induction hypothesis, there exist σ_1, ρ_5, s_1, s'_1, H_1 such that $\rho_4 = \sigma_1\rho_5$, $s_2 = \sigma_1 s_1$, $s'_2 = \sigma_1 s'_1$, $H_2 \sim \sigma_1 H_1$, and $[\![(\nu a)P]\!]\rho_5 s_1 s'_1 H_1$ has been called.

  Then $[\![P]\!](\rho_5[a \mapsto (a[s_1], a[s'_1])])s_1 s'_1 H_1$ has been called, $\rho_2 = \sigma_1(\rho_5[a \mapsto (a[s_1], a[s'_1])])$, $s_2 = \sigma_1 s_1$, $s'_2 = \sigma_1 s'_1$, and $H_2 \sim \sigma_1 H_1$.

- Case $\overline{M}\langle N\rangle.P$: Obvious by induction hypothesis.

- Case $M(x).P$: Assume $[\![P]\!]\rho s s' H$ is called. Then $\rho = \rho_3[x \mapsto (x', x'')]$, $s = (s_3, x')$, $s' = (s'_3, x'')$, $H = H_3 \wedge \mathrm{msg}'(\rho_3(M)_1, x', \rho_3(M)_2, x'')$, and $[\![M(x).P]\!]\rho_3 s_3 s'_3 H_3$ has been called. Let ρ_2, s_2, s'_2, H_2 such that $\Sigma \vdash \rho_2 = \sigma\rho$, $\Sigma \vdash s_2 = \sigma s$, $\Sigma \vdash s'_2 = \sigma s'$, $\Sigma \vdash H_2 \sim \sigma H$, and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_2, s_2, s'_2, H_2\})$.

  Then $\rho_2 = \rho_4[x \mapsto (p', p'')]$, $s_2 = (s_4, p')$, $s'_2 = (s'_4, p'')$, $H_2 = H_4 \wedge \mathrm{msg}'(\rho_4(M)_1, p', \rho_4(M)_2, p'')$ where $\Sigma \vdash \rho_4 = \sigma\rho_3$, $\Sigma \vdash s_4 = \sigma s_3$, $\Sigma \vdash s'_4 = \sigma s'_3$, $\Sigma \vdash H_4 \sim \sigma H_3$, $\Sigma \vdash p' = \sigma x'$, and $\Sigma \vdash p'' = \sigma x''$. (Since P_0 is unevaluated, M is a variable y or $\mathsf{diff}[a, a]$ for some name a. Let $u = y$ in the first case and $u = a$ in the second case. We have $u \in \mathit{dom}(\rho_3) = \mathit{dom}(\rho_4)$. We have $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_2, s_2, s'_2, H_2\})$ so a fortiori $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_4, H_2\})$, and the first and third arguments of msg' are equal to $\rho_4(M)_1 = \rho_4(u)_1$ and $\rho_4(M)_2 = \rho_4(u)_2$ modulo Σ respectively, so they are exactly $\rho_4(M)_1$ and $\rho_4(M)_2$.)

  By induction hypothesis, there exist σ_1, ρ_5, s_5, s'_5, H_5 such that $\rho_4 = \sigma_1\rho_5$, $s_4 = \sigma_1 s_5$, $s'_4 = \sigma_1 s'_5$, $H_4 \sim \sigma_1 H_5$, and $[\![M(x).P]\!]\rho_5 s_5 s'_5 H_5$ has been called. Since x' and x'' are fresh variables, we can define $\sigma_1 x' = p'$ and $\sigma_1 x'' = p''$.

  Then $[\![P]\!](\rho_5[x \mapsto (x', x'')])(s_5, x')(s'_5, x'')(H_5 \wedge \mathrm{msg}'(\rho_5(M)_1, x', \rho_5(M)_2, x''))$ has been called, and $\rho_2 = \sigma_1(\rho_5[x \mapsto (x', x'')])$, $s_2 = \sigma_1(s_5, x')$, $s'_2 = \sigma_1(s'_5, x'')$, and $H_2 \sim \sigma_1(H_5 \wedge \mathrm{msg}'(\rho_5(M)_1, x', \rho_5(M)_2, x''))$.

- Case *let* $x = D$ *in* P *else* Q:

  Subprocess P: Assume $[\![P]\!]\rho s s' H$ is called. Then we have $\rho = (\sigma_1\rho_3)[x \mapsto (p_1, p'_1)]$, $s = (\sigma_1 s_3, p_1)$, $s' = (\sigma_1 s'_3, p'_1)$, and $H = \sigma_1 H_3$ where $[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho_3 s_3 s'_3 H_3$ has

been called and $(\rho(D)_1, \rho(D)_2) \Downarrow' ((p_1, p'_1), \sigma_1)$. Let ρ_2, s_2, s'_2, H_2 such that $\Sigma \vdash \rho_2 = \sigma\rho$, $\Sigma \vdash s_2 = \sigma s$, $\Sigma \vdash s'_2 = \sigma s'$, $\Sigma \vdash H_2 \sim \sigma H$, and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_2, s_2, s'_2, H_2\})$.

Then $\rho_2 = \rho_4[x \mapsto (p_4, p'_4)]$, $s_2 = (s_4, p_4)$, $s'_2 = (s'_4, p'_4)$, $H_2 = H_4$ with $\Sigma \vdash \rho_4 = \sigma\sigma_1\rho_3$, $\Sigma \vdash s_4 = \sigma\sigma_1 s_3$, $\Sigma \vdash s_4 = \sigma\sigma_1 s'_3$, $\Sigma \vdash H_4 \sim \sigma\sigma_1 H_3$, $\Sigma \vdash p_4 = \sigma p_1$, $\Sigma \vdash p'_4 = \sigma p'_1$, and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_4, s_4, s'_4, H_4, p_4, p'_4\})$.

By induction hypothesis, there exist $\sigma'_0, \rho_5, s_5, s'_5, H_5$ such that $\rho_4 = \sigma'_0\rho_5$, $s_4 = \sigma'_0 s_5$, $s'_4 = \sigma'_0 s'_4$, $H_4 \sim \sigma'_0 H_5$, and $[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho_5 s_5 s'_5 H_5$ has been called.

By Lemma 26, there exist σ_2, p_2, p'_2, and σ_3 such that $(\rho_5(D)_1, \rho_5(D)_2) \Downarrow' ((p_2, p'_2), \sigma_2)$, $\sigma'_0 = \sigma_3\sigma_2$ except on fresh variables introduced in the computation of $(\rho_5(D)_1, \rho_5(D)_2) \Downarrow' ((p_2, p'_2), \sigma_2)$, $p_4 = \sigma_3 p_2$, and $p'_4 = \sigma_3 p'_2$.

Moreover, by definition of $[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]$, $[\![P]\!]((\sigma_2\rho_5)[x \mapsto (p_2, p'_2)])(\sigma_2 s_5, p_2)(\sigma_2 s'_5, p'_2)(\sigma_2 H_5)$ has been called, so we obtain the result by letting $\rho_1 = (\sigma_2\rho_5)[x \mapsto (p_2, p'_2)]$, $s_1 = (\sigma_2 s_5, p_2)$, $s'_1 = (\sigma_2 s'_5, p'_2)$, $H_1 = \sigma_2 H_5$: we have $\rho_2 = \rho_4[x \mapsto (p_4, p'_4)] = (\sigma'_0\rho_5)[x \mapsto (\sigma_3 p_2, \sigma_3 p'_2)] = \sigma_3((\sigma_2\rho_5)[x \mapsto (p_2, p'_2)]) = \sigma_3\rho_1$, and similarly for s_2, s'_2, and H_2.

Subprocess Q: Assume $[\![Q]\!]\rho s s' H$ is called. Then $H = H_3 \wedge \rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \rho(\mathrm{fails}(\mathsf{snd}(D)))_2$ and $[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho s s' H_3$ has been called. Let ρ_2, s_2, s'_2, H_2 such that $\Sigma \vdash \rho_2 = \sigma\rho$, $\Sigma \vdash s_2 = \sigma s$, $\Sigma \vdash s'_2 = \sigma s'$, $\Sigma \vdash H_2 \sim \sigma H$, and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{\rho_2, s_2, s'_2, H_2\})$.

Then $H_2 = H_4 \wedge H_{4\mathrm{nounif}}$ where $H_{4\mathrm{nounif}}$ consists of nounif facts, $\Sigma \vdash H_{4\mathrm{nounif}} \sim \sigma\rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \sigma\rho(\mathrm{fails}(\mathsf{snd}(D)))_2$, and $\Sigma \vdash H_4 \sim \sigma H_3$.

By induction hypothesis, there exist $\sigma_1, \rho_1, s_1, s'_1, H_5$ such that $\rho_2 = \sigma_1\rho_1$, $s_2 = \sigma_1 s_1$, $s'_2 = \sigma_1 s'_1$, $H_4 \sim \sigma_1 H_5$, and $[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho_1 s_1 s'_1 H_5$ has been called.

Then $[\![Q]\!]\rho_1 s_1 s'_1(H_5 \wedge \rho_1(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \rho_1(\mathrm{fails}(\mathsf{snd}(D)))_2)$ has been called, which yields the desired result, knowing that $H_2 = H_4 \wedge H_{4\mathrm{nounif}} \sim \sigma_1 H_5 \wedge \sigma_1\rho_1(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \sigma_1\rho_1(\mathrm{fails}(\mathsf{snd}(D)))_2$, since $\Sigma \vdash \sigma_1\rho_1 = \rho_2 = \sigma\rho$. □

Lemma 28 *Let P_0 be a closed, unevaluated process. For all clauses $H \rightarrow C \in \mathcal{R}_{P_0}$, for all closed substitutions σ, for all $H_2 \rightarrow C_2$ such that $\Sigma \vdash H_2 \rightarrow C_2 \sim \sigma(H \rightarrow C)$ and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{H_2, C_2\})$, there exist a closed substitution σ_1 and a clause $H_1 \rightarrow C_1 \in \mathcal{R}_{P_0}$ such that $H_2 \sim \sigma_1 H_1$ and $C_2 = \sigma_1 C_1$.*

Proof The clauses of $[\![P_0]\!]\rho_0\emptyset\emptyset\emptyset$ are generated from the following cases:

- $H \rightarrow C = H \rightarrow \mathrm{input}'(\rho(M)_1, \rho(M)_2)$ where $[\![M(x).P]\!]\rho s s' H$ has been called during the generation of $[\![P_0]\!]\rho_0\emptyset\emptyset\emptyset$. Since $\Sigma \vdash H_2 \rightarrow C_2 \sim \sigma(H \rightarrow C)$ and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{H_2, C_2\})$, we have $\Sigma \vdash H_2 \sim \sigma H$, $C_2 = \mathrm{input}'(p_2, p'_2)$, $\Sigma \vdash p_2 = \sigma\rho(M)_1$, and $\Sigma \vdash p'_2 = \sigma\rho(M)_2$.

  Since P_0 is unevaluated, M is a variable y or $\mathsf{diff}[a, a]$ for some name a. Let $u = y$ in the first case and $u = a$ in the second case. We have $u \in \mathit{dom}(\rho)$. We define ρ_2 by $\rho_2(u) = \mathsf{diff}[p_2, p'_2]$ and extend ρ_2 to $\mathit{dom}(\rho)$ in such a way that $\Sigma \vdash \rho_2 = \sigma\rho$ and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{H_2, \rho_2\})$ by Property S2. We also define s_2 and s'_2 so that $\Sigma \vdash s_2 = \sigma s$, $\Sigma \vdash s'_2 = \sigma s'$, and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\{H_2, \rho_2, s_2, s'_2\})$ by Property S2. By Lemma 27, there exist σ_1, ρ_1, s_1, s'_1, H_1 such that $\rho_2 = \sigma_1\rho_1$, $s_2 = \sigma_1 s_1$, $s'_2 = \sigma_1 s'_1$, $H_2 \sim \sigma_1 H_1$, and $[\![M(x).P]\!]\rho_1 s_1 s'_1 H_1$ has been called.

  Then $H_1 \rightarrow \mathrm{input}'(\rho_1(M)_1, \rho_1(M)_2)$ is in $[\![P_0]\!]\rho_0\emptyset\emptyset\emptyset$, $H_2 \sim \sigma_1 H_1$, $C_2 = \mathrm{input}'(p_2, p'_2) = \mathrm{input}'(\rho_2(M)_1, \rho_2(M)_2) = \sigma_1\mathrm{input}'(\rho_1(M)_1, \rho_1(M)_2)$.

- $H \rightarrow C = H \rightarrow \mathrm{msg}'(\rho(M)_1, \rho(N)_1, \rho(M)_2, \rho(N)_2)$ where $[\![\overline{M}\langle N\rangle.P]\!]\rho s s' H$ has been called. This case is similar to the previous one. (The terms M and N are variables or $\mathsf{diff}[a, a]$ for some name a.)

- $H \to C = \sigma' H' \wedge \sigma'\rho(\text{fails}(\mathsf{snd}(D)))_2 \to \text{bad}$ where $[\![let\ x = D\ in\ P\ else\ Q]\!]\rho s s' H'$ has been called and $\rho(D)_1 \Downarrow' (p', \sigma')$. Since $\Sigma \vdash H_2 \to C_2 \sim \sigma(H \to C)$ and $\text{nf}'_{\mathcal{S},\Sigma}(\{H_2, C_2\})$, we have $H_2 = H_3 \wedge H_{3\text{nounif}}$ where $\Sigma \vdash H_3 \sim \sigma\sigma' H'$ and $H_{3\text{nounif}}$ consists of nounif facts such that $\Sigma \vdash H_{3\text{nounif}} \sim \sigma\sigma'\rho(\text{fails}(\mathsf{snd}(D)))_2$. By Property S2, there exist ρ_3, s_3, s'_3 such that $\Sigma \vdash \rho_3 = \sigma\sigma'\rho$, $\Sigma \vdash s_3 = \sigma\sigma' s$, $\Sigma \vdash s'_3 = \sigma\sigma' s'$, and $\text{nf}'_{\mathcal{S},\Sigma}(\{\rho_3, s_3, s'_3, H_3\})$.

  By Lemma 27, there exist σ_1, ρ_1, s_1, s'_1, H_1 such that $\rho_3 = \sigma_1\rho_1$, $s_3 = \sigma_1 s_1$, $s'_3 = \sigma_1 s'_1$, $H_3 \sim \sigma_1 H_1$, and $[\![let\ x = D\ in\ P\ else\ Q]\!]\rho_1 s_1 s'_1 H_1$ has been called.

  By Property S2, we can choose p such that $\Sigma \vdash p = \sigma p'$ and $\text{nf}_{\mathcal{S},\Sigma}(\{p, \sigma_1\rho_1\})$. By Lemma 26, there exist σ'_1, p'_1, and σ''_1 such that $\rho_1(D)_1 \Downarrow' (p'_1, \sigma'_1)$ and $\sigma_1 = \sigma''_1\sigma'_1$ except on fresh variables introduced in the computation of $\rho_1(D)_1 \Downarrow' (p'_1, \sigma'_1)$. Then $\sigma'_1 H_1 \wedge \sigma'_1\rho_1(\text{fails}(\mathsf{snd}(D)))_2 \to \text{bad}$ is in $[\![P_0]\!]\rho_0\emptyset\emptyset\emptyset$. Moreover $\sigma''_1(\sigma'_1 H_1 \wedge \sigma'_1\rho_1(\text{fails}(\mathsf{snd}(D)))_2) = \sigma_1 H_1 \wedge \sigma_1\rho_1(\text{fails}(\mathsf{snd}(D)))_2 \sim H_3 \wedge H_{3\text{nounif}} \sim H_2$, since $\Sigma \vdash \sigma_1\rho_1 = \rho_3 = \sigma\sigma'\rho$, and $\sigma''_1\text{bad} = \text{bad} = C$, so we have the desired result.

- $H \to C = \sigma' H' \wedge \sigma'\rho(\text{fails}(\mathsf{fst}(D)))_1 \to \text{bad}$ where $[\![let\ x = D\ in\ P\ else\ Q]\!]\rho s s' H'$ has been called and $\rho(D)_2 \Downarrow' (p', \sigma')$. This case is symmetric from the previous one.

For the other clauses:

- For Clause (Rinit), $C_2 = C$, $H_2 = \emptyset$, so we have the result by taking $H_1 \to C_1 = H \to C$.

- For Clauses (Rn), (Rl), (Rs), (Ri), (Rcom), and (Rcom$'$), $H_2 = \sigma' H$ and $C_2 = \sigma' C$ where for all $x \in fv(H \to C)$, $\Sigma \vdash \sigma' x = \sigma x$, and $\text{nf}_{\mathcal{S},\Sigma}(\{\sigma' x \mid x \in fv(H \to C)\})$. (Indeed, the function symbols in H, C do not appear in equations of Σ.) So we obtain the result by taking $H_1 \to C_1 = H \to C$ and $\sigma_1 = \sigma'$.

- For Clause (Rf), $H = \text{att}'(M_1, N_1) \wedge \ldots \wedge \text{att}'(M_n, N_n)$, $C = \text{att}'(M, N)$, $h(M_1, \ldots, M_n) \to M$ in $\text{def}_{\Sigma'}(h)$, $h(N_1, \ldots, N_n) \to N$ in $\text{def}_{\Sigma'}(h)$, $H_2 = \text{att}'(M''_1, N''_1) \wedge \ldots \wedge \text{att}'(M''_n, N''_n)$, $C_2 = \text{att}'(M'', N'')$ with $\Sigma \vdash M'' = \sigma M$, $\Sigma \vdash N'' = \sigma N$, $\Sigma \vdash M''_i = \sigma M_i$ and $\Sigma \vdash N''_i = \sigma N_i$ for all $i \in \{1, \ldots, n\}$, and $\text{nf}_{\mathcal{S},\Sigma}(\{M'', N'', M''_1, \ldots, M''_n, N''_1, \ldots, N''_n\})$. By Lemma 24, there exist σ_1 and $h(M'_1, \ldots, M'_n) \to M'$ in $\text{def}_{\Sigma'}(h)$ such that $M'' = \sigma_1 M'$ and for all $i \in \{1, \ldots, n\}$, $M''_i = \sigma_1 M'_i$. By Lemma 24 again, there exist σ_1 and $h(N'_1, \ldots, N'_n) \to N'$ in $\text{def}_{\Sigma'}(h)$ such that $N'' = \sigma_1 N'$ and for all $i \in \{1, \ldots, n\}$, $N''_i = \sigma_1 N'_i$. (We can use the same substitution σ_1 since the first and second arguments of the predicate att$'$ do not share variables.) Hence $\sigma_1\text{att}'(M'_i, N'_i) = \text{att}'(M''_i, N''_i)$ for all $i \in \{1, \ldots, n\}$ and $\sigma_1\text{att}'(M', N') = \text{att}'(M'', N'')$. We take $H_1 \to C_1 = \text{att}'(M'_1, N'_1) \wedge \ldots \wedge \text{att}'(M'_n, N'_n) \to \text{att}'(M', N')$, which yields the desired result.

- For Clause (Rt), we have $C_2 = C = \text{bad}$, $H = H_{\text{nounif}} \wedge \text{att}'(M_1, x_1) \wedge \ldots \wedge \text{att}'(M_n, x_n)$, $H_2 = H_{2\text{nounif}} \wedge \text{att}'(M''_1, N''_1) \wedge \ldots \wedge \text{att}'(M''_n, N''_n)$ where H_{nounif} and $H_{2\text{nounif}}$ consist of nounif facts, $\Sigma \vdash H_{2\text{nounif}} = \sigma H_{\text{nounif}}$, $g(M_1, \ldots, M_n) \to M$ in $\text{def}_{\Sigma'}(g)$, $\Sigma \vdash M''_i = \sigma M_i$ and $\Sigma \vdash N''_i = \sigma x_i$ for all $i \in \{1, \ldots, n\}$, and $\text{nf}_{\mathcal{S},\Sigma}(\{M''_1, \ldots, M''_n, N''_1, \ldots, N''_n\})$. By Lemma 24, there exist σ_1 and $g(M'_1, \ldots, M'_n) \to M'$ in $\text{def}_{\Sigma'}(g)$ such that $M'' = \sigma_1 M'$ and for all $i \in \{1, \ldots, n\}$, $M''_i = \sigma_1 M'_i$. We extend σ_1 by defining for all $i \in \{1, \ldots, n\}$, $\sigma_1 x_i = N''_i$. Hence $\sigma_1\text{att}'(M'_i, x_i) = \text{att}'(M''_i, N''_i)$ for all $i \in \{1, \ldots, n\}$ and $\Sigma \vdash H_{2\text{nounif}} = \sigma H_{\text{nounif}} = \sigma_1 H_{\text{nounif}}$ since for all $i \in \{1, \ldots, n\}$, $\Sigma \vdash \sigma_1 x_i = N''_i = \sigma x_i$ and $fv(H_{\text{nounif}}) = \{x_1, \ldots, x_n\}$. We take $H_1 \to C_1 = H_{\text{nounif}} \wedge \text{att}'(M'_1, x_1) \wedge \ldots \wedge \text{att}'(M'_n, x_n) \to \text{bad}$ which yields the result.

  The case of Clause (Rt$'$) is symmetric. □

Lemma 29 *Assume P_0 is a closed, unevaluated process. If F is derivable from $\mathcal{R}_{P_0}$, $\Sigma \vdash F'' \sim F$, and $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{F} \cup \{F''\})$, then F'' is derivable from $\mathcal{R}_{P_0}$ by a derivation $\mathcal{D}$ such that $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{F} \cup \{\mathcal{D}\})$.*

Proof The proof is by induction on the derivation of F. Assume that F is derived from $F_1, \dots, F_n$, using a clause $R \in \mathcal{R}_{P_0}$: there exists a closed substitution σ such that $\sigma R = F_1 \wedge \dots \wedge F_n \rightarrow F$. Let F'' such that $\Sigma \vdash F'' \sim F$ and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{F} \cup \{F''\})$. By Property S2, there exist $F''_1, \dots, F''_n$ such that $\Sigma \vdash F''_i \sim F_i$ for all $i \in \{1, \dots, n\}$ and $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{F} \cup \{F'', F''_1, \dots, F''_n\})$. By Lemma 28, there exist a closed substitution σ_1 and a clause $R' = F'_1 \wedge \dots \wedge F'_n \rightarrow F' \in \mathcal{R}_{P_0}$ such that $F'' = \sigma_1 F'$ and $F''_i \sim \sigma_1 F'_i$ for all $i \in \{1, \dots, n\}$. So F'' is derivable from $F''_1, \dots, F''_n$ by R'. Furthermore, for all $i \in \{1, \dots, n\}$, $\Sigma \vdash F''_i \sim F_i$, $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{F} \cup \{F''_1, \dots, F''_n, F''\})$, and F_i is derivable from $\mathcal{R}_{P_0}$. So by induction hypothesis, F''_i is derivable from $\mathcal{R}_{P_0}$, by a derivation $\mathcal{D}_i$ such that $\mathrm{nf}_{\mathcal{S},\Sigma}(\mathcal{F} \cup \{\mathcal{D}_1, \dots, \mathcal{D}_i, F''_{i+1}, \dots, F''_n, F''\})$. (We apply the induction hypothesis with $\mathcal{F} \cup \{\mathcal{D}_1, \dots, \mathcal{D}_{i-1}, F''_{i+1}, \dots, F''_n, F''\}$ instead of $\mathcal{F} \cup \{F''\}$.) Then F'' is derivable from $\mathcal{R}_{P_0}$ by a derivation $\mathcal{D}$ built from $\mathcal{D}_1, \dots, \mathcal{D}_n$ and R', such that $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{F} \cup \{\mathcal{D}\})$. □

Lemma 3 is a particular case of Lemma 29, taking $F = F'' = \mathrm{bad}$.

D Proof of Theorem 3

The following lemma is useful for establishing the soundness of the translation of "there exists no p such that $\sigma D \Downarrow_\Sigma p$" into $\sigma\mathrm{fails}(D)$. This translation appears when we show the soundness of clauses for term evaluations.

Lemma 30 *If $\sigma\mathrm{fails}(D)$ is false then there exists a pattern p such that $\sigma D \Downarrow_\Sigma p$.*

Proof By definition of fails, there exist a pattern p and σ' such that $D \Downarrow' (p, \sigma')$ and $\sigma\mathrm{nounif}(D, GVar(\sigma' D))$ is false. By definition of nounif, there exists a closed σ'' such that $\Sigma \vdash \sigma D = \sigma''\sigma' D$. By Lemma 25, since $D \Downarrow' (p, \sigma')$, there exists p' such that $\sigma''\sigma' D \Downarrow_\Sigma p'$ and $\Sigma \vdash p' = \sigma'' p$. By a variant of Lemma 18 for patterns instead of terms, $\Sigma \vdash \sigma D = \sigma''\sigma' D$ implies $\sigma D \Downarrow_\Sigma p''$ for some p'' such that $\Sigma \vdash p'' = p'$. □

Proof (of Theorem 3) We exploit the theory developed in [3, 16] to prove the hypotheses of Lemma 2. This theory uses a type system to express the invariant that corresponds to the soundness of the clauses, and a subject reduction theorem to show that the invariant is indeed preserved. Here, types range over *pairs of closed patterns*, after adding constant session identifiers λ to the grammar of patterns.

We first define instrumented biprocesses in which a pattern is associated with each name. The syntax of instrumented biprocesses is the same as the syntax of biprocesses except that the replication is replaced with $!^i P$ where i is a variable session identifier and the restriction is replaced with $(\nu a : a_0[M_1, \dots, M_n])$ where a_0 is a function symbol and $M_1, \dots, M_n$ are terms or (constant or variable) session identifiers. In Section 6.3 and below, we reuse the name a as function symbol a_0. In contrast with a and any names occurring in $M_1, \dots, M_n$, however, the function symbol a_0 is not subject to renaming, so we may have $a \neq a_0$ after an α-conversion on a.

To every closed biprocess P with pairwise distinct bound variables, we associate the instrumented biprocess $\mathrm{instr}(P)$ obtained by adding a distinct session identifier i to each replication in P and by labelling each restriction (νa) of P with $(\nu a : a[x_1, \dots, x_n])$ where $x_1, \dots, x_n$ are the variables and session identifiers bound above (νa) in $\mathrm{instr}(P)$. Conversely, we let $\mathrm{delete}(P)$ be the biprocess obtained by erasing instrumentation from any instrumented biprocess P.

We define the semantics of instrumented biprocesses using configurations $\Lambda; P$ where Λ is a countable set of constant session identifiers and P is an instrumented biprocess. Intuitively, Λ is the set of session identifiers not yet used in the reduction of P. The rule (Red Repl) is defined as follows for instrumented biprocesses:

$$\Lambda; !^i P \rightarrow \Lambda - \{\lambda\}; !^i P \mid P\{\lambda/i\} \text{ if } \lambda \in \Lambda$$

This rule chooses a fresh session identifier λ in Λ, removes it from Λ, and uses it for the new copy of P. The other rules of Figures 2 and 3 that define reduction and structural congruence are lifted from $P \rightarrow Q$ to $\Lambda; P \rightarrow \Lambda; Q$ and from $P \equiv Q$ to $\Lambda; P \equiv \Lambda; Q$.

By construction, instrumented biprocesses include the variables that were collected by s and s' in the definition of $[\![_]\!]\rho s s' H$ of Section 6.3. Hence, the clauses $[\![P]\!]\rho_0 \emptyset\emptyset\emptyset$ can be computed from $\mathrm{instr}(P)$ as follows: $[\![P]\!]\rho_0 \emptyset\emptyset\emptyset = [\![\mathrm{instr}(P)]\!]\rho_0 \emptyset$ where

$$
\begin{aligned}
&[\![0]\!]\rho H = \emptyset \\
&[\![!^i P]\!]\rho H = [\![P]\!](\rho[i \mapsto (i,i)])H \\
&[\![P \mid Q]\!]\rho H = [\![P]\!]\rho H \cup [\![Q]\!]\rho H \\
&[\![(\nu a : a[x_1,\dots,x_n])P]\!]\rho H = \\
&\quad [\![P]\!](\rho[a \mapsto (a[\rho(x_1)_1,\dots,\rho(x_n)_1], a[\rho(x_1)_2,\dots,\rho(x_n)_2])])H \\
&[\![M(x).P]\!]\rho H = [\![P]\!](\rho[x \mapsto (x',x'')])(H \wedge \mathrm{msg}'(\rho(M)_1, x', \rho(M)_2, x'')) \\
&\quad \cup \{H \rightarrow \mathrm{input}'(\rho(M)_1, \rho(M)_2)\} \\
&\quad \text{where } x' \text{ and } x'' \text{ are fresh variables} \\
&[\![\overline{M}\langle N\rangle.P]\!]\rho H = [\![P]\!]\rho H \cup \{H \rightarrow \mathrm{msg}'(\rho(M)_1, \rho(N)_1, \rho(M)_2, \rho(N)_2)\} \\
&[\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho H = \\
&\quad \bigcup\{[\![P]\!]((\sigma\rho)[x \mapsto (p,p')])(\sigma H) \mid (\rho(D)_1, \rho(D)_2) \Downarrow' ((p,p'),\sigma)\} \\
&\quad \cup [\![Q]\!]\rho(H \wedge \rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \rho(\mathrm{fails}(\mathsf{snd}(D)))_2) \\
&\quad \cup \{\sigma H \wedge \sigma\rho(\mathrm{fails}(\mathsf{snd}(D)))_2 \rightarrow \mathrm{bad} \mid \rho(D)_1 \Downarrow' (p,\sigma)\} \\
&\quad \cup \{\sigma H \wedge \sigma\rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \rightarrow \mathrm{bad} \mid \rho(D)_2 \Downarrow' (p',\sigma)\}
\end{aligned}
$$

Let C be a plain evaluation context. For each reduction $\mathrm{unevaluated}(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv P$, there is a reduction $\Lambda_0; \mathrm{instr}(\mathrm{unevaluated}(C[P_0])) \rightarrow^*_{\Sigma',\Sigma}\equiv \Lambda; P'$ such that $\mathrm{delete}(P') = P$ (and conversely). Let $P'_0 = \mathrm{instr}(\mathrm{unevaluated}(P_0))$. There exists an unevaluated evaluation context C' such that diff occurs only in terms $\mathsf{diff}[a,a]$ for some names a in C' and $\mathrm{instr}(\mathrm{unevaluated}(C[P_0])) = C'[P'_0]$.

Let $\mathcal{R}_{C',P_0}$ be the set of clauses obtained by adding to $\mathcal{R}_{P_0}$ the clauses

$$\mathrm{att}'(a[x_1,\dots,x_n], a[x_1,\dots,x_n]) \qquad (\mathrm{Rn}')$$

such that either $(\nu a : a[x'_1,\dots,x'_n])$ occurs in C' or $n = 0$, $a \in \mathit{fn}(C')$, and $a \notin \mathit{fn}(P'_0)$. The fact bad is derivable from $\mathcal{R}_{C',P_0}$ if and only if bad is derivable from $\mathcal{R}_{P_0}$, since we can replace all patterns $a[\dots]$ of names created by the context C' with patterns $b[i]$, as long as different names have different images, so we can replace the Clauses (Rn$'$) with Clause (Rn). Hence, the definition of $\mathcal{R}_{P_0}$ is sufficient.

Next, we define a type system, similar to that of [3, Section 7]. Here, the types are pairs of closed patterns. The type environment E is a function from variables and names to types. It is extended to terms as a substitution, so that a term M has type $E(M)$. The typing judgment $E \vdash P$ says that the instrumented biprocess P is well-typed in environment E. This judgment is formally defined in Figure 5, where $\mathcal{F}_{C',P_0}$ is the set of closed facts derivable from $\mathcal{R}_{C',P_0}$.

When $M_1,\dots,M_n$ is a sequence of terms and (variable or constant) session identifiers, as in labels of restrictions, we define $\mathrm{last}(M_1,\dots,M_n)$ as the last M_i that is a session identifier, or $\emptyset$ when no M_i is a session identifier. Let us define the multiset $\mathit{Label}(P)$ as follows: $\mathit{Label}((\nu a : a_0[M_1,\dots,M_n])P) = \{(a_0, \mathrm{last}(M_1,\dots,M_n))\} \cup \mathit{Label}(P)$, $\mathit{Label}(!^i P) = \emptyset$, and in all other cases, $\mathit{Label}(P)$ is the union of the $\mathit{Label}(P')$ for all immediate subprocesses P' of P. When E maps names to closed patterns, let $\mathit{Label}(E) = \{(a_0, \mathrm{last}(M_1,\dots,M_n)) \mid (a \mapsto a_0[M_1,\dots,M_n] \in E\}$. Let $\mathit{Label}(\Lambda) = \{(a,\lambda) \mid \lambda \in \Lambda\}$. We say that $E \vdash \Lambda; P$ is well-labelled

$$\frac{\text{input}'(E(M)_1, E(M)_2) \in \mathcal{F}_{C',P_0} \quad \forall p_1, p_2 \text{ such that } \text{msg}'(E(M)_1, p_1, E(M)_2, p_2) \in \mathcal{F}_{C',P_0}, E[x \mapsto (p_1, p_2)] \vdash P}{E \vdash M(x).P} \quad \text{(Input)}$$

$$\frac{\text{msg}'(E(M)_1, E(N)_1, E(M)_2, E(N)_2) \in \mathcal{F}_{C',P_0} \qquad E \vdash P}{E \vdash \overline{M}\langle N \rangle.P} \quad \text{(Output)}$$

$$\frac{}{E \vdash 0} \quad \text{(Nil)}$$

$$\frac{E \vdash P \qquad E \vdash Q}{E \vdash P \mid Q} \quad \text{(Parallel)}$$

$$\frac{\forall \lambda, E[i \mapsto (\lambda, \lambda)] \vdash P}{E \vdash !^i P} \quad \text{(Replication)}$$

$$\frac{E[a \mapsto (a_0[E(M_1)_1, \dots, E(M_n)_1], a_0[E(M_1)_2, \dots, E(M_n)_2])] \vdash P}{E \vdash (\nu a : a_0[M_1, \dots, M_n])P} \quad \text{(Restriction)}$$

$$\frac{\begin{array}{c}\forall p_1, p_2 \text{ such that } E(D)_1 \Downarrow_{\Sigma'} p_1 \text{ and } E(D)_2 \Downarrow_{\Sigma'} p_2, E[x \mapsto (p_1, p_2)] \vdash P \\ \text{if } \nexists p_1, E(D)_1 \Downarrow_{\Sigma} p_1 \text{ and } \nexists p_2, E(D)_2 \Downarrow_{\Sigma} p_2, \text{ then } E \vdash Q \\ \text{if } \exists p_1, E(D)_1 \Downarrow_{\Sigma'} p_1 \text{ and } \nexists p_2, E(D)_2 \Downarrow_{\Sigma} p_2, \text{ then bad} \in \mathcal{F}_{C',P_0} \\ \text{if } \nexists p_1, E(D)_1 \Downarrow_{\Sigma} p_1 \text{ and } \exists p_2, E(D)_2 \Downarrow_{\Sigma'} p_2, \text{ then bad} \in \mathcal{F}_{C',P_0}\end{array}}{E \vdash \mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q} \quad \text{(Term evaluation)}$$

Figure 5: Type rules

when the multisets $\mathit{Label}(E_1) \cup \mathit{Label}(\Lambda) \cup \mathit{Label}(P)$ and $\mathit{Label}(E_2) \cup \mathit{Label}(\Lambda) \cup \mathit{Label}(P)$ contain no duplicates, where E_1 and E_2 are the first and second components of E. We say that $E \vdash \Lambda; P$ when $E \vdash \Lambda; P$ is well-labelled and $E \vdash P$. Showing that $\mathit{Label}(E_1)$ and $\mathit{Label}(E_2)$ contain no duplicates guarantees that different terms have different types. More precisely, if E maps names to closed patterns $a[\dots]$, E is extended to terms as a substitution, and $\mathit{Label}(E)$ contains no duplicates, then we have the following properties:

E1. E is an injection (if $E(M) = E(N)$ then $M = N$) and also an injection modulo Σ (if $\Sigma \vdash E(M) = E(N)$ then $\Sigma \vdash M = N$).

E2. Let N be a term not containing names; if $E(M')$ is an instance of N, then M' is an instance of N; if $E(M')$ is an instance of N modulo Σ, then M' is an instance of N modulo Σ.

E3. If $D \Downarrow_{\Sigma'} M$, then $E(D) \Downarrow_{\Sigma'} E(M)$. (This is proved by induction on D.)

E4. If $\Sigma \vdash D' = E(D)$ and $D' \Downarrow_{\Sigma} p'$ then there exists M such that $\Sigma \vdash p' = E(M)$ and $D \Downarrow_{\Sigma} M$. (This is proved by induction on D, using E2.)

Let $E_0 = \{a \mapsto (a[\,], a[\,]) \mid a \in \mathit{fn}(C'[P'_0])\}$.

1. *Typability of the adversary:* Let P' be a subprocess of C'. Let E be an environment such that for all $a \in \mathit{fn}(P')$, $\text{att}'(E(a), E(a)) \in \mathcal{F}_{C',P_0}$ and for all $x \in \mathit{fv}(P')$, $\text{att}'(E(x), E(x)) \in \mathcal{F}_{C',P_0}$. We show that $E \vdash P'$ by induction on P', similarly to [3, Lemma 5.1.4].

   We detail the case of term evaluations, since it significantly differs from that in [3]. In order to show the desired property in this case, it suffices to show that if for all $a \in \mathit{fn}(D)$, $\text{att}'(E(a), E(a)) \in \mathcal{F}_{C',P_0}$ and for all $x \in \mathit{fv}(D)$, $\text{att}'(E(x), E(x)) \in \mathcal{F}_{C',P_0}$, then we have the two properties:

(a) if $E(D)_1 \Downarrow_{\Sigma'} p_1$ and $E(D)_2 \Downarrow_{\Sigma'} p_2$, then $\mathrm{att}'(p_1, p_2) \in \mathcal{F}_{C',P_0}$;
(b) if $E(D)_1 \Downarrow_{\Sigma'} p_1$ and $/\exists p_2, E(D)_2 \Downarrow_{\Sigma} p_2$, then $\mathrm{bad} \in \mathcal{F}_{C',P_0}$; symmetrically, if $E(D)_2 \Downarrow_{\Sigma'} p_2$ and $\nexists p_1, E(D)_1 \Downarrow_{\Sigma} p_1$, then $\mathrm{bad} \in \mathcal{F}_{C',P_0}$.

The proof is by induction on D.

- Case $D = \mathsf{diff}[a, a]$: We have $E(D)_1 = E(a)_1 \Downarrow_{\Sigma'} E(a)_1$ and $E(D)_2 = E(a)_2 \Downarrow_{\Sigma'} E(a)_2$, and by hypothesis $\mathrm{att}'(E(a)_1, E(a)_2) \in \mathcal{F}_{C',P_0}$, so Property (a) holds. We also have $E(D)_1 = E(a)_1 \Downarrow_{\Sigma} E(a)_1$ and $E(D)_2 = E(a)_2 \Downarrow_{\Sigma} E(a)_2$, so Property (b) holds.
- Case $D = x$: This case is similar to that for $D = \mathsf{diff}[a, a]$.
- Case $D = \mathrm{eval}\ h(D_1, \dots, D_n)$: Property (a) follows from the induction hypothesis and Clause (Rf). Next, we prove the first part of Property (b). The second part of Property (b) follows by symmetry.

  Since $E(D)_1 \Downarrow_{\Sigma'} p_1$, there exist $h(N_1, \dots, N_n) \to N$ in $\mathrm{def}_{\Sigma'}(h)$, p_1, $p_{1,1}$, $\dots$, $p_{1,n}$, and σ such that $E(D_i)_1 \Downarrow_{\Sigma'} p_{1,i}$ for all $i \in \{1, \dots, n\}$, $p_1 = \sigma N$, and $p_{1,i} = \sigma N_i$ for all $i \in \{1, \dots, n\}$. Since there exists no p_2 such that $E(D)_2 \Downarrow_{\Sigma} p_2$, either for some $i \in \{1, \dots, n\}$ there exists no $p_{2,i}$ such that $E(D_i)_2 \Downarrow_{\Sigma} p_{2,i}$ (and $\mathrm{bad} \in \mathcal{F}_{C',P_0}$ by induction hypothesis), or for all $i \in \{1, \dots, n\}$ there exists $p_{2,i}$ such that $E(D_i)_2 \Downarrow_{\Sigma} p_{2,i}$, and there exist no $h(N'_1, \dots, N'_n) \to N'$ in $\mathrm{def}_{\Sigma}(h)$ and σ such that for all $i \in \{1, \dots, n\}$, $\Sigma \vdash p_{2,i} = \sigma N'_i$. Hence, h must be a destructor.

  By Property S2, there exists an environment E' such that $\Sigma \vdash E'(a) = E(a)$ for all $a \in \mathit{fn}(D)$, $\Sigma \vdash E'(x) = E(x)$ for all $x \in \mathit{fv}(D)$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(E')$. By Lemma 29, $\mathrm{att}'(E'(a)_1, E'(a)_2) \in \mathcal{F}_{C',P_0}$ for all $a \in \mathit{fn}(D)$ and $\mathrm{att}'(E'(x)_1, E'(x)_2) \in \mathcal{F}_{C',P_0}$ for all $x \in \mathit{fv}(D)$. We have $\mathrm{nf}_{\mathcal{S},\Sigma}(E'(D_i))$ and $\Sigma \vdash E'(D_i)_2 = E(D_i)_2$. By Property S2, there exist $p'_{2,1}$, $\dots$, $p'_{2,n}$ such that $\Sigma \vdash p'_{2,i} = p_{2,i}$ for all $i \in \{1, \dots, n\}$ and $\mathrm{nf}_{\mathcal{S},\Sigma}(E', p'_{2,1}, \dots, p'_{2,n})$. By a variant of Lemma 14 for patterns instead of terms, $E'(D_i)_2 \Downarrow_{\Sigma'} p'_{2,i}$ for all $i \in \{1, \dots, n\}$.

  By a variant of Lemma 18 for patterns instead of terms, $E'(D_i)_1 \Downarrow_{\Sigma} p'_{1,i}$ for some $p'_{1,i}$ such that $\Sigma \vdash p'_{1,i} = p_{1,i}$. By Property S2, there exist $p''_{1,1}, \dots, p''_{1,n}, p''_1$ such that $\Sigma \vdash p''_{1,i} = p'_{1,i}$ for all $i \in \{1, \dots, n\}$, $\Sigma \vdash p''_1 = p_1$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(E', p'_{2,1}, \dots, p'_{2,n}, p''_{1,1}, \dots, p''_{1,n}, p''_1)$. By a variant of Lemma 14 for patterns instead of terms, $E'(D_i)_1 \Downarrow_{\Sigma'} p''_{1,i}$ for all $i \in \{1, \dots, n\}$. By induction hypothesis, Property (a), we obtain $\mathrm{att}'(p''_{1,i}, p'_{2,i}) \in \mathcal{F}_{C',P_0}$ for all $i \in \{1, \dots, n\}$.

  Since $\Sigma \vdash p'_{2,i} = p_{2,i}$, there exist no σ and $h(N'_1, \dots, N'_n) \to N'$ in $\mathrm{def}_{\Sigma}(h)$ such that for all $i \in \{1, \dots, n\}$, $\Sigma \vdash p'_{2,i} = \sigma N'_i$. By Lemma 16, there exist no σ and $h(N'_1, \dots, N'_n) \to N'$ in $\mathrm{def}_{\Sigma'}(h)$ such that for all $i \in \{1, \dots, n\}$, $\Sigma \vdash p'_{2,i} = \sigma N'_i$, that is, we have

$$\bigwedge_{h(N'_1,\dots,N'_n) \to N' \text{ in } \mathrm{def}_{\Sigma'}(h)} \mathrm{nounif}((p'_{2,1}, \dots, p'_{2,n}), \mathit{GVar}(N'_1, \dots, N'_n))$$

  Since $\Sigma \vdash p''_{1,i} = p_{1,i}$, $\Sigma \vdash p''_1 = p_1$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(p''_{1,1}, \dots, p''_{1,n}, p''_1)$, by Lemma 16 and a variant of Lemma 13 for patterns instead of terms, there exist $h(N_1, \dots, N_n) \to N$ is $\mathrm{def}_{\Sigma'}(h)$ and Σ such that $p''_{1,i} = \sigma N_i$ for all $i \in \{1, \dots, n\}$ and $p''_1 = \sigma N$. Hence, by Clause (Rt), $\mathrm{bad} \in \mathcal{F}_{C',P_0}$.

2. *Typability of P'_0:* We prove by induction on the process P, subprocess of P'_0, that, if (a) ρ binds all free names and variables of P, (b) σ is a closed substitution, (c) $\mathcal{R}_{C',P_0} \supseteq [\![P]\!]\rho H$, and (d) σH can be derived from $\mathcal{R}_{C',P_0}$, then $\sigma\rho \vdash P$.

   Again, we detail the case of term evaluations. We suppose that ρ binds all free names and variables of *let* $x = D$ *in* P *else* Q, σ is a closed substitution, $\mathcal{R}_{C',P_0} \supseteq [\![$*let* $x =$

D *in* P *else* $Q]\!]\rho H$, and σH is derivable from $\mathcal{R}_{C',P_0}$. We show that $\sigma\rho \vdash$ *let* $x = D$ *in* P *else* Q. To apply the type rule (Term evaluation), it suffices to show that:

- For all p_1, p_2 such that $\sigma\rho(D)_1 \Downarrow_{\Sigma'} p_1$ and $\sigma\rho(D)_2 \Downarrow_{\Sigma'} p_2$, we have $\sigma\rho[x \mapsto (p_1, p_2)] \vdash P$.

  By a variant of Lemma 11 for patterns instead of terms, there exist p'_1, p'_2, σ', and σ'' such that $(\rho(D)_1, \rho(D)_2) \Downarrow' ((p'_1, p'_2), \sigma')$, $p_1 = \sigma'' p'_1$, $p_2 = \sigma'' p'_2$, and $\sigma = \sigma''\sigma'$ except on the fresh variables introduced in the computation of $(\rho(D)_1, \rho(D)_2) \Downarrow' ((p'_1, p'_2), \sigma')$.

  Hence $\sigma''\sigma' H = \sigma H$ can be derived from $\mathcal{R}_{C',P_0}$, and $[\![P]\!]((\sigma'\rho)[x \mapsto (p'_1, p'_2)])(\sigma' H) \subseteq [\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho H \subseteq \mathcal{R}_{C',P_0}$ so, by induction hypothesis, $\sigma''(\sigma'\rho[x \mapsto (p'_1, p'_2)]) \vdash P$, that is, $\sigma\rho[x \mapsto (p_1, p_2)] \vdash P$.

- If there exists no p_1 such that $\sigma\rho(D)_1 \Downarrow_{\Sigma} p_1$ and there exists no p_2 such that $\sigma\rho(D)_2 \Downarrow_{\Sigma} p_2$, then $\sigma\rho \vdash Q$.

  By Lemma 30, $\sigma\rho(\mathrm{fails}(\mathsf{fst}(D)))_1$ and $\sigma\rho(\mathrm{fails}(\mathsf{snd}(D)))_2$ are true, so $\sigma(H \wedge \rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \rho(\mathrm{fails}(\mathsf{snd}(D)))_2)$ can be derived from $\mathcal{R}_{C',P_0}$. Moreover $[\![Q]\!]\rho(H \wedge \rho(\mathrm{fails}(\mathsf{fst}(D)))_1 \wedge \rho(\mathrm{fails}(\mathsf{snd}(D)))_2) \subseteq [\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho H \subseteq \mathcal{R}_{C',P_0}$ so, by induction hypothesis, $\sigma\rho \vdash Q$.

- If there exists p_1 such that $\sigma\rho(D)_1 \Downarrow_{\Sigma'} p_1$ and there exists no p_2 such that $\sigma\rho(D)_2 \Downarrow_{\Sigma} p_2$, then bad $\in \mathcal{F}_{C',P_0}$.

  By a variant of Lemma 11 for patterns instead of terms, there exist p'_1, σ', and σ'' such that $\rho(D)_1 \Downarrow' (p'_1, \sigma')$, $p_1 = \sigma'' p'_1$, and $\sigma = \sigma''\sigma'$ except on the fresh variables introduced in the computation of $\rho(D)_1 \Downarrow' (p'_1, \sigma')$. There exists no p_2 such that $\sigma''\sigma'\rho(D)_2 \Downarrow_{\Sigma} p_2$, so by Lemma 30, $\sigma''\sigma'\rho(\mathrm{fails}(\mathsf{snd}(D)))_2$ holds, hence $\sigma''(\sigma' H \wedge \sigma'\rho(\mathrm{fails}(\mathsf{snd}(D)))_2)$ can be derived from $\mathcal{R}_{C',P_0}$. Since $\sigma' H \wedge \sigma'\rho(\mathrm{fails}(\mathsf{snd}(D)))_2 \rightarrow \mathrm{bad} \in [\![\mathit{let}\ x = D\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho H \subseteq \mathcal{R}_{C',P_0}$, bad $\in \mathcal{F}_{C',P_0}$.

- If there exists no p_1 such that $\sigma\rho(D)_1 \Downarrow_{\Sigma} p_1$ and there exists p_2 such that $\sigma\rho(D)_2 \Downarrow_{\Sigma'} p_2$, then bad $\in \mathcal{F}_{C',P_0}$. This property follows from the one above by symmetry.

By definition, $\mathcal{R}_{C',P_0} \supseteq [\![P'_0]\!]\rho_0\emptyset$, where $\rho_0 = \{a \mapsto (a[\,], a[\,]) \mid a \in \mathit{fn}(P'_0)\}$. Taking $P = P'_0$, we obtain $E \vdash P'_0$ with $E = \sigma\rho_0 = \{a \mapsto (a[\,], a[\,]) \mid a \in \mathit{fn}(P'_0)\}$. (This result is similar to [3, Lemma 7.2.2].)

3. *Properties of* $C'[P'_0]$*:* We show that $E_0 \vdash \Lambda_0; C'[P'_0]$. In order to prove this result, we show that $E_0 \vdash C'[P'_0]$ by induction on C'.

   When $C' = [\,]$, the result follows from Property 2. When $C' = (\nu a : a[\,])C''$, the result follows by induction hypothesis and the type rule (Restriction). When $C' = C'' \mid Q$, the result follows from Property 1 and the type rule (Parallel).

4. *Substitution lemma:* Let $E' = E[x \mapsto (E(M)_1, E(M)_2)]$. We show by induction on M' that $E(M'\{M/x\}) = E'(M')$. We show by induction on P that, if $E' \vdash P$, then $E \vdash P\{M/x\}$. This is similar to [3, Lemma 5.1.1].

5. *Subject congruence:* If $E \vdash \Lambda; P$ and $P \equiv P'$, then $E \vdash \Lambda; P'$. We prove by induction on the derivation of $P \equiv P'$ that if $E \vdash P$ and $P \equiv P'$, then $E \vdash P'$ and $\mathit{Label}(P') = \mathit{Label}(P)$, similarly to [3, Lemma 5.1.2].

6. *Subject reduction:* If $E \vdash \Lambda; P$ and $\Lambda; P \rightarrow \Lambda'; P'$, then $E \vdash \Lambda', P'$. We prove by induction on the derivation of $\Lambda; P \rightarrow \Lambda'; P'$ that if $E \vdash \Lambda; P$ and $\Lambda; P \rightarrow \Lambda'; P'$, then $E \vdash \Lambda', P'$ and $\mathit{Label}(\Lambda') \cup \mathit{Label}(P') \subseteq \mathit{Label}(\Lambda) \cup \mathit{Label}(P)$, similarly to [3, Lemma 5.1.3].

7. *Proof of the second hypothesis of Lemma 2:* Assume that

$$\text{unevaluated}(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv C_1[let\ y = D\ in\ Q\ else\ Q']$$

and $\mathsf{fst}(D) \Downarrow_{\Sigma'} M_1$ for some M_1. Then $\Lambda_0; C'[P'_0] \rightarrow^*_{\Sigma',\Sigma}\equiv \Lambda; C'_1[let\ y = D\ in\ Q_1\ else\ Q'_1]$ where $\text{delete}(C'_1[let\ y = D\ in\ Q_1\ else\ Q'_1]) = C_1[let\ y = D\ in\ Q\ else\ Q']$. We have $E_0 \vdash \Lambda_0; C'[P'_0]$, so by subject reduction and subject congruence, $E_0 \vdash \Lambda; C'_1[let\ y = D\ in\ Q_1\ else\ Q'_1]$. Since $E_0 \vdash C'_1[let\ y = D\ in\ Q_1\ else\ Q'_1]$ has been derived by type rules (Restriction) and (Parallel), there exists an environment E such that $E \vdash let\ y = D\ in\ Q_1\ else\ Q'_1$ and since $Label((E_0)_1) \cup Label(\Lambda) \cup Label(C'_1[let\ y = D\ in\ Q_1\ else\ Q'_1])$ and $Label((E_0)_2) \cup Label(\Lambda) \cup Label(C'_1[let\ y = D\ in\ Q_1\ else\ Q'_1])$ contain no duplicates, $Label(E_1)$ and $Label(E_2)$ contain no duplicates.

Since $\mathsf{fst}(D) \Downarrow_{\Sigma'} M_1$, by Property E3, $E(D)_1 \Downarrow_{\Sigma'} E(M_1)_1$. Since $E \vdash let\ y = D\ in\ Q_1\ else\ Q'_1$ has been derived by type rule (Term evaluation) and bad $\notin \mathcal{F}_{C',P_0}$, there exists p_2 such that $E(D)_2 \Downarrow_{\Sigma} p_2$. So by Property E4, there exists M_2 such that $\mathsf{snd}(D) \Downarrow_{\Sigma} M_2$, which establishes the second hypothesis of Lemma 2.

8. *Proof of the first hypothesis of Lemma 2:* Assume that

$$\text{unevaluated}(C[P_0]) \rightarrow^*_{\Sigma',\Sigma}\equiv C_1[\overline{N}\langle M\rangle.Q \mid N'(x).R]$$

and $\mathsf{fst}(N) = \mathsf{fst}(N')$. As above, there exists an environment E such that $E \vdash \overline{N}\langle M\rangle.Q' \mid N'(x).R'$ and E_1 and E_2 satisfy Properties E1, E2, E3, and E4.

Since $E \vdash \overline{N}\langle M\rangle.Q' \mid N'(x).R'$ has been derived by type rules (Parallel), (Output), and (Input), we have $\text{msg}'(E(N)_1, E(M)_1, E(N)_2, E(M)_2) \in \mathcal{F}_{C',P_0}$ and $\text{input}'(E(N')_1, E(N')_2) \in \mathcal{F}_{C',P_0}$. Since $\mathsf{fst}(N) = \mathsf{fst}(N')$, $E(N)_1 = E(N')_1$. Since bad is not derivable from $\mathcal{R}_{C',P_0}$, $\text{nounif}(E(N)_2, E(N')_2)$ is false—otherwise bad would be derivable by (Rcom)—so, by definition of nounif, $\Sigma \vdash E(N)_2 = E(N')_2$. By Property E1, E_2 is injective modulo Σ and we obtain $\Sigma \vdash \mathsf{snd}(N') = \mathsf{snd}(N)$.

The symmetric hypotheses of Lemma 2 follow by symmetry.

To conclude our proof of Theorem 3, we apply Lemma 2 and Corollary 1. □

E Proof of Theorem 4

E.1 Unification modulo the equational theory

We use the standard convention that, when computing a most general unifier σ_u of M_i, N_i for $i \in \{1, \ldots, n\}$, we always arrange that $dom(\sigma_u) \cap fv(im(\sigma_u)) = \emptyset$ and $dom(\sigma_u) \cup fv(\sigma_u M_1, \sigma_u N_1, \ldots, \sigma_u M_n, \sigma_u N_n) \subseteq \cup_i (fv(M_i) \cup fv(N_i))$. (We recall that $dom(\sigma) = \{x \mid x \neq \sigma x\}$.) Since $dom(\sigma_u) \cap fv(im(\sigma_u)) = \emptyset$, σ_u is idempotent.

If σ is a most general unifier of M_i, N_i for $i \in \{1, \ldots, n\}$ and σ' is a most general unifier of $\sigma M'_i, \sigma N'_i$ for $i \in \{1, \ldots, n'\}$ then $\sigma'\sigma$ is a most general unifier of M_i, N_i for $i \in \{1, \ldots, n\}$ and M'_i, N'_i for $i \in \{1, \ldots, n'\}$.

Lemma 31 *If $\sigma D \Downarrow' (M', \sigma')$ and σ is a most general unifier, then $\sigma'\sigma$ is also a most general unifier, and there exists M'' such that $M' = \sigma'\sigma M''$.*

Proof The proof is by mutual induction following the definition of $\Downarrow'$. All cases are easy. □

Lemma 32 *We have $\Sigma \vdash \sigma M = \sigma M'$ if and only if there exist N, N', σ', and σ_u such that* $\text{addeval}(M, M') \Downarrow' ((N, N'), \sigma')$*, σ_u is the most general unifier of N and N', and for all $x \in fv(M, M')$, $\Sigma \vdash \sigma x = \sigma\sigma_u\sigma' x$.*

Proof Assume $\Sigma \vdash \sigma M = \sigma M'$. By Property S2, there exist M'' and σ' such that $\Sigma \vdash M'' = \sigma M = \sigma M'$, $\Sigma \vdash \sigma x = \sigma' x$ for all $x \in fv(M, M')$, and $\mathrm{nf}_{\mathcal{S},\Sigma}(\{M''\} \cup \{\sigma' x \mid x \in fv(M, M')\})$. Since $\Sigma \vdash \sigma' M = \sigma' M' = M''$, by Lemma 12 we have $\sigma'\mathrm{addeval}(M) \Downarrow_{\Sigma'} M''$ and $\sigma'\mathrm{addeval}(M') \Downarrow_{\Sigma'} M''$. By Lemma 11, there exist $N, N', \sigma_1, \sigma'_1$ such that $\mathrm{addeval}(M, M') \Downarrow' ((N, N'), \sigma_1)$, $M'' = \sigma'_1 N$, $M'' = \sigma'_1 N'$, and $\sigma' = \sigma'_1 \sigma_1$ except on fresh variables introduced in the computation of $\mathrm{addeval}(M, M') \Downarrow' ((N, N'), \sigma_1)$.

So N and N' unify. Let σ_u be their most general unifier. Let σ''_1 such that $\sigma'_1 = \sigma''_1 \sigma_u$. Let $x \in fv(M, M')$. We have $\Sigma \vdash \sigma x = \sigma' x = \sigma''_1 \sigma_u \sigma_1 x = \sigma''_1 \sigma_u \sigma_1 \sigma_u \sigma_1 x = \sigma' \sigma_u \sigma_1 x = \sigma \sigma_u \sigma_1 x$. (Indeed, $\sigma_u \sigma_1$ is a most general unifier by Lemma 31 and the composition of most general unifiers, so it is idempotent.)

Conversely, assume that there exist $N, N', \sigma' \sigma_u$ such that $\mathrm{addeval}(M, M') \Downarrow' ((N, N'), \sigma')$, σ_u is the most general unifier of N and N' and for all $x \in fv(M, M')$, $\Sigma \vdash \sigma x = \sigma \sigma_u \sigma' x$. Then

$$\begin{aligned}
\Sigma \vdash \sigma M &= \sigma \sigma_u \sigma' M \\
&= \sigma \sigma_u N && \text{by Lemma 15} \\
&= \sigma \sigma_u N' && \text{since } \sigma_u \text{ is the most general unifier of } N \text{ and } N' \\
&= \sigma \sigma_u \sigma' M' && \text{by Lemma 15 again} \\
&= \sigma M' && \square
\end{aligned}$$

E.2 Soundness of the solving algorithm

The following proofs are partly adaptations of previous proofs [15, 18]. In addition, they establish the soundness of all simplifications for nounif.

Let $\mathcal{R}_0 = \mathcal{R}_{P_0}$ be the initial set of clauses, $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$ be the final set of clauses, and $\mathcal{R}$ be the set of clauses during the saturation. At the end of the saturation algorithm, we have $\mathcal{R}_1 = \{R \in \mathcal{R} \mid \mathsf{sel}(R) = \emptyset\}$.

Lemma 33 *At the end of the saturation, $\mathcal{R}$ satisfies the following properties:*

1. *For all $R \in simplify(\mathcal{R}_0)$, there exists $R' \in \mathcal{R}$ such that $R' \sqsupseteq R$.*

2. *Let $R \in \mathcal{R}$ and $R' \in \mathcal{R}$. Assume that $\mathsf{sel}(R) = \emptyset$ and there exists $F_0 \in \mathsf{sel}(R')$ such that $R \circ_{F_0} R'$ is defined. In this case, for all $R'' \in simplify(R \circ_{F_0} R')$, there exists $R''' \in \mathcal{R}$ such that $R''' \sqsupseteq R''$.*

Proof To prove the first property, let $R \in simplify(\mathcal{R}_0)$. We show that during the whole execution of the saturation, there exists $R' \in \mathcal{R}$ such that $R' \sqsupseteq R$.

The algorithm first builds $simplify(\mathcal{R}_0)$ (which obviously satisfies the required property), then removes subsumed clauses by *condense*. The property is preserved by elimination of subsumed clauses. So $\mathcal{R} = condense(\mathcal{R}_0)$ satisfies the property. Further additions of clauses and eliminations of subsumed clauses preserve the property, so we have the result.

The second property states that the fixpoint is reached at the end of saturation. $\square$

We now give a precise definition of derivations.

Definition 7 (Derivation) Let $\mathcal{T}_{\text{facts}}$ be the set of true nounif facts. Let $\mathcal{R}$ be a set of clauses and F be a closed fact. A derivation of F from $\mathcal{R}$ is a finite tree defined as follows:

1. Nodes (except the root) are labelled by clauses $R \in \mathcal{R}$ or nounif facts in $\mathcal{T}_{\text{facts}}$.

2. Edges are labelled by closed facts. (Edges go from a node to each of its sons.)

3. The root has one outgoing edge, labelled by F.

4. If the tree contains a node labelled by R with one incoming edge labelled by F_0 and n outgoing edges labelled by $F_1, \dots, F_n$, then $R \sqsupseteq \{F_1, \dots, F_n\} \rightarrow F_0$. If the tree contains a node labelled by a fact in $\mathcal{T}_{\text{facts}}$, then this node has one incoming edge labelled by the same fact and no outgoing edge.

In a derivation, if there is a node labelled by R with one incoming edge labelled by F_0 and n outgoing edges labelled by $F_1, \dots, F_n$, then the clause R can be used to infer F_0 from $F_1, \dots, F_n$. Therefore, there exists a derivation of F from $\mathcal{R}$ if and only if F can be inferred from clauses in $\mathcal{R}$.

The key idea of the proof of the algorithm is the following. Assume that bad is derivable from $\mathcal{R}_0$ and consider a derivation of bad from $\mathcal{R}_0$. Assume that the clauses R and R' are applied one after the other in the derivation of bad. Also assume that these clauses have been combined by $R \circ_{F_0} R'$, yielding clause R''. In this case, we replace R' by R'' in the derivation of bad. When no more replacement can be made, we show that all remaining clauses have no selected hypothesis. Then all these clauses are in $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$, and we have built a derivation of bad from $\mathcal{R}_1$. Moreover, this replacement process terminates because the number of nodes of the derivation strictly decreases.

Lemma 34 *Consider a derivation that contains a node η', labelled R'. Let F_0 be a hypothesis of R'. Then there exists a son η of η', labelled R, such that the edge from η' to η is labelled by an instance of F_0, $R \circ_{F_0} R'$ is defined, and we still have a derivation of the same fact if we replace the nodes η and η' by a node η'' labelled $R'' = R \circ_{F_0} R'$.*

Proof This proof is already given in [18], with a figure. Let $R' = H' \rightarrow C'$, H'_1 be the multiset of the labels of the outgoing edges of η', and C'_1 the label of its incoming edge. We have $R' \sqsupseteq (H'_1 \rightarrow C'_1)$, then there exists σ such that $\sigma H' \subseteq H'_1$ and $\sigma C' = C'_1$. Then there is an outgoing edge of η' labelled σF_0, since $\sigma F_0 \in H'_1$. Let η be the node at the end of this edge, let $R = H \rightarrow C$ be the label of η. We rename the variables of R so that they are distinct from the variables of R'. Let H_1 be the multiset of the labels of the outgoing edges of η. Then $R \sqsupseteq (H_1 \rightarrow \sigma F_0)$. By the above choice of distinct variables, we can then extend σ in such a way that $\sigma H \subseteq H_1$ and $\sigma C = \sigma F_0$.

The edge from η' to η is labelled σF_0, which is an instance of F_0. We have $\sigma C = \sigma F_0$, then C and F_0 are unifiable, then $R \circ_{F_0} R'$ is defined. Let σ' be the most general unifier of C and F_0, and σ'' such that $\sigma = \sigma''\sigma'$. We have $R \circ_{F_0} R' = \sigma'(H \cup (H' - F_0)) \rightarrow \sigma' C'$. Moreover, $\sigma''\sigma'(H \cup (H' - F_0)) \subseteq H_1 \cup (H'_1 - \sigma F_0)$ and $\sigma''\sigma' C' = \sigma C' = C'_1$. Then $R'' = R \circ_{F_0} R' \sqsupseteq (H_1 \cup (H'_1 - \sigma F_0)) \rightarrow C'_1$. The multiset of labels of outgoing edges of η'' is precisely $H_1 \cup (H'_1 - \sigma F_0)$ and the label of its incoming edge is C'_1, so we have obtained a correct derivation by replacing η and η' with η''. □

Lemma 35 *If $\mathcal{D}$ is a derivation whose node η is labelled R, then we obtain a derivation $\mathcal{D}'$ of the same fact by relabelling η with a clause R' such that $R' \sqsupseteq R$.*

Proof Let H be the multiset of labels of outgoing edges of the considered node η, and C be the label of its incoming edge. We have $R \sqsupseteq H \rightarrow C$. By transitivity of $\sqsupseteq$, $R' \sqsupseteq H \rightarrow C$. So we can relabel η with R'. □

We now prove the soundness of each simplification function described in Section 7, and of their composition *simplify*.

Lemma 36 *Let f range over the simplification functions simpeq, elimvar, elimGVar $\circ$ swap $\circ$ unify, elimnouniffalse, elimdup, elimattx, elimtaut, and simplify.*

Let $\mathcal{R}_t = \{(1), (2)\}$ when $f \in \{elimvar, simplify\}$ and $\mathcal{R}_t = \emptyset$ otherwise.

Let $\mathcal{D}$ be a derivation of bad *such that* $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$ *with a node η labelled R.*

We obtain a derivation $\mathcal{D}'$ of bad *by relabelling the node η with some clause $R' \in f(\{R\}) \cup \mathcal{R}_t$, deleting nodes, and modifying nodes labelled by a fact in $\mathcal{T}_{\text{facts}}$.*

The set of clauses $\mathcal{R}_t$ collects clauses that must be included in the clause set for the transformation to be correct. The proofs closely follow the intuitions for soundness given in Section 7.

Proof (for *simpeq***)** Since $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$, the facts of σR (except nounif facts) are irreducible by $\mathcal{S}$, so a fortiori the facts of R (except nounif facts) are irreducible by $\mathcal{S}$, hence $\mathit{simpeq}(\{R\}) = \{R\}$, which obviously implies the desired result. □

Proof (for *elimvar***)** Let $R = H \rightarrow C$, where $H = \text{att}'(x, y) \wedge \text{att}'(x, y') \wedge \ldots$ and $R' = R\{y/y'\}$. (The case $H = \text{att}'(y, x) \wedge \text{att}'(y', x) \wedge \ldots$ is symmetric.) Let H' be the multiset of labels of outgoing edges of η and C' the label of its incoming edge. Since $\mathcal{D}$ is a derivation, there exists σ such that $\sigma H \subseteq H'$, and $\sigma C = C'$.

- Assume $\Sigma \vdash \sigma y = \sigma y'$. Since we have $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$, $\sigma y = \sigma y'$. Then $\sigma R' = \sigma R$, so $\mathcal{D}'$ obtained from $\mathcal{D}$ by relabelling η with R' is a derivation.
- Otherwise, $\Sigma \vdash \sigma y \neq \sigma y'$ and thus $\sigma\text{nounif}(y, y') \in \mathcal{T}_{\text{facts}}$. Let $\mathcal{D}'$ be obtained by relabelling the node η with the clause $\text{att}'(x, y) \wedge \text{att}'(x, y') \wedge \text{nounif}(y, y') \rightarrow \text{bad}$ (1), adding the son $\sigma\text{nounif}(y, y')$, and returning the subtree with root η. Since $\text{att}'(x, y) \in H$, we have $\sigma\text{att}'(x, y) \in \sigma H \subseteq H'$, and similarly for $\text{att}'(x, y')$. Thus, $\mathcal{D}'$ is a derivation of bad. □

Proof (for *elimGVar* $\circ$ *swap* $\circ$ *unify***)** Let $R = H \wedge F \rightarrow C$ be the clause modified by *elimGVar* $\circ$ *swap* $\circ$ *unify*. We show that if $\sigma F \in \mathcal{T}_{\text{facts}}$, then *elimGVar* $\circ$ *swap* $\circ$ *unify* replaces R with $R' = H \wedge F'_1 \wedge \ldots \wedge F'_n \rightarrow C$, and $\sigma F'_1, \ldots, \sigma F'_n \in \mathcal{T}_{\text{facts}}$.

It is easy to infer the lemma from this property. Indeed, let H' be the multiset of labels of outgoing edges of η and C' the label of its incoming edge. Since $\mathcal{D}$ is a derivation, there exists σ such that $\sigma H \wedge \sigma F \subseteq H'$, and $\sigma C = C'$. Then σF is derived by a son of η, so $\sigma F \in \mathcal{T}_{\text{facts}}$. Then by the above property $\sigma F'_1, \ldots, \sigma F'_n \in \mathcal{T}_{\text{facts}}$, and $\mathcal{D}'$ obtained from $\mathcal{D}$ by relabelling η with $R' = H \wedge F'_1 \wedge \ldots \wedge F'_n \rightarrow C$ and replacing σF with $\sigma F'_1, \ldots, \sigma F'_n$ as sons of η is also a derivation.

- We now prove that *unify* replaces $F = \text{nounif}(p, p')$ with $F'_1 \wedge \ldots \wedge F'_n$ such that, if $\sigma F \in \mathcal{T}_{\text{facts}}$, then $\sigma F'_1, \ldots, \sigma F'_n \in \mathcal{T}_{\text{facts}}$.

  By definition of nounif, $\sigma F \in \mathcal{T}_{\text{facts}}$ if and only if there exists no closed substitution σ' with domain *GVar* such that $\Sigma \vdash \sigma'\sigma p = \sigma'\sigma p'$. By Lemma 32, $\Sigma \vdash \sigma'\sigma p = \sigma'\sigma p'$ if and only if there exist $N, N', \sigma'', \sigma_u$ such that $\text{addeval}(p, p') \Downarrow' ((N, N'), \sigma'')$, σ_u is the most general unifier of N and N', and for all $x \in fv(M, M')$, $\Sigma \vdash \sigma\sigma' x = \sigma\sigma'\sigma_u\sigma'' x$. The fact F is replaced with $F'_1, \ldots, F'_n$, where $F'_j = \text{nounif}(p_j, p'_j) = \text{nounif}((x^j_1, \ldots, x^j_{k_j}), \sigma_u\sigma''(x^j_1, \ldots, x^j_{k_j}))$ for each $\sigma_u\sigma''$ obtained as above. So $\Sigma \vdash \sigma'\sigma p = \sigma'\sigma p'$ if and only if there exists $j \in \{1, \ldots, n\}$ such that $\Sigma \vdash \sigma'\sigma p_j = \sigma'\sigma p'_j$. So $\sigma F \in \mathcal{T}_{\text{facts}}$ if and only if $\sigma F'_1, \ldots, \sigma F'_n \in \mathcal{T}_{\text{facts}}$. This equivalence implies the result.

- Next, we show that *swap* replaces $F = \text{nounif}(p_1, p_2)$ with $F' = \text{nounif}(p'_1, p'_2)$ such that, if $\sigma F \in \mathcal{T}_{\text{facts}}$, then $\sigma F' \in \mathcal{T}_{\text{facts}}$.

  We can easily show that for all σ' with domain $\mathit{GVar} \cup \mathit{Var}$, $\Sigma \vdash \sigma' p_1 = \sigma' p_2$ if and only if $\Sigma \vdash \sigma' p'_1 = \sigma' p'_2$. This equivalence yields the result.

- Finally, we show that *elimGVar* replaces $F = \text{nounif}((\mathsf{g}, p_1, \ldots, p_n), (p'_0, \ldots, p'_n))$ (where $\mathsf{g} \in \mathit{GVar}$) with $F' = \text{nounif}((p_1, \ldots, p_n), (p'_1, \ldots, p'_n))$ such that, if $\sigma F \in \mathcal{T}_{\text{facts}}$, then $\sigma F' \in \mathcal{T}_{\text{facts}}$.

Assume $\sigma F \in \mathcal{T}_{\text{facts}}$. Then there exists no σ' with domain *GVar* such that $\Sigma \vdash \sigma'\sigma(\mathsf{g}, p_1, \ldots, p_n) = \sigma'\sigma(p'_0, \ldots, p'_n)$. So there exists no σ'_1 such that $\Sigma \vdash \sigma'_1\sigma(p_1, \ldots, p_n) = \sigma'_1\sigma(p'_1, \ldots, p'_n)$. Indeed, if σ'_1 existed, $\sigma' = \sigma'_1\{\sigma p'_0/\mathsf{g}\}$ would contradict the non-existence of σ'. (Note that g does not occur elsewhere in F, because F is obtained after applying *unify* and *swap*.) Then $\sigma F' \in \mathcal{T}_{\text{facts}}$. □

Proof (for *elimnouniffalse***)** Let $F = \text{nounif}((), ())$. For all σ, $\sigma F \notin \mathcal{T}_{\text{facts}}$. So $R = H \wedge F \to C$ cannot be the label of a node in a derivation $\mathcal{D}$. (Hence *elimnouniffalse* may harmlessly remove R.) □

Proof (for *elimdup***)** The result is obvious: the hypotheses of R' are included in the hypotheses of R, so $R' \sqsupseteq R$. □

Proof (for *elimattx***)** The result is obvious: the hypotheses of R' are included in the hypotheses of R, so $R' \sqsupseteq R$. □

Proof (for *elimtaut***)** The result is obvious: we remove η and replace it with one of its subtrees. □

Proof (for *simplify***)** We apply Lemma 36 for every simplification function that defines *simplify*. □

Theorem 5 *If* $\mathsf{saturate}(\mathcal{R}_0)$ *terminates and there is a derivation* $\mathcal{D}$ *of* bad *from* $\mathcal{R}_0$ *with* $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$*, then there is a derivation* $\mathcal{D}'$ *of* bad *from* $\mathsf{saturate}(\mathcal{R}_0)$ *with* $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{D}')$*.*

The key idea of the proof is to replace clauses as allowed by the previous lemmas. When the replacement terminates, we can show that all clauses are in $\mathsf{saturate}(\mathcal{R}_0)$. We show the termination using the decrease of the number of nodes of the derivation not in $\mathcal{T}_{\text{facts}}$.

Proof Let us consider a derivation $\mathcal{D}$ of bad from $\mathcal{R}_0$ such that $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$. (The property $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$ is preserved by the transformations of the derivation described below: these transformations do not introduce new non-nounif intermediately derived facts.)

For each clause R in $\mathcal{R}_0$, for each $R'' \in \mathit{simplify}(R)$, there exists a clause R' in $\mathcal{R}$ such that $R' \sqsupseteq R''$ (Lemma 33, Property 1). Assume that there exists a node labelled R in this derivation. By Lemma 36, we can replace R with some $R'' \in \mathit{simplify}(R) \cup \{(1), (2)\}$. Clauses (1) and (2) are subsumed by some clause in $\mathcal{R}$, since they are obtained by simplification from (Rt), resp. (Rt′) for $g = $ equals. So, in all cases, there exists $R' \in \mathcal{R}$ such that $R' \sqsupseteq R''$. By Lemma 35, we can replace R'' with R'. Therefore, we can replace nodes labelled by R with nodes labelled by R'. This way, we obtain a derivation of bad from $\mathcal{R}$.

Next, we build a derivation of bad from $\mathcal{R}_1$, where $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$.

Consider a derivation $\mathcal{D}$ of bad from $\mathcal{R}$ such that $\text{nf}'_{\mathcal{S},\Sigma}(\mathcal{D})$. If $\mathcal{D}$ contains a node labelled by a clause not in $\mathcal{R}_1 \cup \mathcal{T}_{\text{facts}}$, we can transform $\mathcal{D}$ as follows. Let η' be a lowest node of $\mathcal{D}$ labelled by a clause not in $\mathcal{R}_1 \cup \mathcal{T}_{\text{facts}}$. Then all sons of η' are labelled by clauses in $\mathcal{R}_1 \cup \mathcal{T}_{\text{facts}}$. Let R' be the clause labelling η'. Since $R' \notin \mathcal{R}_1 \cup \mathcal{T}_{\text{facts}}$, $\mathsf{sel}(R') \neq \emptyset$. Take $F_0 \in \mathsf{sel}(R')$. By Lemma 34, there exists a son η of η' labelled R, such that $R \circ_{F_0} R'$ is defined. Since all sons of η' are labelled by clauses in $\mathcal{R}_1 \cup \mathcal{T}_{\text{facts}}$, $R \in \mathcal{R}_1 \cup \mathcal{T}_{\text{facts}}$. Moreover, by definition of the selection function, F_0 is not a nounif fact, so $R \notin \mathcal{T}_{\text{facts}}$, so $R \in \mathcal{R}_1$, hence $\mathsf{sel}(R) = \emptyset$ and $R \in \mathcal{R}$. By Lemma 34, we can replace η and η' with η'' labelled by $R \circ_{F_0} R'$. By Lemma 36, we can replace $R \circ_{F_0} R'$ with some $R''' \in \mathit{simplify}(R \circ_{F_0} R') \cup \{(1), (2)\}$. By Lemma 33, Property 2, for each $R''' \in \mathit{simplify}(R \circ_{F_0} R')$, there exists $R'' \in \mathcal{R}$ such that $R'' \sqsupseteq R'''$; as noted above, this is also true for (1) and (2) so for all $R''' \in \mathit{simplify}(R \circ_{F_0} R') \cup \{(1), (2)\}$, there exists $R'' \in \mathcal{R}$ such that $R'' \sqsupseteq R'''$. By Lemma 35, we can replace R''' with R'', and we obtain a derivation $\mathcal{D}'$ of bad

from $\mathcal{R}$, such that $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{D}')$ and $\mathcal{D}'$ contains fewer nodes not in $\mathcal{T}_{\mathrm{facts}}$ as $\mathcal{D}$ (since the resolution of two clauses removes one node, and simplifications do not add nodes not in $\mathcal{T}_{\mathrm{facts}}$).

Since the number of nodes not in $\mathcal{T}_{\mathrm{facts}}$ strictly decreases, this transformation process terminates.

When we cannot perform this transformation any more, all nodes of the derivation are labelled by clauses in $\mathcal{R}_1 \cup \mathcal{T}_{\mathrm{facts}}$, hence we have obtained a derivation $\mathcal{D}'$ of bad from $\mathcal{R}_1$ such that $\mathrm{nf}'_{\mathcal{S},\Sigma}(\mathcal{D}')$. □

Proof (of Theorem 4) If bad is derivable from $\mathcal{R}_{P_0}$ then it is derivable from $\mathcal{R}_{P_0}$ by a derivation that satisfies $\mathrm{nf}'_{\mathcal{S},\Sigma}$ (by Lemma 3), then it is derivable from $\mathsf{saturate}(\mathcal{R}_{P_0})$ by a derivation that satisfies $\mathrm{nf}'_{\mathcal{S},\Sigma}$ (by Theorem 5), then $\mathsf{saturate}(\mathcal{R}_{P_0})$ contains a clause of the form $H \rightarrow \mathrm{bad}$. □

A Computationally Sound Mechanized Prover for Security Protocols

Bruno Blanchet*†

Abstract

We present a new mechanized prover for secrecy properties of security protocols. In contrast to most previous provers, our tool does not rely on the Dolev-Yao model, but on the computational model. It produces proofs presented as sequences of games; these games are formalized in a probabilistic polynomial-time process calculus. Our tool provides a generic method for specifying security properties of the cryptographic primitives, which can handle shared-key and public-key encryption, signatures, message authentication codes, and hash functions. Our tool produces proofs valid for a number of sessions polynomial in the security parameter, in the presence of an active adversary. We have implemented our tool and tested it on a number of examples of protocols from the literature.

1 Introduction

There exist two main approaches for analyzing security protocols. In the computational model, messages are bitstrings, and the adversary is a probabilistic polynomial-time Turing machine. This model is close to the real execution of protocols, but the proofs are usually manual and informal. In contrast, in the formal, Dolev-Yao model, cryptographic primitives are considered as perfect blackboxes, modeled by function symbols in an algebra of terms, possibly with equations. The adversary can compute using these blackboxes. This abstract model makes it possible to build automatic verification tools, but the security proofs are in general not sound with respect to the computational model.

Since the seminal paper by Abadi and Rogaway [3], there has been much interest in relating both frameworks (see for example [1, 9, 12, 23, 27, 28, 37, 38]), to show the soundness of the Dolev-Yao model with respect to the computational model, and thus obtain automatic proofs of protocols in the computational model. However, this approach has limitations: since the computational and Dolev-Yao models do not correspond exactly, additional hypotheses are necessary in order to guarantee soundness. (For example, key cycles have to be excluded, or a specific security definition of encryption is needed [5].)

In this paper, we propose a different approach for automatically proving protocols in the computational model: we have built a mechanized prover that works directly in the computational model, without considering the Dolev-Yao model. Our tool produces proofs valid for a number of sessions polynomial in the security parameter, in the presence of an active adversary. These proofs are presented as sequences of games, as used by cryptographers [16,44,45]: the initial game represents the protocol to prove; the goal is to show that the probability of breaking a certain security property (secrecy in this paper) is negligible in this game; intermediate games are obtained each from the previous one by transformations such that the difference of probability between consecutive games is negligible; the final game is such that the desired probability is obviously negligible from the form of the game. The desired probability is then negligible in the initial game.

We represent games in a process calculus. This calculus is inspired by the pi-calculus and by the calculi of [33, 34, 39] and of [32]. In this calculus, messages are bitstrings, and cryptographic primitives are functions from bitstrings to bitstrings. The calculus has a probabilistic semantics, and all processes run in polynomial time. The main tool for specifying security properties is observational equivalence: Q is observationally equivalent to Q', $Q \approx Q'$, when the adversary has a negligible probability of distinguishing Q from Q'. With respect to previous calculi mentioned above, our calculus introduces an important novelty which is key for the automatic proof of security protocols: the values of all variables during the execution of a process are stored in arrays. For instance, $x[i]$ is the value of x in the i-th copy of the process that defines x. Arrays replace lists often used by cryptographers in their manual proofs of protocols. For example, consider the definition of security of a message authentication code (MAC). Informally, this definition says that the adversary has a negligible probability of forging a MAC, that is, that all correct MACs have been computed by calling the MAC oracle. So, in cryptographic proofs, one defines a list containing the arguments of calls to the MAC oracle, and when checking a MAC of a message m, one can additionally check that m is in this list, with a negligible change in probability. In our calculus, the arguments of the MAC oracle are stored in arrays, and we perform a lookup in these arrays in order to find the message m. Arrays make it easier to automate proofs since they are always present in the calculus: one does not need to add explicit instructions to insert values in them, in contrast to the lists used in manual proofs. Therefore, many trivially sound but difficult to automate syntactic transformations disappear. Furthermore, relations between elements of arrays can easily be expressed by equalities, possibly involving computations on array indices.

Our prover relies on a collection of game transformations, in order to transform the initial protocol into a game on which the desired security property is obvious. The most important kind of transformations exploits the definition of security of cryptographic primitives in order to obtain a simpler game. As described in Section 3.2, these transformations can be specified

*B. Blanchet is with CNRS, École Normale Supérieure, Paris, France E-mail: blanchet@di.ens.fr

†A short version of this paper appears at IEEE Symposium on Security and Privacy, Oakland, California, May 2006.

in a generic way: we represent the definition of security of each cryptographic primitive by an observational equivalence $L \approx R$, where the processes L and R encode functions: they input the arguments of the function and send its result back. Then, the prover can automatically transform a process Q that calls the functions of L (more precisely, contains as subterms terms that perform the same computations as functions of L) into a process Q' that calls the functions of R instead. We have used this technique to specify several variants of shared-key and public-key encryption, signature, message authentication codes, and hash functions, simply by giving the appropriate equivalence $L \approx R$ to the prover. Other game transformations are syntactic transformations, used in order to be able to apply the definition of cryptographic primitives, or to simplify the game obtained after applying these definitions.

In order to prove protocols, these game transformations are organized using a proof strategy based on advice: when a transformation fails, it suggests other transformations that should be applied before, in order to enable the desired transformation. Thanks to this strategy, protocols can often be proved in a fully automatic way. For delicate cases, our prover has an interactive mode, in which the user can manually specify the transformations to apply. It is usually sufficient to specify a few transformations coming from the security definitions of primitives, by indicating the concerned cryptographic primitive and the concerned secret key if any; the prover infers the intermediate syntactic transformations by the advice strategy. This mode is helpful for proving some public-key protocols, in which several security definitions of primitives can be applied, but only one leads to a proof of the protocol. Importantly, our prover is always sound: whatever indications the user gives, when the prover shows a security property of the protocol, the property indeed holds assuming the given hypotheses on the cryptographic primitives.

Our prover CryptoVerif has been implemented in Ocaml (17300 lines of code for version 1.03 of CryptoVerif) and is available at `http://www.di.ens.fr/~blanchet/cryptoc-eng.html`.

1.1 Outline

The next section presents our process calculus for representing games. Section 3 describes the game transformations that we use for proving protocols. Section 4 gives criteria for proving secrecy properties of protocols. Section 5 explains how the prover chooses which transformation to apply at each point. Section 6 presents our experimental results. Section 7 discusses related work and Section 8 concludes. The appendices contain additional formal details, proof sketches, and details on the modeling of some cryptographic primitives.

1.2 Notations

We recall the following standard notations. We denote by $\{M_1/x_1, \dots, M_m/x_m\}$ the substitution that replaces x_j with M_j for each $j \leq m$. The cardinal of a set or multiset S is denoted by $|S|$. If S is a finite set, $x \overset{R}{\leftarrow} S$ chooses a random element uniformly in S and assigns it to x. If $\mathcal{A}$ is a probabilistic algorithm, $x \leftarrow \mathcal{A}(x_1, \dots, x_m)$ denotes the experiment of choosing random coins r and assigning to x the result of running $\mathcal{A}(x_1, \dots, x_m)$ with coins r. Otherwise, $x \leftarrow M$ is a simple assignment statement.

$M, N ::=$	terms
$\quad i$	replication index
$\quad x[M_1, \dots, M_m]$	variable access
$\quad f(M_1, \dots, M_m)$	function application
$Q ::=$	input process
$\quad 0$	nil
$\quad Q \mid Q'$	parallel composition
$\quad !^{i \leq n} Q$	replication n times
$\quad \mathsf{newChannel}\ c; Q$	channel restriction
$\quad c[M_1, \dots, M_l](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k); P$	input
$P ::=$	output process
$\quad \overline{c[M_1, \dots, M_l]}\langle N_1, \dots, N_k \rangle; Q$	output
$\quad \mathsf{new}\ x[i_1, \dots, i_m] : T; P$	random number
$\quad \mathsf{let}\ x[i_1, \dots, i_m] : T = M\ \mathsf{in}\ P$	assignment
$\quad \mathsf{if}\ \mathsf{defined}(M_1, \dots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditional
$\quad \mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ $\mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)$ $\mathsf{else}\ P$	array lookup

Figure 1: Syntax of the process calculus

2 A Calculus for Games

2.1 Syntax and Informal Semantics

The syntax of our calculus is summarized in Figure 1. This calculus was inspired by the pi calculus and by the calculi of [33, 34, 39] and of [32]. We denote by η the security parameter, which determines in particular the length of keys.

This calculus assumes a countable set of channel names, denoted by c. There is a mapping maxlen_η from channels to integers, such that $\mathrm{maxlen}_\eta(c)$ is the maximum length of a message sent on channel c. Longer messages are truncated. For all c, $\mathrm{maxlen}_\eta(c)$ is polynomial in η. (This is key to guaranteeing that all processes run in probabilistic polynomial time.)

Our calculus also uses parameters, denoted by n, which correspond to integer values polynomial in the security parameter. So, denoting by $I_\eta(n)$ the interpretation of n for a given value of the security parameter η, $I_\eta(n)$ is a polynomially bounded, efficiently computable function of η.

Our calculus also uses types, denoted by T. For each value of the security parameter η, each type corresponds to a subset $I_\eta(T)$ of *Bitstring* $\cup \{\bot\}$ where *Bitstring* is the set of all bitstrings and $\bot$ is a special symbol. The set $I_\eta(T)$ must be recognizable in polynomial time, that is, there exists an algorithm that decides whether $x \in I_\eta(T)$ in time polynomial in the length of x and the value of η. Let *fixed-length* types be types T such that

$I_\eta(T)$ is the set of all bitstrings of a certain length, this length being a function of η bounded by a polynomial. Let *large* types be types T such that $\frac{1}{|I_\eta(T)|}$ is negligible. ($f(\eta)$ is *negligible* when for all polynomials q, there exists $\eta_o \in \mathbb{N}$ such that for all $\eta > \eta_0$, $f(\eta) \leq \frac{1}{q(\eta)}$.) Particular types are predefined: *bool*, such that $I_\eta(bool) = \{\text{true}, \text{false}\}$, where false is 0 and true is 1; *bitstring*, such that $I_\eta(bitstring) = Bitstring$; $bitstring_\perp$ such that $I_\eta(bitstring_\perp) = Bitstring \cup \{\perp\}$; $[1, n]$ where n is a parameter, such that $I_\eta([1, n]) = [1, I_\eta(n)]$. (We consider integers as bitstrings without leading zeroes.)

The calculus also uses function symbols f. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \rightarrow T$. For each value of η, each function symbol f corresponds to a function $I_\eta(f)$ from $I_\eta(T_1) \times \ldots \times I_\eta(T_m)$ to $I_\eta(T)$, such that $I_\eta(f)(x_1, \ldots, x_m)$ is computable in polynomial time in the lengths of $x_1, \ldots, x_m$ and the value of η. Particular functions are predefined, and some of them use the infix notation: $M = N$ for the equality test, $M \neq N$ for the inequality test (both taking two values of the same type T and returning a value of type *bool*), $M \vee N$ for the boolean or, $M \wedge N$ for the boolean and, $\neg M$ for the boolean negation (taking and returning values of type *bool*).

In this calculus, terms represent computations on bitstrings. The replication index i is an integer which serves in distinguishing different copies of a replicated process $!^{i \leq n}$. (Replication indices are typically used as array indices.) The variable access $x[M_1, \ldots, M_m]$ returns the content of the cell of indices $M_1, \ldots, M_m$ of the m-dimensional array variable x. We use x, y, z, u as variable names. The function application $f(M_1, \ldots, M_m)$ returns the result of applying function f to $M_1, \ldots, M_m$.

The calculus distinguishes two kinds of processes: input processes Q are ready to receive a message on a channel; output processes P output a message on a channel after executing some internal computations. The input process 0 does nothing; $Q \mid Q'$ is the parallel composition of Q and Q'; $!^{i \leq n} Q$ represents n copies of Q in parallel, each with a different value of $i \in [1, n]$; newChannel $c; Q$ creates a new private channel c and executes Q; the semantics of the input $c[M_1, \ldots, M_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$ will be explained below together with the semantics of the output.

The output process new $x[i_1, \ldots, i_m] : T; P$ chooses a new random number uniformly in $I_\eta(T)$, stores it in $x[i_1, \ldots, i_m]$, and executes P. (The type T must be a fixed-length type, because probabilistic polynomial-time Turing machines can choose random numbers uniformly only in such types.) Function symbols represent deterministic functions, so all random numbers must be chosen by new $x[i_1, \ldots, i_m] : T$. Deterministic functions make automatic syntactic manipulations easier: we can duplicate a term without changing its value. The process let $x[i_1, \ldots, i_m] : T = M$ in P stores the bitstring value of M (which must be in $I_\eta(T)$) in $x[i_1, \ldots, i_m]$ and executes P. Next, we explain the process find $(\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P, where $\widetilde{i}$ denotes a tuple $i_1, \ldots, i_{m'}$. The order and array indices on tuples are taken component-wise, so for instance, $u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ can be further abbreviated $\widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$. A simple example is the following: find $u \leq n$ suchthat defined$(x[u]) \wedge x[u] = a$ then P' else P tries to find an index u such that $x[u]$ is defined and $x[u] = a$, and when such a u is found, it executes P' with that value of u; otherwise, it executes P. In other words, this find construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. Therefore, the find construct allows us to access arrays, which is key for our purpose. More generally, find $u_1[\widetilde{i}] \leq n_1, \ldots, u_m[\widetilde{i}] \leq n_m$ suchthat defined$(M_1, \ldots, M_l) \wedge M$ then P' else P tries to find values of $u_1, \ldots, u_m$ for which $M_1, \ldots, M_l$ are defined and M is true. In case of success, it executes P'. In case of failure, it executes P. This is further generalized to m branches: find $(\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \ldots, u_{jm_j}$ for which $M_{j1}, \ldots, M_{jl_j}$ are defined and M_j is true. In case of success, it executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ for each j and each value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ in $[1, n_{j1}] \times \ldots \times [1, n_{jm_j}]$. If none of these conditions is true, it executes P. Otherwise, it chooses randomly with uniform[1] probability one j and one value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ such that the corresponding condition is true and executes P_j. The conditional if defined$(M_1, \ldots, M_l) \wedge M$ then P else P' executes P if $M_1, \ldots, M_l$ are defined and M evaluates to true. Otherwise, it executes P'. This conditional is defined as syntactic sugar for find suchthat defined$(M_1, \ldots, M_l) \wedge M$ then P else P'. The conjunct defined$(M_1, \ldots, M_l)$ can be omitted when $l = 0$ and M can be omitted when it is true.

Finally, let us explain the output $\overline{c[M_1, \ldots, M_l]}\langle N_1, \ldots, N_k \rangle; Q$. A channel $c[M_1, \ldots, M_l]$ consists of both a channel name c and a tuple of terms $M_1, \ldots, M_l$. Channel names c allow us to define private channels to which the adversary can never have access, by newChannel c. (This is useful in the proofs, although all channels of protocols are often public.) Terms $M_1, \ldots, M_l$ are intuitively analogous to IP addresses and ports which are numbers that the adversary may guess. A semantic configuration always consists of a single output process (the process currently being executed) and several input processes. When the output process executes $\overline{c[M_1, \ldots, M_l]}\langle N_1, \ldots, N_k \rangle; Q$, one looks for an input on channel $c[M'_l \ldots, M'_l]$, where $M'_1, \ldots, M'_l$ evaluate to the same bitstrings as $M_1, \ldots, M_l$, and with the same arity k, in the available input processes. If no such input process is found, the process blocks. Otherwise, one such input process $c[M'_1, \ldots, M'_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$ is chosen randomly with uniform probability. The communication is then executed: for each $j \leq k$, the output message N_j is evaluated, its result is truncated to length $\mathrm{maxlen}_\eta(c)$, the obtained bitstring is stored in $x_j[\widetilde{i}]$ if it is in $I_\eta(T_j)$ (otherwise the process blocks).

[1] A probabilistic polynomial-time Turing machine can choose a random number uniformly in a set of cardinal m only when m is a power of 2. When m is not a power of 2, there exist approximate algorithms: for example, in order to obtain a random integer in $[0, m-1]$, we can choose a random integer r uniformly among $[0, 2^k - 1]$ for a certain k large enough and return $r \mod m$. The distribution can be made as close as we wish to the uniform distribution by choosing k large enough.

Finally, the output process P that follows the input is executed. The input process Q that follows the output is stored in the available input processes for future execution. Note that the syntax requires an output to be followed by an input process, as in [32]. If one needs to output several messages consecutively, one can simply insert fictitious inputs between the outputs. The adversary can then schedule the outputs by sending messages to these inputs.

Using different channels for each input and output allows the adversary to control the network. For instance, we may write $!^{i \le n} c[i](x[i] : T) \ldots \overline{c'[i]}\langle M \rangle \ldots$ The adversary can then decide which copy of the replicated process receives its message, simply by sending it on $c[i]$ for the appropriate value of i.

An else branch of find or if may be omitted when it is else $\overline{yield}\langle\rangle; 0$. (Note that "else 0" would not be syntactically correct.) A trailing 0 after an output may be omitted.

Variables can be defined by assignments, inputs, restrictions, and array lookups. The *current replication indices* at a certain program point in a process are $i_1, \ldots, i_m$ where the replications above the considered program point are $!^{i_1 \le n_1} \ldots !^{i_m \le n_m}$. We often abbreviate $x[i_1, \ldots, i_m]$ by x when $i_1, \ldots, i_m$ are the current replication indices, but it should be kept in mind that this is only an abbreviation. Variables defined under a replication must be arrays: for example $!^{i_1 \le n_1} \ldots !^{i_m \le n_m}$let $x[i_1, \ldots, i_m] : T = M$ in ... More formally, we require the following invariant:

Invariant 1 (Single definition) The process Q_0 satisfies Invariant 1 if and only if

1. in every definition of $x[i_1, \ldots, i_m]$ in Q_0, the indices $i_1, \ldots, i_m$ of x are the current replication indices at that definition, and
2. two different definitions of the same variable x in Q_0 are in different branches of a find (or if).

Invariant 1 guarantees that each variable is assigned at most once for each value of its indices. (Indeed, item 2 shows that only one definition of each variable can be executed for given indices in each trace.)

Invariant 2 (Defined variables) The process Q_0 satisfies Invariant 2 if and only if every occurrence of a variable access $x[M_1, \ldots, M_m]$ in Q_0 is either

- syntactically under the definition of $x[M_1, \ldots, M_m]$ (in which case $M_1, \ldots, M_m$ are in fact the current replication indices at the definition of x);
- or in a defined condition in a find process;
- or in M'_j or P_j in a process of the form find $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[\widetilde{i}] \le \widetilde{n_j}$ suchthat defined$(M'_{j1}, \ldots, M'_{jl_j}) \wedge M'_j$ then $P_j)$ else P where for some $k \le l_j$, $x[M_1, \ldots, M_m]$ is a subterm of M'_{jk}.

Invariant 2 guarantees that variables can be accessed only when they have been initialized. It checks that the definition of the variable access is either in scope (first item) or checked by a find (last item). Both invariants are checked by the prover for the initial game and preserved by all game transformations.

We say that a function $f : T_1 \times \ldots \times T_m \to T$ is *poly-injective* when it is injective and its inverses can be computed in polynomial time, that is, there exist functions $f_j^{-1} : T \to T_j$ $(1 \le j \le m)$ such that $f_j^{-1}(f(x_1, \ldots, x_m)) = x_j$ and f_j^{-1} can be computed in polynomial time in the length of $f(x_1, \ldots, x_m)$ and in the security parameter. When f is poly-injective, we define a pattern matching construct let $f(x_1, \ldots, x_m) = M$ in P else Q as an abbreviation for let $y : T = M$ in let $x_1 : T_1 = f_1^{-1}(y)$ in ... let $x_m : T_m = f_m^{-1}(y)$ in if $f(x_1, \ldots, x_m) = y$ then P else Q. We naturally generalize this construct to let $N = M$ in P else Q where N is built from poly-injective functions and variables.

We denote by $\mathrm{var}(P)$ the set of variables that occur in P and by $\mathrm{fc}(P)$ the set of free channels of P. (We use similar notations for input processes.)

2.2 Example

Let us introduce two cryptographic primitives that we use below.

Definition 1 Let T_{mr}, T_{mk}, and T_{ms} be types that correspond intuitively to random seeds, keys, and message authentication codes, respectively; T_{mr} is a fixed-length type. A message authentication code [15] consists of three function symbols:

- $\mathrm{mkgen} : T_{mr} \to T_{mk}$ where $I_\eta(\mathrm{mkgen}) = \mathrm{mkgen}_\eta$ is the key generation algorithm taking as argument a random bitstring and returning a key. (Usually, mkgen is a randomized algorithm; here, since we separate the choice of random numbers from computation, mkgen takes an additional argument representing the random coins.)
- $\mathrm{mac} : \mathit{bitstring} \times T_{mk} \to T_{ms}$ where $I_\eta(\mathrm{mac}) = \mathrm{mac}_\eta$ is the MAC algorithm taking as argument a message and a key, and returning the corresponding tag. (We assume here that mac is deterministic; we could easily encode a randomized mac by adding an additional argument as for mkgen.)
- $\mathrm{check} : \mathit{bitstring} \times T_{mk} \times T_{ms} \to \mathit{bool}$ where $I_\eta(\mathrm{check}) = \mathrm{check}_\eta$ is a checking algorithm such that $\mathrm{check}_\eta(m, k, t) = \mathrm{true}$ if and only if t is a valid MAC of message m under key k. (Since mac is deterministic, $\mathrm{check}_\eta(m, k, t)$ is typically $\mathrm{mac}_\eta(m, k) = t$.)

We have $\forall m \in \mathit{Bitstring}, \forall r \in I_\eta(T_{mr}), \mathrm{check}_\eta(m, \mathrm{mkgen}_\eta(r), \mathrm{mac}_\eta(m, \mathrm{mkgen}_\eta(r))) = \mathrm{true}$.

A MAC is UF-CMA (satisfies unforgeability under chosen message attacks) if and only if for all polynomials q,

$$\max_{\mathcal{A}} \Pr\left[\begin{array}{l} r \xleftarrow{R} I_\eta(T_{mr}); k \leftarrow \mathrm{mkgen}_\eta(r); \\ (m, t) \leftarrow \mathcal{A}^{\mathrm{mac}_\eta(.,k), \mathrm{check}_\eta(.,k,.)} : \mathrm{check}_\eta(m, k, t) \end{array}\right]$$

is negligible, where the adversary $\mathcal{A}$ is any probabilistic Turing machine, running in time $q(\eta)$, with oracle access to $\mathrm{mac}_\eta(., k)$ and $\mathrm{check}_\eta(., k, .)$, and $\mathcal{A}$ has not called $\mathrm{mac}_\eta(., k)$ on message m.

Definition 2 Let T_r and T'_r be fixed-length types; let T_k and T_e be types. A symmetric encryption scheme [13] consists of

three function symbols $\mathrm{kgen} : T_r \rightarrow T_k$, $\mathrm{enc} : \mathit{bitstring} \times T_k \times T'_r \rightarrow T_e$, and $\mathrm{dec} : T_e \times T_k \rightarrow \mathit{bitstring}_\perp$, with $I_\eta(\mathrm{kgen}) = \mathrm{kgen}_\eta$, $I_\eta(\mathrm{enc}) = \mathrm{enc}_\eta$, $I_\eta(\mathrm{dec}) = \mathrm{dec}_\eta$, such that for all $m \in \mathit{Bitstring}$, $r \in I_\eta(T_r)$, and $r' \in I_\eta(T'_r)$, $\mathrm{dec}_\eta(\mathrm{enc}_\eta(m, \mathrm{kgen}_\eta(r), r'), \mathrm{kgen}_\eta(r)) = m$.

Let $LR(x, y, b) = x$ if $b = 0$ and $LR(x, y, b) = y$ if $b = 1$, defined only when x and y are bitstrings of the same length. A symmetric encryption scheme is IND-CPA (satisfies indistinguishability under chosen plaintext attacks) if and only if for all polynomials q,

$$\max_{\mathcal{A}} 2 \Pr \begin{bmatrix} b \stackrel{R}{\leftarrow} \{0,1\}; r \stackrel{R}{\leftarrow} I_\eta(T_r); k \leftarrow \mathrm{kgen}_\eta(r); \\ b' \leftarrow \mathcal{A}^{r' \stackrel{R}{\leftarrow} I_\eta(T'_r); \mathrm{enc}_\eta(LR(.,.,b),k,r')} : b' = b \end{bmatrix} - 1$$

is negligible, where the adversary $\mathcal{A}$ is any probabilistic Turing machine, running in time $q(\eta)$, with oracle access to the left-right encryption algorithm which, given two bitstrings a_0 and a_1 of the same length, returns $r' \stackrel{R}{\leftarrow} I_\eta(T'_r); \mathrm{enc}_\eta(LR(a_0, a_1, b), k, r')$, that is, encrypts a_0 when $b = 0$ and a_1 when $b = 1$.

Example 1 Let us consider the following trivial protocol:

$$A \rightarrow B : e, \mathrm{mac}(e, x_{mk}) \quad \text{where } e = \mathrm{enc}(x'_k, x_k, x''_r) \text{ and } x''_r, x'_k \text{ are fresh random numbers}$$

A and B are assumed to share a key x_k for a symmetric encryption scheme and a key x_{mk} for a message authentication code. A creates a fresh key x'_k and sends it encrypted under x_k to B. A MAC is appended to the message, in order to guarantee integrity. The goal of the protocol is that x'_k should be a secret key shared between A and B. This protocol can be modeled in our calculus by the following process Q_0:

$$\begin{aligned}
Q_0 = {} & \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in} \\
& \mathsf{new}\ x'_r : T_{mr}; \mathsf{let}\ x_{mk} : T_{mk} = \mathrm{mkgen}(x'_r)\ \mathsf{in} \\
& \overline{c}\langle\rangle; (Q_A \mid Q_B) \\
Q_A = {} & !^{i \le n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x''_r : T'_r; \\
& \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x''_r)\ \mathsf{in} \\
& \overline{c_A[i]}\langle x_m, \mathrm{mac}(x_m, x_{mk})\rangle \\
Q_B = {} & !^{i' \le n} c_B[i'](x'_m, x_{ma}); \\
& \mathsf{if}\ \mathrm{check}(x'_m, x_{mk}, x_{ma})\ \mathsf{then} \\
& \mathsf{let}\ \mathrm{i}_\perp(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle
\end{aligned}$$

When Q_0 receives a message on channel start, it begins execution: it generates the keys x_k and x_{mk} by choosing random coins x_r and $x_{r'}$ and applying the appropriate key generation algorithms. Then it yields control to the context (the adversary), by outputting on channel c. After this output, n copies of processes for A and B are ready to be executed, when the context outputs on channels $c_A[i]$ or $c_B[i]$ respectively. In a session that runs as expected, the context first sends a message on $c_A[i]$. Then Q_A creates a fresh key x'_k (T_k is assumed to be a fixed-length type), encrypts it under x_k with random coins x''_r, computes the MAC under x_{mk} of the ciphertext, and sends the ciphertext and the MAC on $c_A[i]$. The function $\mathrm{k2b} : T_k \rightarrow \mathit{bitstring}$ is the natural injection $I_\eta(\mathrm{k2b})(x) = x$; it is needed only for type conversion. The context is then expected to forward this message on $c_B[i]$. When Q_B receives this message, it checks the MAC, decrypts, and stores the obtained key in x''_k. (The function $\mathrm{i}_\perp : \mathit{bitstring} \rightarrow \mathit{bitstring}_\perp$ is the natural injection; it is useful to check that decryption succeeded.) This key x''_k should be secret.

The context is responsible for forwarding messages from A to B. It can send messages in unexpected ways in order to mount an attack.

Although we use a trivial running example due to length constraints, this example is sufficient to illustrate the main features of our prover. Section 6 presents results obtained on more realistic protocols.

2.3 Type System

We use a type system to check that bitstrings of the proper type are passed to each function and that array indices are used correctly.

To be able to type variable accesses used not under their definition (such accesses are guarded by a find construct), the type-checking algorithm proceeds in two passes. In the first pass, it builds a type environment $\mathcal{E}$, which maps variable names x to types $[1, n_1] \times \ldots \times [1, n_m] \rightarrow T$, where the definition of $x[i_1, \ldots, i_m]$ of type T occurs under replications $!^{i_1 \le n_1}, \ldots, !^{i_m \le n_m}$. The tool checks that all definitions of the same variable x yield the same value of $\mathcal{E}(x)$, so that $\mathcal{E}$ is properly defined.

In the second pass, the process is typechecked in the type environment $\mathcal{E}$ by a simple type system. This type system is detailed in Appendix A. It defines the judgment $\mathcal{E} \vdash Q$, which means that the process Q is well-typed in environment $\mathcal{E}$.

Invariant 3 (Typing) The process Q_0 satisfies Invariant 3 if and only if the type environment $\mathcal{E}$ for Q_0 is well-defined, and $\mathcal{E} \vdash Q_0$.

We require the adversary to be well-typed. This requirement does not restrict its computing power, because it can always define type-cast functions $f : T \rightarrow T'$ to bypass the type system. Similarly, the type system does not restrict the class of protocols that we consider, since the protocol may contain type-cast functions. The type system just makes explicit which set of bitstrings may appear at each point of the protocol.

2.4 Formal Semantics

The semantics is defined by a probabilistic reduction relation formally detailed in Appendix B. The notation $E, M \Downarrow a$ means that the term M evaluates to the bitstring a in environment E. We denote by $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a\rangle]$ the probability that at least one of the outputs of Q on channel c sends the bitstring a. (When c is not free in Q, $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a\rangle] = 0$.)

Our semantics is such that, for each process Q, there exists a probabilistic polynomial time Turing machine that simulates Q. (Processes run in polynomial time since the number of processes created by a replication and the length of messages sent

on channels are bounded by polynomials.) Conversely, our calculus can simulate a probabilistic polynomial-time Turing machine, simply by choosing coins by new and by applying a function symbol defined to perform the same computations as the Turing machine.

2.5 Observational Equivalence

A context is a process containing a hole $[\,]$. An evaluation context C is a context built from $[\,]$, newChannel $c; C$, $Q \mid C$, and $C \mid Q$. We use an evaluation context to represent the adversary. We denote by $C[Q]$ the process obtained by replacing the hole $[\,]$ in the context C with the process Q. Our definition of observational equivalence is adapted from definitions for previous calculi such as [39].

Definition 3 (Observational equivalence) Let Q and Q' be two processes and V a set of variables. Assume that Q and Q' satisfy Invariants 1, 2, and 3 and the variables of V are defined in Q and Q', with the same types.

An evaluation context C is said to be *acceptable* for Q, Q', V if and only if $\mathrm{var}(C) \cap (\mathrm{var}(Q) \cup \mathrm{var}(Q')) \subseteq V$ and $C[Q]$ satisfies Invariants 1, 2, and 3. (Then $C[Q']$ also satisfies these invariants.)

We say that Q and Q' are *observationally equivalent* with public variables V, written $Q \approx^V Q'$, when for all evaluation contexts C acceptable for Q, Q', V, for all channels c and bitstrings a, $|\Pr[C[Q] \rightsquigarrow_\eta \overline{c}\langle a\rangle] - \Pr[C[Q'] \rightsquigarrow_\eta \overline{c}\langle a\rangle]|$ is negligible.

Intuitively, the goal of the adversary represented by context C is to distinguish Q from Q'. When it succeeds, it performs a different output, for example $\overline{c}\langle 0\rangle$ when it has recognized Q and $\overline{c}\langle 1\rangle$ when it has recognized Q'. When $Q \approx^V Q'$, the context has negligible probability of distinguishing Q from Q'.

The unusual requirement on variables of C comes from the presence of arrays and of the associated find construct which gives C direct access to variables of Q and Q': the context C is allowed to access variables of Q and Q' only when they are in V. (In more standard settings, the calculus does not have constructs that allow the context to access variables of Q and Q'.) The following result is not difficult to prove:

Lemma 1 *$\approx^V$ is an equivalence relation, and $Q \approx^V Q'$ implies that $C[Q] \approx^{V'} C[Q']$ for all evaluation contexts C acceptable for Q, Q', V and all $V' \subseteq V \cup (\mathrm{var}(C) \setminus (\mathrm{var}(Q) \cup \mathrm{var}(Q')))$.*

We denote by $Q \approx_0^V Q'$ the particular case in which for all evaluation contexts C acceptable for Q, Q', V, for all channels c and bitstrings a, $\Pr[C[Q] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C[Q'] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$. When V is empty, we write $Q \approx Q'$ instead of $Q \approx^V Q'$ and $Q \approx_0 Q'$ instead of $Q \approx_0^V Q'$.

3 Game Transformations

In this section, we describe the game transformations that allow us to transform the process that represents the initial protocol into a process on which the desired security property can be proved directly, by criteria given in Section 4. These transformations are parametrized by the set V of variables that the context can access. As we shall see in Section 4, V contains variables that we would like to prove secret. (The context will contain test queries that access these variables.) These transformations transform a process Q_0 into a process Q_0' such that $Q_0 \approx^V Q_0'$.

3.1 Syntactic Transformations

RemoveAssign(x): When x is defined by an assignment let $x[i_1, \dots, i_l] : T = M$ in P, we replace x with its value. Precisely, the transformation is performed only when x does not occur in M (non-cyclic assignment). When x has several definitions, we simply replace $x[i_1, \dots, i_l]$ with M in P. (For accesses to x guarded by find, we do not know which definition of x is actually used.) When x has a single definition, we replace everywhere in the game $x[M_1, \dots, M_l]$ with $M\{M_1/i_1, \dots, M_l/i_l\}$. We additionally update the defined conditions of find to preserve Invariant 2 and to make sure that, if a condition of find guarantees that $x[M_1, \dots, M_l]$ is defined in the initial game, then so does the corresponding condition of find in the transformed game. (Essentially, when $y[M_1', \dots, M_{l'}']$ occurs in M, the transformation typically creates new occurrences of $y[M_1'', \dots, M_{l'}'']$ for some $M_1'', \dots, M_{l'}''$, so the condition that $y[M_1'', \dots, M_{l'}'']$ is defined must sometimes be explicitly added to conditions of find in order to preserve Invariant 2.) When $x \in V$, its definition is kept unchanged. Otherwise, when x is not referred to at all after the transformation, we remove the definition of x. When x is referred to only at the root of defined tests, we replace its definition with a constant. (The definition point of x is important, but not its value.)

Example 2 In the process of Example 1, the transformation **RemoveAssign**(x_{mk}) substitutes $\mathrm{mkgen}(x_r')$ for x_{mk} in the whole process and removes the assignment let $x_{mk} : T_{mk} = \mathrm{mkgen}(x_r')$. After this substitution, $\mathrm{mac}(x_m, x_{mk})$ becomes $\mathrm{mac}(x_m, \mathrm{mkgen}(x_r'))$ and $\mathrm{check}(x_m', x_{mk}, x_{ma})$ becomes $\mathrm{check}(x_m', \mathrm{mkgen}(x_r'), x_{ma})$, thus exhibiting terms required in Section 3.2. The situation is similar for **RemoveAssign**(x_k).

SArename(x): The transformation **SArename** (single assignment rename) aims at renaming variables so that each variable has a single definition in the game; this is useful for distinguishing cases depending on which definition of x has set $x[\widetilde{i}]$. This transformation can be applied only when $x \notin V$. When x has $m > 1$ definitions, we rename each definition of x to a different variable $x_1, \dots, x_m$. Terms $x[\widetilde{i}]$ under a definition of $x_j[\widetilde{i}]$ are then replaced with $x_j[\widetilde{i}]$. Each branch of find $FB = \widetilde{u}[\widetilde{i}] \leq \widetilde{n}$ suchthat defined$(M_1', \dots, M_{l'}') \wedge M$ then P where $x[M_1, \dots, M_l]$ is a subterm of some M_k' for $k \leq l'$ is replaced with m branches $FB\{x_j[M_1, \dots, M_l]/x[M_1, \dots, M_l]\}$ for $1 \leq j \leq m$.

Example 3 Consider the following process

$$\begin{aligned}
&start(); \mathsf{new}\ r_A : T_r; \mathsf{let}\ k_A : T_k = \mathrm{kgen}(r_A)\ \mathsf{in}\\
&\mathsf{new}\ r_B : T_r; \mathsf{let}\ k_B : T_k = \mathrm{kgen}(r_B)\ \mathsf{in}\ \overline{yield}\langle\rangle; (Q_K \mid Q_S)\\
&Q_K = !^{i \le n} c[i](h : T_h, k : T_k)\\
&\qquad \mathsf{if}\ h = A\ \mathsf{then\ let}\ k' : T_k = k_A\ \mathsf{in}\ \overline{yield}\langle\rangle\ \mathsf{else}\\
&\qquad \mathsf{if}\ h = B\ \mathsf{then\ let}\ k' : T_k = k_B\ \mathsf{in}\ \overline{yield}\langle\rangle\ \mathsf{else}\\
&\qquad \mathsf{let}\ k' : T_k = k\ \mathsf{in}\ \overline{yield}\langle\rangle\\
&Q_S = !^{i' \le n'} c'[i'](h' : T_h); \mathsf{find}\ u \le n\ \mathsf{suchthat}\\
&\qquad \mathsf{defined}(h[u], k'[u]) \wedge h' = h[u]\ \mathsf{then}\ P_1(k'[u])\ \mathsf{else}\ P_2
\end{aligned}$$

The process Q_K stores in (h, k') a table of pairs (host name, key): the key for A is k_A, for B, k_B, and for any other h, the adversary can choose the key k. The process Q_S queries this table of keys to find the key $k'[u]$ of host h', then executes $P_1(k'[u])$. If h' is not found, it executes P_2.

By the transformation **SArename**(k'), we can perform a case analysis, to distinguish the cases in which $k' = k_A$, $k' = k_B$, or $k' = k$. After transformation, we obtain the following processes:

$$\begin{aligned}
&Q'_K = !^{i \le n} c[i](h : T_h, k : T_k)\\
&\qquad \mathsf{if}\ h = A\ \mathsf{then\ let}\ k'_1 : T_k = k_A\ \mathsf{in}\ \overline{yield}\langle\rangle\ \mathsf{else}\\
&\qquad \mathsf{if}\ h = B\ \mathsf{then\ let}\ k'_2 : T_k = k_B\ \mathsf{in}\ \overline{yield}\langle\rangle\ \mathsf{else}\\
&\qquad \mathsf{let}\ k'_3 : T_k = k\ \mathsf{in}\ \overline{yield}\langle\rangle\\
&Q'_S = !^{i' \le n'} c'[i'](h' : T_h);\\
&\qquad \mathsf{find}\ u \le n\ \mathsf{suchthat\ defined}(h[u], k'_1[u])\\
&\qquad\qquad \wedge h' = h[u]\ \mathsf{then}\ P_1(k'_1[u])\\
&\qquad \oplus u \le n\ \mathsf{suchthat\ defined}(h[u], k'_2[u])\\
&\qquad\qquad \wedge h' = h[u]\ \mathsf{then}\ P_1(k'_2[u])\\
&\qquad \oplus u \le n\ \mathsf{suchthat\ defined}(h[u], k'_3[u])\\
&\qquad\qquad \wedge h' = h[u]\ \mathsf{then}\ P_1(k'_3[u])\ \mathsf{else}\ P_2
\end{aligned}$$

After the simplification (sketched below), Q'_S becomes:

$$\begin{aligned}
&Q''_S = !^{i' \le n'} c'[i'](h' : T_h);\\
&\qquad \mathsf{find}\ u \le n\ \mathsf{suchthat\ defined}(h[u], k'_1[u])\\
&\qquad\qquad \wedge h' = A\ \mathsf{then}\ P_1(k_A)\\
&\qquad \oplus u \le n\ \mathsf{suchthat\ defined}(h[u], k'_2[u])\\
&\qquad\qquad \wedge h' = B\ \mathsf{then}\ P_1(k_B)\\
&\qquad \oplus u \le n\ \mathsf{suchthat\ defined}(h[u], k'_3[u])\\
&\qquad\qquad \wedge h' = h[u]\ \mathsf{then}\ P_1(k[u])\ \mathsf{else}\ P_2
\end{aligned}$$

since, when $k'_1[u]$ is defined, $k'_1[u] = k_A$ and $h[u] = A$, and similarly for $k'_2[u]$ and $k'_3[u]$.

Simplify: The prover uses a simplification algorithm, based on an equational prover, using an algorithm similar to the Knuth-Bendix completion [29]. This equational prover uses:

- User-defined equations, of the form $\forall x_1 : T_1, \ldots, \forall x_m : T_m, M$ which mean that for all environments E, if for all $j \le m$, $E(x_j) \in I_\eta(T_j)$, then $E, M \Downarrow \mathrm{true}$. For example, considering MAC and encryption shemes as in Definitions 1 and 2 respectively, we have:

$$\begin{aligned}
&\forall r : T_{mr}, \forall m : bitstring,\\
&\quad \mathrm{check}(m, \mathrm{mkgen}(r), \mathrm{mac}(m, \mathrm{mkgen}(r))) = \mathrm{true} \qquad \text{(mac)}
\end{aligned}$$

$$\begin{aligned}
&\forall m : bitstring; \forall r : T_r, \forall r' : T'_r,\\
&\quad \mathrm{dec}(\mathrm{enc}(m, \mathrm{kgen}(r), r'), \mathrm{kgen}(r)) = \mathrm{i}_\perp(m) \qquad \text{(enc)}
\end{aligned}$$

  We express the poly-injectivity of the function k2b of Example 1 by

$$\begin{aligned}
&\forall x : T_k, \forall y : T_k, (\mathrm{k2b}(x) = \mathrm{k2b}(y)) = (x = y)\\
&\forall x : T_k, \mathrm{k2b}^{-1}(\mathrm{k2b}(x)) = x
\end{aligned} \qquad \text{(k2b)}$$

  where $\mathrm{k2b}^{-1}$ is a function symbol that denotes the inverse of k2b. We have similar formulas for $\mathrm{i}_\perp$.

- Equations that come from the process. For example, in the process if M then P else P', we have $M = \mathrm{true}$ in P and $M = \mathrm{false}$ in P'.

- The low probability of collision between random values. For example, when x is defined by new $x : T$ and T is a large type, $x[M_1, \ldots, M_m] = x[M'_1, \ldots, M'_m]$ implies $M_1 = M'_1, \ldots, M_m = M'_m$ up to negligible probability.

  Similarly, when 1) x is defined by new $x : T$ and T is a large type, 2) for each value of M_1, there is at most one value of x (or of a part of x of a large type) that can yield that value of M_1, and 3) M_2 does not depend on x, then $M_1 \neq M_2$ up to negligible probability. The fact that M_2 does not depend on x is proved using a dependency analysis.

The prover combines these properties to simplify terms, and uses simplified forms of terms to simplify processes. For example, if M simplifies to true, then if M then P else P' simplifies to P. Similarly, a branch of find is removed when the associated condition simplifies to false.

Details on the simplification procedure can be found in Appendix C and the proof of the following proposition in Appendix E.1.

Proposition 1 *Let Q_0 be a process that satisfies Invariants 1, 2, and 3 and Q'_0 the process obtained from Q_0 by one of the transformations above. Then Q'_0 satisfies Invariants 1, 2, and 3, and $Q_0 \approx^V Q'_0$.*

3.2 Applying the Definition of Security of Primitives

The security of cryptographic primitives is defined using observational equivalences given as axioms. Importantly, this formalism allows us to specify many different primitives in a generic way. Such equivalences are then used by the prover in order to transform a game into another, observationally equivalent game, as explained below in this section.

The primitives are specified using equivalences of the form $(G_1, \dots, G_m) \approx (G'_1, \dots, G'_m)$ where G is defined by the following grammar, with $l \geq 0$ and $m \geq 1$:

$G ::=$ group of functions
 $!^{i \leq n}\mathsf{new}\ y_1 : T_1; \dots; \mathsf{new}\ y_l : T_l; (G_1, \dots, G_m)$ replication, restrictions
 $(x_1 : T_1, \dots, x_l : T_l) \to FP$ function

$FP ::=$ functional processes
 M term
 $\mathsf{new}\ x[\widetilde{i}] : T; FP$ random number
 $\mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ FP$ assignment
 $\mathsf{find}\ (\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}\ \mathsf{suchthat}$
 $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ FP_j)\ \mathsf{else}\ FP$ array lookup

Intuitively, $(x_1 : T_1, \dots, x_l : T_l) \to FP$ represents a function that takes as argument values $x_1, \dots, x_l$ of types $T_1, \dots, T_l$ respectively and returns a result computed by FP. The observational equivalence $(G_1, \dots, G_m) \approx (G'_1, \dots, G'_m)$ expresses that the adversary has a negligible probability of distinguishing functions in the left-hand side from corresponding functions in the right-hand side. Formally, functions can be encoded as processes that input their arguments and output their result on a channel, as shown in Figure 2: $[\![FP]\!]^{\widetilde{j}}_{\widetilde{i}}$ denotes the translation of the functional process FP into an output process; $[\![G]\!]^{\widetilde{j}}_{\widetilde{i}}$ denotes the translation of the group of functions G into an input process. The translation of $!^{i \leq n}\mathsf{new}\ y_1 : T_1; \dots; \mathsf{new}\ y_l : T_l; (G_1, \dots, G_m)$ inputs and outputs on channel $c_{\widetilde{j}}$ so that the context can trigger the generation of random numbers $y_1, \dots, y_l$. The translation of $(x_1 : T_1, \dots, x_l : T_l) \to FP$ inputs the arguments of the function on channel $c_{\widetilde{j}}$ and translates FP, which outputs the result of FP on $c_{\widetilde{j}}$. (In the left-hand side of equivalences, the result FP of functions must simply be a term M.) The observational equivalence $(G_1, \dots, G_m) \approx (G'_1, \dots, G'_m)$ is then an abbreviation for $[\![(G_1, \dots, G_m)]\!] \approx [\![(G'_1, \dots, G'_m)]\!]$.

For example, the security of a MAC (Definition 1) is represented by the equivalence $L \approx R$ where:

$$
\begin{aligned}
L = {} & !^{i'' \leq n''}\mathsf{new}\ r : T_{mr}; (\\
& !^{i \leq n}(x : \mathit{bitstring}) \to \mathrm{mac}(x, \mathrm{mkgen}(r)), \\
& !^{i' \leq n'}(m : \mathit{bitstring}, ma : T_{ms}) \to \\
& \quad \mathrm{check}(m, \mathrm{mkgen}(r), ma)) \\
R = {} & !^{i'' \leq n''}\mathsf{new}\ r : T_{mr}; (\\
& !^{i \leq n}(x : \mathit{bitstring}) \to \mathrm{mac}'(x, \mathrm{mkgen}'(r)), \\
& !^{i' \leq n'}(m : \mathit{bitstring}, ma : T_{ms}) \to \\
& \quad \mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[u]) \wedge (m = x[u]) \\
& \quad \wedge \mathrm{check}'(m, \mathrm{mkgen}'(r), ma)\ \mathsf{then}\ \mathrm{true}\ \mathsf{else}\ \mathrm{false})
\end{aligned}
\qquad (\mathrm{mac}_{\mathrm{eq}})
$$

where mac', check', and mkgen' are function symbols with the same types as mac, check, and mkgen respectively. (We use different function symbols on the left- and right-hand sides, just to prevent a repeated application of the transformation induced by this equivalence. Since we add these function symbols, we also add the equation

$$
\begin{aligned}
& \forall r : T_{mr}, \forall m : \mathit{bitstring}, \\
& \quad \mathrm{check}'(m, \mathrm{mkgen}'(r), \mathrm{mac}'(m, \mathrm{mkgen}'(r))) = \mathrm{true}
\end{aligned}
\qquad (\mathrm{mac}')
$$

which restates (mac) for mac', check', and mkgen'.) Intuitively, the equivalence $L \approx R$ leaves MAC computations unchanged (except for the use of primed function symbols in R), and allows one to replace a MAC checking $\mathrm{check}(m, \mathrm{mkgen}(r), ma)$ with a lookup in the array x of messages whose mac has been computed with key $\mathrm{mkgen}(r)$: if m is found in the array x and $\mathrm{check}(m, \mathrm{mkgen}(r), ma)$, we return true; otherwise, the check fails (up to negligible probability), so we return false. (If the check succeeded with m not in the array x, the adversary would have forged a MAC.) Obviously, the form of L requires that r is used only to compute or check MACs, for the equivalence to be correct. Formally, the following result shows the correctness of our modeling. It is a fairly easy consequence of Definition 1, and is proved in Appendix E.3.

Proposition 2 *If* $(\mathrm{mkgen}, \mathrm{mac}, \mathrm{check})$ *is a UF-CMA message authentication code,* $I_\eta(\mathrm{mkgen}') = I_\eta(\mathrm{mkgen})$*,* $I_\eta(\mathrm{mac}') = I_\eta(\mathrm{mac})$*, and* $I_\eta(\mathrm{check}') = I_\eta(\mathrm{check})$*, then* $[\![L]\!] \approx [\![R]\!]$.

Similarly, if $(\mathrm{kgen}, \mathrm{enc}, \mathrm{dec})$ is an IND-CPA symmetric encryption scheme (Definition 2), then we have the following equivalence:

$$
\begin{aligned}
& !^{i' \leq n'}\mathsf{new}\ r : T_r; !^{i \leq n}(x : \mathit{bitstring}) \to \\
& \qquad \mathsf{new}\ r' : T'_r; \mathrm{enc}(x, \mathrm{kgen}(r), r') \\
\approx {} & !^{i' \leq n'}\mathsf{new}\ r : T_r; !^{i \leq n}(x : \mathit{bitstring}) \to \\
& \qquad \mathsf{new}\ r' : T'_r; \mathrm{enc}'(\mathrm{Z}(x), \mathrm{kgen}'(r), r')
\end{aligned}
\qquad (\mathrm{enc}_{\mathrm{eq}})
$$

where enc' and kgen' are function symbols with the same types as enc and kgen respectively, and $\mathrm{Z} : \mathit{bitstring} \to \mathit{bitstring}$ is the function that returns a bitstring of the same length as its argument, consisting only of zeroes. Using equations such as $\forall x : T, \mathrm{Z}(\mathrm{T2b}(x)) = \mathrm{Z}_T$, we can prove that $\mathrm{Z}(\mathrm{T2b}(x))$ does not depend on x when x is of a fixed-length type and $\mathrm{T2b} : T \to \mathit{bitstring}$ is the natural injection. The representation of other primitives can be found in Appendix D.3. The equivalences that formalize the security assumptions on primitives are designed and proved correct by hand from security assumptions in a more standard form, as in the MAC example. Importantly, these manual proofs are done only once for each primitive, and the obtained equivalence can be reused for proving many different protocols automatically.

We use such equivalences $L \approx R$ in order to transform a process Q_0 observationally equivalent to $C[[\![L]\!]]$ into a process Q'_0 observationally equivalent to $C[[\![R]\!]]$, for some evaluation context C. In order to check that $Q_0 \approx^V C[[\![L]\!]]$, the prover uses sufficient conditions, which essentially guarantee that all uses of certain secret variables of Q_0, in a set S, can be implemented by calling functions of L. Let $\mathcal{M}$ be a set of occurrences of terms,

$$\llbracket (G_1, \ldots, G_m) \rrbracket = \llbracket G_1 \rrbracket^1 \mid \ldots \mid \llbracket G_m \rrbracket^m$$

$$\llbracket !^{i \le n} \mathsf{new}\ y_1 : T_1; \ldots; \mathsf{new}\ y_l : T_l; (G_1, \ldots, G_m) \rrbracket_{\widetilde{i}}^{\widetilde{j}} =$$
$$!^{i \le n} c_{\widetilde{j}}[\widetilde{i}, i](); \mathsf{new}\ y_1 : T_1; \ldots; \mathsf{new}\ y_l : T_l; \overline{c_{\widetilde{j}}[\widetilde{i}, i]}\langle\rangle; (\llbracket G_1 \rrbracket_{\widetilde{i},i}^{\widetilde{j},1} \mid \ldots \mid \llbracket G_m \rrbracket_{\widetilde{i},i}^{\widetilde{j},m})$$

$$\llbracket (x_1 : T_1, \ldots, x_l : T_l) \to FP \rrbracket_{\widetilde{i}}^{\widetilde{j}} = c_{\widetilde{j}}[\widetilde{i}](x_1 : T_1, \ldots, x_l : T_l); \llbracket FP \rrbracket_{\widetilde{i}}^{\widetilde{j}}$$

$$\llbracket M \rrbracket_{\widetilde{i}}^{\widetilde{j}} = \overline{c_{\widetilde{j}}[\widetilde{i}]}\langle M \rangle$$

$$\llbracket \mathsf{new}\ x[\widetilde{i}] : T; FP \rrbracket_{\widetilde{i}}^{\widetilde{j}} = \mathsf{new}\ x[\widetilde{i}] : T; \llbracket FP \rrbracket_{\widetilde{i}}^{\widetilde{j}}$$

$$\llbracket \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ FP \rrbracket_{\widetilde{i}}^{\widetilde{j}} = \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ \llbracket FP \rrbracket_{\widetilde{i}}^{\widetilde{j}}$$

$$\llbracket \mathsf{find}\ (\textstyle\bigoplus_{j=1}^m \widetilde{u_j}[\widetilde{i}] \le \widetilde{n_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ FP_j)\ \mathsf{else}\ FP \rrbracket_{\widetilde{i}}^{\widetilde{j}} =$$
$$\mathsf{find}\ (\textstyle\bigoplus_{j=1}^m \widetilde{u_j}[\widetilde{i}] \le \widetilde{n_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ \llbracket FP_j \rrbracket_{\widetilde{i}}^{\widetilde{j}})\ \mathsf{else}\ \llbracket FP \rrbracket_{\widetilde{i}}^{\widetilde{j}}$$

where $c_{\widetilde{j}}$ are pairwise distinct channels, $\widetilde{i} = i_1, \ldots, i_{l'}$, and $\widetilde{j} = j_0, \ldots, j_{l'}$.

Figure 2: Translation from functional processes to processes

corresponding to uses of variables of S. Informally, the prover shows the following properties.

- For each $M \in \mathcal{M}$, there exist a term N_M, which is the result of a function of L, and a substitution σ_M such that $M = \sigma_M N_M$. (Precisely, σ_M applies to the abbreviated form of N_M in which we write x instead of $x[\widetilde{i}]$.) Intuitively, the evaluation of M in Q_0 will correspond to a call to the function with result N_M in $C[\llbracket L \rrbracket]$.
- The variables of S do not occur in V, are bound by restrictions in Q_0, and occur only in terms $M = \sigma_M N_M \in \mathcal{M}$ in Q_0, at occurrences that are images by σ_M of variables bound by restrictions in L. (To be precise, the variables of S are also allowed to occur at the root of defined conditions; in that case, their value does not matter, just the fact that they are defined.)
- Let $\widetilde{i}$ and $\widetilde{i'}$ be the sequences of current replication indices at N_M in L and at M in Q_0, respectively. The prover shows that there exists a function mapIdx_M that maps the array indices at M in Q_0 to the array indices at N_M in L: the evaluation of M when $\widetilde{i'} = \widetilde{a}$ will correspond in $C[\llbracket L \rrbracket]$ to the evaluation of N_M when $\widetilde{i} = \mathrm{mapIdx}_M(\widetilde{a})$. Thus, σ_M and mapIdx_M induce a correspondence between terms and variables of Q_0 and variables of L: for all $M \in \mathcal{M}$, for all $x[\widetilde{i''}]$ that occur in N_M, $(\sigma_M x)\{\widetilde{a}/\widetilde{i'}\}$ corresponds to $x[\widetilde{i''}]\{\mathrm{mapIdx}_M(\widetilde{a})/\widetilde{i}\}$, that is, $(\sigma_M x)\{\widetilde{a}/\widetilde{i'}\}$ in a trace of Q_0 has the same value as $x[\widetilde{i''}]\{\mathrm{mapIdx}_M(\widetilde{a})/\widetilde{i}\}$ in the corresponding trace of $C[\llbracket L \rrbracket]$ ($\widetilde{i''}$ is a prefix of $\widetilde{i}$). We detail below conditions that this correspondence has to satisfy.

For example, consider a process Q_0 that contains $M_1 = \mathrm{enc}(M'_1, \mathrm{kgen}(x_r), x'_r[i_1])$ under a replication $!^{i_1 \le n_1}$ and $M_2 = \mathrm{enc}(M'_2, \mathrm{kgen}(x_r), x''_r[i_2])$ under a replication $!^{i_2 \le n_2}$, where x_r, x'_r, x''_r are bound by restrictions. Let $S = \{x_r, x'_r, x''_r\}$, $\mathcal{M} = \{M_1, M_2\}$, and $N_{M_1} = N_{M_2} = \mathrm{enc}(x[i', i], \mathrm{kgen}(r[i']), r'[i', i])$. The functions mapIdx_{M_1} and mapIdx_{M_2} are defined by

$$\mathrm{mapIdx}_{M_1}(a_1) = (1, a_1) \text{ for } a_1 \in [1, I_\eta(n_1)]$$
$$\mathrm{mapIdx}_{M_2}(a_2) = (1, a_2 + I_\eta(n_1)) \text{ for } a_2 \in [1, I_\eta(n_2)]$$

Then $M'_1\{a_1/i_1\}$ corresponds to $x[1, a_1]$, x_r to $r[1]$, $x'_r[a_1]$ to $r'[1, a_1]$, $M'_2\{a_2/i_2\}$ to $x[1, a_2 + I_\eta(n_1)]$, and $x''_r[a_2]$ to $r'[1, a_2 + I_\eta(n_1)]$. The functions mapIdx_{M_1} and mapIdx_{M_2} are such that $x_{r'}[a_1]$ and $x_{r''}[a_2]$ never correspond to the same cell of r'; indeed, $x_{r'}[a_1]$ and $x_{r''}[a_2]$ are independent random numbers in Q_0, so their images in $C[\llbracket L \rrbracket]$ must also be independent random numbers.

The above correspondence must satisfy the following soundness conditions:

- when x is a function argument in L, the term that corresponds to $x[\widetilde{a'}]$ must have the same type as $x[\widetilde{a'}]$, and when two terms correspond to the same $x[\widetilde{a'}]$, they must evaluate to the same value;
- when x is bound by $\mathsf{new}\ x : T$ in L, the term that corresponds to $x[\widetilde{a'}]$ must evaluate to $z[\widetilde{a''}]$ where $z \in S$ and z is bound by $\mathsf{new}\ z : T$ in Q_0, and the relation that associates $z[\widetilde{a''}]$ to $x[\widetilde{a'}]$ is an injective function (so that independent random numbers in L correspond to independent random numbers in Q_0).

It is easy to check that, in the previous example, these conditions are satisfied.

The transformation of Q_0 into Q'_0 consists in two steps. First, we replace the restrictions that define variables of S with restrictions that define fresh variables corresponding to variables bound by new in R. The correspondence between variables of Q_0 and variables of $C[\llbracket L \rrbracket]$ is extended to include these fresh variables. Second, we reorganize Q_0 so that each evaluation of a term $M \in \mathcal{M}$ first stores the values of the arguments $x_1, \ldots, x_m$ of the function $(x_1 : T_1, \ldots, x_m : T_m) \to N_M$ in fresh variables, then computes N_M and stores its result in a fresh variable, and uses this variable instead of M; then we simply replace the computation of N_M with the corresponding functional

process of R, taking into account the correspondence of variables.

The full formal description of this transformation is given Appendix D.1. The following proposition shows the soundness of the transformation and is proved in Appendix E.4.

Proposition 3 *Let Q_0 be a process that satisfies Invariants 1, 2, and 3 and Q'_0 the process obtained from Q_0 by the above transformation. Then Q'_0 satisfies Invariants 1, 2, and 3 and, if $[\![L]\!] \approx [\![R]\!]$ for all polynomials $\mathrm{maxlen}_\eta(c_{j_0,\dots,j_l})$ and $I_\eta(n)$ where n is any replication bound of L or R, then $Q_0 \approx^V Q'_0$.*

Example 4 In order to treat Example 1, the prover is given as input the indication that T_{mr}, T_r, T'_r, and T_k are fixed-length types; the type declarations for the functions $\mathrm{mkgen}, \mathrm{mkgen}' : T_{mr} \rightarrow T_{mk}$, $\mathrm{mac}, \mathrm{mac}' : \mathit{bitstring} \times T_{mk} \rightarrow T_{ms}$, $\mathrm{check}, \mathrm{check}' : \mathit{bitstring} \times T_{mk} \times T_{ms} \rightarrow \mathit{bool}$, $\mathrm{kgen}, \mathrm{kgen}' : T_r \rightarrow T_k$, $\mathrm{enc}, \mathrm{enc}' : \mathit{bitstring} \times T_k \times T'_r \rightarrow T_e$, $\mathrm{dec} : T_e \times T_k \rightarrow \mathit{bitstring}_\bot$, $\mathrm{k2b} : T_k \rightarrow \mathit{bitstring}$, $\mathrm{i}_\bot : \mathit{bitstring} \rightarrow \mathit{bitstring}_\bot$, $\mathrm{Z} : \mathit{bitstring} \rightarrow \mathit{bitstring}$, and the constant $\mathrm{Z}_k : \mathit{bitstring}$; the equations (mac), (mac′), (enc), and $\forall x : T_k, \mathrm{Z}(\mathrm{k2b}(x)) = \mathrm{Z}_k$ (which expresses that all keys have the same length); the indication that k2b and $\mathrm{i}_\bot$ are poly-injective (which generates the equations (k2b) and similar equations for $\mathrm{i}_\bot$); equivalences $L \approx R$ for MAC ($\mathrm{mac}_{\mathrm{eq}}$) and encryption ($\mathrm{enc}_{\mathrm{eq}}$); and the process Q_0 of Example 1.

The prover first applies **RemoveAssign**(x_{mk}) to the process Q_0 of Example 1, as described in Example 2. The process can then be transformed using the security of the MAC. Let $S = \{x'_r\}$, $M_1 = \mathrm{mac}(x_m[i], \mathrm{mkgen}(x'_r))$, $M_2 = \mathrm{check}(x'_m[i'], \mathrm{mkgen}(x'_r), x_{ma}[i'])$, and $\mathcal{M} = \{M_1, M_2\}$. We have $N_{M_1} = \mathrm{mac}(x[i'', i], \mathrm{mkgen}(r[i'']))$, $N_{M_2} = \mathrm{check}(m[i'', i'], \mathrm{mkgen}(r[i'']), ma[i'', i'])$, $\mathrm{mapIdx}_{M_1}(a_1) = (1, a_1)$, and $\mathrm{mapIdx}_{M_2}(a_2) = (1, a_2)$, so $x_m[a_1]$ corresponds to $x[1, a_1]$, x'_r to $r[1]$, $x'_m[a_2]$ to $m[1, a_2]$, and $x_{ma}[a_2]$ to $ma[1, a_2]$.

After transformation, we get the following process Q'_0:

$$
\begin{aligned}
&Q'_0 = \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{let}\ x_k : T_k = \mathrm{kgen}(x_r)\ \mathsf{in}\\
&\quad \mathsf{new}\ x'_r : T_{mr}; \overline{c}\langle\rangle; (Q'_A \mid Q'_B)\\
&Q'_A = !^{i \le n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x''_r : T'_r;\\
&\quad \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{k2b}(x'_k), x_k, x''_r)\ \mathsf{in}\\
&\quad \overline{c_A[i]}\langle x_m, \mathrm{mac}'(x_m, \mathrm{mkgen}'(x'_r))\rangle\\
&Q'_B = !^{i' \le n} c_B[i'](x'_m, x_{ma});\\
&\quad \mathsf{find}\ u \le n\ \mathsf{suchthat}\ \mathsf{defined}(x_m[u]) \wedge x'_m = x_m[u] \wedge\\
&\qquad\quad \mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})\ \mathsf{then}\\
&\qquad (\mathsf{if}\ \mathrm{true}\ \mathsf{then}\ \mathsf{let}\ \mathrm{i}_\bot(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\\
&\qquad\qquad \overline{c_B[i']}\langle\rangle)\\
&\quad \mathsf{else}\\
&\qquad (\mathsf{if}\ \mathrm{false}\ \mathsf{then}\ \mathsf{let}\ \mathrm{i}_\bot(\mathrm{k2b}(x''_k)) = \mathrm{dec}(x'_m, x_k)\ \mathsf{in}\\
&\qquad\qquad \overline{c_B[i']}\langle\rangle)
\end{aligned}
$$

The initial definition of x'_r is removed and replaced with a new definition, which we still call x'_r. The term $\mathrm{mac}(x_m, \mathrm{mkgen}(x'_r))$ is replaced with $\mathrm{mac}'(x_m, \mathrm{mkgen}'(x'_r))$. The term $\mathrm{check}(x'_m, \mathrm{mkgen}(x'_r), x_{ma})$ becomes $\mathsf{find}\ u \le n\ \mathsf{suchthat}\ \mathsf{defined}(x_m[u]) \wedge x'_m = x_m[u] \wedge \mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})$ $\mathsf{then}\ \mathrm{true}\ \mathsf{else}\ \mathrm{false}$, which yields Q'_B after transformation of functional processes into processes. The process looks up the message x'_m in the array x_m, which contains the messages whose MAC has been computed with key $\mathrm{mkgen}(x'_r)$. If the MAC of x'_m has never been computed, the check always fails (it returns false) by the definition of security of the MAC. Otherwise, it returns true when $\mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})$.

After applying **Simplify**, Q'_A is unchanged and Q'_B becomes

$$
\begin{aligned}
&Q''_B = !^{i' \le n} c_B[i'](x'_m, x_{ma});\\
&\quad \mathsf{find}\ u \le n\ \mathsf{suchthat}\ \mathsf{defined}(x_m[u], x'_k[u]) \wedge\\
&\qquad x'_m = x_m[u] \wedge \mathrm{check}'(x'_m, \mathrm{mkgen}'(x'_r), x_{ma})\ \mathsf{then}\\
&\quad \mathsf{let}\ x''_k : T_k = x'_k[u]\ \mathsf{in}\ \overline{c_B[i']}\langle\rangle
\end{aligned}
$$

First, the tests $\mathsf{if}\ \mathrm{true}\ \mathsf{then}\ \dots$ and $\mathsf{if}\ \mathrm{false}\ \mathsf{then}\ \dots$ are simplified. The term $\mathrm{dec}(x'_m, x_k)$ is simplified knowing $x'_m = x_m[u]$ by the find condition, $x_m[u] = \mathrm{enc}(\mathrm{k2b}(x'_k[u]), x_k, x''_r[u])$ by the assignment that defines x_m, $x_k = \mathrm{kgen}(x_r)$ by the assignment that defines x_k, and $\mathrm{dec}(\mathrm{enc}(m, \mathrm{kgen}(r), r'), \mathrm{kgen}(r)) = \mathrm{i}_\bot(m)$ by (enc). So we have $\mathrm{dec}(x'_m, x_k) = \mathrm{i}_\bot(\mathrm{k2b}(x'_k[u]))$. By injectivity of $\mathrm{i}_\bot$ and k2b, the assignment to x''_k simply becomes $x''_k = x'_k[u]$, using the equations $\forall x : \mathit{bitstring}, \mathrm{i}_\bot^{-1}(\mathrm{i}_\bot(x)) = x$ and $\forall x : T_k, \mathrm{k2b}^{-1}(\mathrm{k2b}(x)) = x$.

After applying **RemoveAssign**(x_k), we apply the security of encryption: $\mathrm{enc}(\mathrm{k2b}(x'_k), \mathrm{kgen}(x_r), x''_r)$ becomes $\mathrm{enc}'(\mathrm{Z}(\mathrm{k2b}(x'_k)), \mathrm{kgen}(x_r), x''_r)$. After **Simplify**, it becomes $\mathrm{enc}'(\mathrm{Z}_k, \mathrm{kgen}(x_r), x''_r)$, using $\forall x : T_k, \mathrm{Z}(\mathrm{k2b}(x)) = \mathrm{Z}_k$ (which expresses that all keys have the same length).

So we obtain the following game:

$$
\begin{aligned}
&Q''_0 = \mathit{start}(); \mathsf{new}\ x_r : T_r; \mathsf{new}\ x'_r : T_{mr}; \overline{c}\langle\rangle; (Q''_A \mid Q''_B)\\
&Q''_A = !^{i \le n} c_A[i](); \mathsf{new}\ x'_k : T_k; \mathsf{new}\ x''_r : T'_r;\\
&\quad \mathsf{let}\ x_m : \mathit{bitstring} = \mathrm{enc}(\mathrm{Z}_k, \mathrm{kgen}(x_r), x''_r)\ \mathsf{in}\\
&\quad \overline{c_A[i]}\langle x_m, \mathrm{mac}'(x_m, \mathrm{mkgen}'(x'_r))\rangle
\end{aligned}
$$

where Q''_B remains as above.

Using arrays instead of lists simplifies this transformation: we do not need to add instructions that insert values in the list, since all variables are always implicitly arrays. Moreover, if there are several occurrences of $\mathrm{mac}(x_i, k)$ with the same key in the initial process, each $\mathrm{check}(m_j, k, ma_j)$ is replaced with a find with one branch for each occurrence of mac. Therefore, the prover distinguishes automatically the cases in which the checked MAC ma_j comes from each occurrence of mac, that is, it distinguishes cases depending on the value of i such that $m_j = x_i$. Typically, distinguishing these cases is useful in the following steps of the proof of the protocol. (A similar situation arises for other cryptographic primitives specified using find.)

4 Criteria for Proving Secrecy Properties

Let us now define syntactic criteria that allow us to prove secrecy properties of protocols. The proofs for these results can be found in Appendix E.5.

Definition 4 (One-session secrecy) The process Q *preserves the one-session secrecy of* x when $Q \mid Q_x \approx Q \mid Q'_x$, where

$$Q_x = c(u_1 : [1, n_1], \dots, u_m : [1, n_m]); \quad \text{if defined}(x[u_1, \dots, u_m]) \text{ then } \overline{c}\langle x[u_1, \dots, u_m]\rangle$$
$$Q'_x = c(u_1 : [1, n_1], \dots, u_m : [1, n_m]); \quad \text{if defined}(x[u_1, \dots, u_m]) \text{ then new } y : T; \overline{c}\langle y\rangle$$

$c \notin \mathrm{fc}(Q)$, $u_1, \dots, u_m, y \notin \mathrm{var}(Q)$, and $\mathcal{E}(x) = [1, n_1] \times \dots \times [1, n_m] \to T$.

Intuitively, the adversary cannot distinguish a process that outputs the value of the secret from one that outputs a random number. The adversary performs a single test query, modeled by Q_x and Q'_x.

Proposition 4 (One-session secrecy) *Consider a process Q such that there exists a set of variables S such that 1) the definitions of x are either restrictions* new $x[\widetilde{i}] : T$ *and $x \in S$, or assignments* let $x[\widetilde{i}] : T = z[M_1, \dots, M_l]$ *where z is defined by restrictions* new $z[i'_1, \dots, i'_l] : T$*, and $z \in S$, and 2) all accesses to variables $y \in S$ in Q are of the form* "let $y'[\widetilde{i}] : T' = y[M_1, \dots, M_l]$" *with $y' \in S$. Then $Q \mid Q_x \approx_0 Q \mid Q'_x$, hence Q preserves the one-session secrecy of x.*

Intuitively, only the variables in S depend on the restriction that defines x; the sent messages and the control flow of the process are independent of x, so the adversary obtains no information on x. In the implementation, the set S is computed by fixpoint iteration, starting from x or z and adding variables y' defined by "let $y'[\widetilde{i}] : T' = y[M_1, \dots, M_l]$" when $y \in S$.

Definition 5 (Secrecy) The process Q *preserves the secrecy of* x when $Q \mid R_x \approx Q \mid R'_x$, where

$$R_x = !^{i \le n} c(u_1 : [1, n_1], \dots, u_m : [1, n_m]); \quad \text{if defined}(x[u_1, \dots, u_m]) \text{ then } \overline{c}\langle x[u_1, \dots, u_m]\rangle$$
$$\begin{aligned} R'_x = {} & !^{i \le n} c(u_1 : [1, n_1], \dots, u_m : [1, n_m]); \\ & \text{if defined}(x[u_1, \dots, u_m]) \text{ then} \\ & \text{find } u' \le n \text{ suchthat defined}(y[u'], u_1[u'], \dots, u_m[u']) \\ & \quad \wedge u_1[u'] = u_1 \wedge \dots \wedge u_m[u'] = u_m \\ & \text{then } \overline{c}\langle y[u']\rangle \text{ else new } y : T; \overline{c}\langle y\rangle \end{aligned}$$

$c \notin \mathrm{fc}(Q)$, $u_1, \dots, u_m, u', y \notin \mathrm{var}(Q)$, $\mathcal{E}(x) = [1, n_1] \times \dots \times [1, n_m] \to T$, and $I_\eta(n) \ge I_\eta(n_1) \times \dots \times I_\eta(n_m)$.

Intuitively, the adversary cannot distinguish a process that outputs the value of the secret for several indices from one that outputs independent random numbers. In this definition, the adversary can perform several test queries, modeled by R_x and R'_x. This corresponds to the "real-or-random" definition of security [4]. (As shown in [4], this notion is stronger than the more standard approach in which the adversary can perform a single test query and some reveal queries, which always reveal $x[u_1, \dots, u_m]$.)

Proposition 5 (Secrecy) *Assume that Q satisfies the hypothesis of Proposition 4.*

When $\mathcal{T}$ is a trace of $C[Q]$ for some evaluation context C, we define $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}])$*, the defining restriction of $x[\widetilde{a}]$ in trace $\mathcal{T}$, as follows: if $x[\widetilde{a}]$ is defined by* new $x[\widetilde{a}] : T$ *in $\mathcal{T}$, $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = x[\widetilde{a}]$; if $x[\widetilde{a}]$ is defined by* let $x[\widetilde{a}] : T = z[M_1, \dots, M_l]$, *$\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = z[a'_1, \dots, a'_l]$ where $E, M_k \Downarrow a'_k$ for all $k \le l$ and E is the environment in $\mathcal{T}$ at the definition of $x[\widetilde{a}]$.*

Assume that for all evaluation contexts C acceptable for Q, 0, $\{x\}$, the probability $\Pr[\exists(\mathcal{T}, \widetilde{a}, \widetilde{a'}), C[Q]$ reduces according to $\mathcal{T} \wedge \widetilde{a} \ne \widetilde{a'} \wedge \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])]$ is negligible. Then Q preserves the secrecy of x.

The last hypothesis can be verified using the same equational prover as for **Simplify** in Section 3.1, as detailed in Appendix E.2. Intuitively, this hypothesis guarantees that when $\widetilde{a} \ne \widetilde{a'}$, we have $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) \ne \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])$ except in cases of negligible probability, so $x[\widetilde{a}]$ and $x[\widetilde{a'}]$ are defined by different restrictions, so they are independent random numbers.

As we show in [18], this notion of secrecy composed with correspondence assertions [48] can be used to prove security of a key exchange. (Correspondence assertions are properties of the form "if some event $e(\widetilde{M})$ has been executed then some events $e_i(\widetilde{M_i})$ for $i \le m$ have been executed". We have recently implemented the verification of correspondence assertions in CryptoVerif [18].)

Lemma 2 *If $Q \approx^{\{x\}} Q'$ and Q preserves the one-session secrecy of x then Q' preserves the one-session secrecy of x. The same result holds for secrecy.*

We can then apply the following technique. When we want to prove that Q_0 preserves the (one-session) secrecy of x, we transform Q_0 by the transformations described in Section 3 with $V = \{x\}$. By Propositions 1 and 3, we obtain a process Q'_0 such that $Q_0 \approx^V Q'_0$. We use Propositions 4 or 5 to show that Q'_0 preserves the (one-session) secrecy of x and finally conclude that Q_0 also preserves the (one-session) secrecy of x by Lemma 2.

Example 5 After the transformations of Example 4, the only variable access to x'_k in the considered process is let $x''_k : T_k = x'_k[u]$ and x''_k is not used in the considered process. So by Proposition 4, the considered process preserves the one-session secrecy of x''_k (with $S = \{x'_k, x''_k\}$). By Lemma 2, the process of Example 1 also preserves the one-session secrecy of x''_k. However, this process does not preserve the secrecy of x''_k, because the adversary can force several sessions of B to use the same key x''_k, by replaying the message sent by A. (Accordingly, the hypothesis of Proposition 5 is not satisfied.)

The criteria given in this section might seem restrictive, but in fact, they should be sufficient for all protocols, provided the previous transformation steps are powerful enough to transform the protocol into a simpler protocol, on which these criteria can then be applied.

5 Proof Strategy

Up to now, we have described the available game transformations. Next, we explain how we organize these transformations in order to prove protocols.

At the beginning of the proof and after each successful cryptographic transformation (that is, a transformation of Section 3.2), the prover executes **Simplify** and tests whether the desired security properties are proved, as described in Section 4. If so, it stops.

In order to perform the cryptographic transformations and the other syntactic transformations, our proof strategy relies of the idea of advice. Precisely, the prover tries to execute each available cryptographic transformation in turn. When such a cryptographic transformation fails, it returns some syntactic transformations that could make the desired transformation work. (These are the advised transformations.) Then the prover tries to perform these syntactic transformations. If they fail, they may also suggest other advised transformations, which are then executed. When the syntactic transformations finally succeed, we retry the desired cryptographic transformation, which may succeed or fail, perhaps with new advised transformations, and so on.

The prover determines the advised transformations as follows:

- Assume that we try to execute a cryptographic transformation, and need to recognize a certain term M of L, but we find in Q_0 only part of M, the other parts being variable accesses $x[\ldots]$ while we expect function applications. In this case, we advise **RemoveAssign**(x). For example, if Q_0 contains $\mathrm{enc}(M', x_k, x'_r)$ and we look for $\mathrm{enc}(x_m, \mathrm{kgen}(x_r), x_{r'})$, we advise **RemoveAssign**(x_k). If Q_0 contains $\mathsf{let}\ x_k = \mathrm{mkgen}(x_r)$ and we look for $\mathrm{mac}(x_m, \mathrm{mkgen}(x_r))$, we also advise **RemoveAssign**(x_k). (The transformation of Example 2 is advised for this reason.)
- When we try to execute **RemoveAssign**(x), x has several definitions, and there are accesses to variable x guarded by find in Q_0, we advise **SArename**(x).
- When we check whether x is secret or one-session secret, we have an assignment $\mathsf{let}\ x[\widetilde{i}] : T = y[\widetilde{M}]$ in P, and there is at least one assignment defining y, we advise **RemoveAssign**(y).

  When we check whether x is secret or one-session secret, we have an assignment $\mathsf{let}\ x[\widetilde{i}] : T = y[\widetilde{M}]$ in P, y is defined by restrictions, y has several definitions, and some variable accesses to y are not of the form $\mathsf{let}\ y'[\widetilde{i'}] : T = y[\widetilde{M'}]$ in P', we advise **SArename**(y).

These pieces of advice are the only ones we use, but one may obviously extend them if needed.

6 Experimental Results

We have successfully tested our prover on a number of protocols given in the literature. All these protocols have been tested in a configuration in which the honest participants are willing to run sessions with the adversary, and we prove secrecy of keys for sessions between honest participants. In these examples, shared-key encryption is encoded using a symmetric encryption scheme and a MAC as in Example 1, public-key encryption is assumed to be IND-CCA2 (indistinguishability under adaptive chosen-ciphertext attacks) [14], public-key signature is assumed to be UF-CMA (unforgeability under chosen message attacks).

For each proof, the prover outputs the sequence of games it has built, a succinct explanation of the transformation performed between consecutive games, and an indication whether the proof succeeded or failed. When the proof fails, the prover still outputs a sequence of games, but the last game of this sequence does not show the desired property and cannot be transformed further by the prover. Manual inspection of this game often makes it possible to understand why the proof failed: because there is an attack (if there is an attack on the last game), because of a limitation of the prover (if it should in fact be able to prove the property or to transform the game further), for other reasons (such as the protocol cannot be proved from the given assumptions; this situation may not lead immediately to a practical attack in the computational model).

Otway-Rees [42] We automatically prove the secrecy of the exchanged key.

Yahalom [20] For the original version of the protocol, our prover cannot show the one-session secrecy of the exchanged key, because the protocol is not secure, at least using encrypt-then-MAC as definition of encryption. Indeed, there is a confirmation round $\{N_B\}_K$ where K is the exchanged key. This message may reveal some information on K. After removing this confirmation round, our prover shows the one-session secrecy of K. However, it cannot show the secrecy of K, since in the absence of a confirmation round, the adversary may force several sessions of Yahalom to use the same key.

Needham-Schroeder shared-key [40] As in the Yahalom protocol, a key confirmation round may reveal some information on the key. After removing this round, our prover shows the one-session secrecy of the exchanged key. It does not prove the secrecy of the exchanged key, because the adversary may force several sessions of the protocol to use the same key. Our prover shows the secrecy for the corrected version [41].

Denning-Sacco public-key [25] Our prover cannot show the one-session secrecy of the exchanged key, since there is an attack against this protocol [2]. The one-session secrecy of the exchanged key is proved for the corrected version [2]. Secrecy is not proved since the adversary can force several sessions of the protocol to use the same key. (We do not model timestamps in this protocol.) In contrast to the previous examples, we give the main proof steps to the prover manually, as follows:

```
SArename Rkey
crypto enc rkB
crypto sign rkS
crypto sign rkA
success
```

The variable Rkey defines a table of public keys and is assigned at three places, corresponding to principals A and B, and to other principals defined by the adversary (like the variable k' in Example 3). The instruction SArename Rkey allows us to distinguish these three cases. The instruction crypto enc rkB means that the prover should apply the definition of security of encryption (primitive enc), for the key generated from random number rkB. The instruction success means that prover should check whether the desired security properties are proved.

Needham-Schroeder public-key [40] This protocol is an authentication protocol. Since our prover cannot check authentication yet, we transform it into a key exchange protocol in several ways, by choosing for the key either one of the nonces N_A and N_B shared between A and B, or $H(N_A, N_B)$ where H is a hash function (in the random oracle model). When the key is $H(N_A, N_B)$, the one-session secrecy of the key cannot be proved for the original protocol, due to the well-known attack [35]. For the corrected version [35], our prover shows secrecy of the key $H(N_A, N_B)$. For both the original and the corrected versions, the prover cannot prove the one-session secrecy of N_A or N_B. For N_B, the failure of the proof corresponds to an attack: the adversary can check whether it is given N_B or a random number by sending $\{N'_B\}_{pk_B}$ to B as the last message of the protocol: B accepts if and only if $N'_B = N_B$. For N_A, the failure of the proof comes from limitations of our prover: the prover cannot take into account that N_A is accepted only after all messages that contain N_A have been sent, which prevents the previous attack. (This is the only case in our examples where the failure of the proof comes from limitations of the prover. This problem could probably be solved by improving the transformation **Simplify**.) Like for the Denning-Sacco protocol, we provided the main proof steps to the prover manually, as follows when the distributed key is N_A or N_B:

```
SArename Rkey
crypto sign rkS
crypto enc rkA
crypto enc rkB
success
```

When the distributed key is $H(N_A, N_B)$, the proof is as follows:

```
SArename Rkey
crypto sign rkS
crypto enc rkA
crypto enc rkB
crypto hash
SArename Na_39
simplify
success
```

The total runtime for all these tests is 77 s on a Pentium M 1.8 GHz, for version 1.03 of our prover CryptoVerif. These examples are included in the CryptoVerif distribution available at http://www.di.ens.fr/~blanchet/cryptoc-eng.html.

7 Related Work

Results that show the soundness of the Dolev-Yao model with respect to the computational model, e.g. [23,28,38], make it possible to use Dolev-Yao provers in order to prove protocols in the computational model. However, these results have limitations, in particular in terms of allowed cryptographic primitives (they must satisfy strong security properties so that they correspond to Dolev-Yao style primitives), and they require some restrictions on protocols (such as the absence of key cycles).

Several frameworks exist for formalizing proofs of protocols in the computational model. Backes, Pfitzmann, and Waidner [7,9,10] have designed an abstract cryptographic library including symmetric and public-key encryption, message authentication codes, signatures, and nonces and shown its soundness with respect to computational primitives, under arbitrary active attacks. Backes and Pfitzmann [8] relate the computational and formal notions of secrecy in the framework of this library. Recently, this framework has been used for a computationally-sound machine-checked proof of the Needham-Schroeder-Lowe protocol [46]. Canetti [21] introduced the notion of universal composability. With Herzog [22], they show how a Dolev-Yao-style symbolic analysis can be used to prove security properties of protocols within the framework of universal composability, for a restricted class of protocols using public-key encryption as only cryptographic primitive. Then, they use the automatic Dolev-Yao verification tool Proverif [17] for verifying protocols in this framework. Lincoln, Mateus, Mitchell, Mitchell, Ramanathan, Scedrov, and Teague [33,34,36,39,43] developed a probabilistic polynomial-time calculus for the analysis of security protocols. They define a notion of process equivalence for this calculus, derive compositionality properties, and define an equational proof system for this calculus. Datta, Derek, Mitchell, Shmatikov, and Turuani [24] have designed a computationally sound logic that enables them to prove computational security properties using a logical deduction system. The frameworks mentioned in this paragraph can be used to prove security properties of protocols in the computational sense, but, except for [22] which relies on a Dolev-Yao prover and for the machine-checked proofs of [46], they have not been mechanized up to now, as far as we know.

Laud [30] designed an automatic analysis for proving secrecy for protocols using shared-key encryption, with passive adversaries. He extended it [31] to active adversaries, but with only one session of the protocol. This work is the closest to ours. We extend it considerably by handling more primitives and a polynomial number of sessions.

Recently, Laud [32] designed a type system for proving security protocols in the computational model. This type system handles shared-key and public-key encryption, with an unbounded number of sessions. This system relies on the Backes-Pfitzmann-Waidner library. A type inference algorithm is given in [6].

Barthe, Cerderquist, and Tarento [11, 47] have formalized the generic model and the random oracle model in the interactive theorem prover Coq, and proved signature schemes in this framework. In contrast to our specialized prover, proofs in generic interactive theorem provers require a lot of human ef-

fort, in order to build a detailed enough proof for the theorem prover to check it.

Halevi [26] explains that implementing an automatic prover based on sequences of games would be useful and suggests ideas in this direction, but does not actually implement one.

8 Conclusion

This paper presents a prover for security protocols sound in the computational model. This prover works with no or very little help from the user, can handle a wide variety of cryptographic primitives in a generic way, and produces proofs valid for a polynomial number of sessions in the presence of an active adversary. Thus, it represents important progress with respect to previous work in this area.

We have recently extended our prover to provide exact security proofs (that is, proofs with an explicit probability of an attack, instead of the asymptotic result that this probability is negligible) [19] and to prove correspondence assertions [18]. In the future, it would also be interesting to handle even more cryptographic primitives, such as Diffie-Hellman key agreements. (The equivalence $!^{i\leq n}\mathsf{new}\ a : T; \mathsf{new}\ b : T; (() \rightarrow g^a, () \rightarrow g^b, () \rightarrow g^{ab}) \approx\ !^{i\leq n}\mathsf{new}\ a : T; \mathsf{new}\ b : T; \mathsf{new}\ c : T; (() \rightarrow g^a, () \rightarrow g^b, () \rightarrow g^c)$ models the decisional Diffie-Hellman assumption. However, it is not sufficient for our prover to handle protocols that use Diffie-Hellman key agreements, because the corresponding cryptographic transformation would require g^{ab} to be formed only for a and b chosen in the same copy of a single replicated process, which is typically not the case: a and b are chosen by two different participants of the protocol. So a more involved equivalence is needed, and in fact the language of equivalences that we use to specify the security properties of primitives will need to be extended.)

The essential idea of simulating proofs by sequences of games in an automatic tool can be applied to any protocol or cryptographic scheme. However, our tool applies in a fairly direct way the security assumptions on the primitives and cannot perform deep mathematical reasoning. Therefore, it is best suited for proving security protocols that use rather high-level primitives such as encryption and signatures. It is more limited for proving the security of such primitives from lower-level primitives, since more subtle mathematical arguments are often needed.

Acknowledgments

I warmly thank David Pointcheval for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him. I also thank Jacques Stern for initiating this work. This work was partly supported by the ANR project ARA SSIA Formacrypt.

References

[1] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In N. Kobayashi and B. Pierce, editors, *Theoretical Aspects of Computer Software (TACS'01)*, volume 2215 of *Lecture Notes on Computer Science*, pages 82–94, Sendai, Japan, Oct. 2001. Springer.

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[4] M. Abdalla, P.-A. Fouque, and D. Pointcheval. Password-based authenticated key exchange in the three-party setting. *IEE Proceedings Information Security*, 153(1):27–39, Mar. 2006.

[5] P. Adão, G. Bana, J. Herzog, and A. Scedrov. Soundness of formal encryption in the presence of key-cycles. In S. de Capitani di Vimercati, P. Syverson, and D. Gollmann, editors, *Proceedings of the 10th European Symposium On Research In Computer Security (ESORICS 2005)*, volume 3679 of *Lecture Notes on Computer Science*, pages 374–396, Milan, Italy, Sept. 2005. Springer.

[6] M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *Proceedings of 13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 370–379, Alexandria, VA, Nov. 2006. ACM.

[7] M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *17th IEEE Computer Security Foundations Workshop*, pages 204–218, Pacific Grove, CA, June 2004. IEEE.

[8] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. *IEEE Transactions on Dependable and Secure Computing*, 2(2):109–123, Apr. 2005.

[9] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *10th ACM conference on Computer and communication security (CCS'03)*, pages 220–230, Washington D.C., Oct. 2003. ACM.

[10] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In E. Snekkenes and D. Gollman, editors, *Computer Security - ESORICS 2003, 8th European Symposium on Research in Computer Security*, volume 2808 of *Lecture Notes on Computer Science*, pages 271–290, Gjøovik, Norway, Oct. 2003. Springer.

[11] G. Barthe, J. Cederquist, and S. Tarento. A machine-checked formalization of the generic model and the random oracle model. In D. Basin and M. Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning (IJCAR'04)*, volume 3097 of *Lecture Notes on Computer Science*, pages 385–399, Cork, Ireland, July 2004. Springer.

[12] M. Baudet, V. Cortier, and S. Kremer. Computationally sound implementations of equational theories against passive adversaries. In L. Caires and L. Monteiro, editors, *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, volume 3580 of *Lecture Notes on Computer Science*, pages 652–663, Lisboa, Portugal, July 2005. Springer.

[13] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Symposium on Foundations of Computer Science (FOCS'97)*, pages 394–403, Miami Beach, Florida, Oct. 1997. IEEE. Full paper available at `http://www-cse.ucsd.edu/users/mihir/papers/sym-enc.html`.

[14] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO 1998*, volume 1462 of *Lecture Notes on Computer Science*, pages 26–45, Santa Barbara, California, USA, Aug. 1998. Springer.

[15] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, Dec. 2000.

[16] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *Advances in Cryptology – Eurocrypt 2006 Proceedings*, volume 4004 of *Lecture Notes on Computer Science*, pages 409–426, Saint Petersburg, Russia, May 2006. Springer. Extended version available at `http://eprint.iacr.org/2004/331`.

[17] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[18] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 97–111, Venice, Italy, July 2007. IEEE. Extended version available as ePrint Report 2007/128, `http://eprint.iacr.org/2007/128`.

[19] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In C. Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes on Computer Science*, pages 537–554, Santa Barbara, CA, Aug. 2006. Springer.

[20] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

[21] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, Las Vegas, Nevada, Oct. 2001. IEEE. An updated version is available at Cryptology ePrint Archive, `http://eprint.iacr.org/2000/067`.

[22] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In S. Halevi and T. Rabin, editors, *Proceedings, Theory of Cryptography Conference (TCC'06)*, volume 3876 of *Lecture Notes on Computer Science*, pages 380–403, New York, NY, Mar. 2006. Springer. Extended version available at `http://eprint.iacr.org/2004/334`.

[23] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In M. Sagiv, editor, *Proc. 14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes on Computer Science*, pages 157–171, Edimbourg, U.K., Apr. 2005. Springer.

[24] A. Datta, A. Derek, J. C. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In L. Caires and L. Monteiro, editors, *ICALP 2005: the 32nd International Colloquium on Automata, Languages and Programming*, volume 3580 of *Lecture Notes on Computer Science*, pages 16–29, Lisboa, Portugal, July 2005. Springer.

[25] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, Aug. 1981.

[26] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, June 2005. Available at `http://eprint.iacr.org/2005/181`.

[27] J. Herzog. A computational interpretation of Dolev-Yao adversaries. *Theoretical Computer Science*, 340:57–81, June 2005.

[28] R. Janvier, Y. Lakhnech, and L. Mazaré. Completing the picture: Soundness of formal encryption in the presence of active adversaries. In M. Sagiv, editor, *Proc. 14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes on Computer Science*, pages 172–185, Edimbourg, U.K., Apr. 2005. Springer.

[29] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, U.K., 1970.

[30] P. Laud. Handling encryption in an analysis for secure information flow. In P. Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP'03*, volume 2618 of *Lecture Notes on Computer Science*, pages 159–173, Warsaw, Poland, Apr. 2003. Springer.

[31] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *IEEE Symposium on Security and Privacy*, pages 71–85, Oakland, California, May 2004.

[32] P. Laud. Secrecy types for a simulatable cryptographic library. In *12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 26–35, Alexandria, VA, Nov. 2005. ACM.

[33] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Computer and Communication Security (CCS-5)*, pages 112–121, San Francisco, California, Nov. 1998.

[34] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 World Congress On Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes on Computer Science*, pages 776–793, Toulouse, France, Sept. 1999. Springer.

[35] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes on Computer Science*, pages 147–166. Springer, 1996.

[36] P. Mateus, J. Mitchell, and A. Scedrov. Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. In R. Amadio and D. Lugiez, editors, *CONCUR 2003 - Concurrency Theory, 14-th International Conference*, volume 2761 of *Lecture Notes on Computer Science*, pages 327–349, Marseille, France, Sept. 2003. Springer.

[37] D. Micciancio and B. Warinschi. Completeness theorems for the Abadi-Rogaway logic of encrypted expressions. *Journal of Computer Security*, 12(1):99–129, 2004.

[38] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In M. Naor, editor, *Theory of Cryptography Conference (TCC'04)*, volume 2951 of *Lecture Notes on Computer Science*, pages 133–151, Cambridge, MA, USA, Feb. 2004. Springer.

[39] J. C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 353(1–3):118–164, Mar. 2006.

[40] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

[41] R. M. Needham and M. D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.

[42] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[43] A. Ramanathan, J. Mitchell, A. Scedrov, and V. Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In I. Walukiewicz, editor, *FOSSACS 2004 - Foundations of Software Science and Computation Structures*, volume 2987 of *Lecture Notes on Computer Science*, pages 468–483, Barcelona, Spain, Mar. 2004. Springer.

[44] V. Shoup. A proposal for an ISO standard for public-key encryption, Dec. 2001. ISO/IEC JTC 1/SC27.

[45] V. Shoup. OAEP reconsidered. *Journal of Cryptology*, 15(4):223–249, Sept. 2002.

[46] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *19th IEEE Computer Security Foundations Workshop (CSFW-19)*, pages 153–166, Venice, Italy, July 2006. IEEE.

[47] S. Tarento. Machine-checked security proofs of cryptographic signature schemes. In S. de Capitani di Vimercati, P. Syverson, and D. Gollmann, editors, *Proceedings of the 10th European Symposium On Research In Computer Security (ESORICS 2005)*, volume 3679 of *Lecture Notes on Computer Science*, pages 140–158, Milan, Italy, Sept. 2005. Springer.

[48] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 1993.

Appendices

A Type System

In this section, we define the type system, used in our calculus to check that bitstrings belong to the expected type.

To be able to type variable accesses used not under their definition (such accesses are guarded by a find construct), the typechecking algorithm proceeds in two passes. In the first pass, we build a type environment $\mathcal{E}$, which maps variable names x to types $T_1 \times \ldots \times T_m \rightarrow T$, where $T_1, \ldots, T_m$ are the interval types of the indices of x, and T is the type of $x[i_1, \ldots, i_m]$. This type environment is built as follows:

- If x is defined by $\mathsf{new}\ x[i_1, \ldots, i_m] : T$, $\mathsf{let}\ x[i_1, \ldots, i_m] : T = M$, or $c[M_1, \ldots, M_l](\ldots, x[i_1, \ldots, i_m] : T, \ldots)$, and the replications above this subprocess are $!^{i_1 \leq n_1}$, $\ldots$, $!^{i_m \leq n_m}$, then $\mathcal{E}(x) = [1, n_1] \times \ldots \times [1, n_m] \rightarrow T$.
- If u is defined by $\mathsf{find} \ldots \oplus \ldots u[i_1, \ldots, i_m] \leq n \ldots\ \mathsf{suchthat}\ \mathsf{defined}(\ldots) \wedge \ldots \mathsf{then} \ldots \oplus \ldots$ and the replications above this find are $!^{i_1 \leq n_1}$, $\ldots$, $!^{i_m \leq n_m}$, then $\mathcal{E}(u) = [1, n_1] \times \ldots \times [1, n_m] \rightarrow [1, n]$.

We require that all definitions of the same variable x yield the same value of $\mathcal{E}(x)$, so that $\mathcal{E}$ is properly defined.

A process can then be typechecked in the type environment $\mathcal{E}$ using the rules of Figure 3. This figure defines three judgments:

$$\frac{\mathcal{E}(i) = T}{\mathcal{E} \vdash i : T} \quad \text{(TIndex)}$$

$$\frac{\mathcal{E}(x) = T_1 \times \ldots \times T_m \to T \qquad \forall j \leq m, \mathcal{E} \vdash M_j : T_j}{\mathcal{E} \vdash x[M_1, \ldots, M_m] : T} \quad \text{(TVar)}$$

$$\frac{f : T_1 \times \ldots \times T_m \to T \qquad \forall j \leq m, \mathcal{E} \vdash M_j : T_j}{\mathcal{E} \vdash f(M_1, \ldots, M_m) : T} \quad \text{(TFun)}$$

$$\mathcal{E} \vdash 0 \quad \text{(TNil)}$$

$$\frac{\mathcal{E} \vdash Q \qquad \mathcal{E} \vdash Q'}{\mathcal{E} \vdash Q \mid Q'} \quad \text{(TPar)}$$

$$\frac{\mathcal{E}[i \mapsto [1, n]] \vdash Q}{\mathcal{E} \vdash !^{i \leq n} Q} \quad \text{(TRepl)}$$

$$\frac{\mathcal{E} \vdash Q}{\mathcal{E} \vdash \mathsf{newChannel}\ c; Q} \quad \text{(TNewChannel)}$$

$$\frac{\forall j \leq l, \mathcal{E} \vdash M_j : T'_j \qquad \forall j \leq k, \mathcal{E} \vdash x_j[\widetilde{i}] : T_j \qquad \mathcal{E} \vdash P}{\mathcal{E} \vdash c[M_1, \ldots, M_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P} \quad \text{(TIn)}$$

$$\frac{\forall j \leq l, \mathcal{E} \vdash M_j : T'_j \qquad \forall j \leq k, \mathcal{E} \vdash N_j : T_j \qquad \mathcal{E} \vdash Q}{\mathcal{E} \vdash \overline{c[M_1, \ldots, M_l]}\langle N_1, \ldots, N_k \rangle; Q} \quad \text{(TOut)}$$

$$\frac{T \text{ fixed-length type} \qquad \mathcal{E} \vdash x[\widetilde{i}] : T \qquad \mathcal{E} \vdash P}{\mathcal{E} \vdash \mathsf{new}\ x[\widetilde{i}] : T; P} \quad \text{(TNew)}$$

$$\frac{\mathcal{E} \vdash M : T \qquad \mathcal{E} \vdash x[\widetilde{i}] : T \qquad \mathcal{E} \vdash P}{\mathcal{E} \vdash \mathsf{let}\ x[\widetilde{i}] : T = M \ \mathsf{in}\ P} \quad \text{(TLet)}$$

$$\frac{\begin{array}{c}\forall j \leq m, \forall k \leq m_j, \mathcal{E} \vdash u_{jk}[\widetilde{i}] : [1, n_{jk}] \\ \forall j \leq m, \forall k \leq l_j, \mathcal{E} \vdash M_{jk} : T_{jk} \\ \forall j \leq m, \mathcal{E} \vdash M_j : \mathit{bool} \qquad \forall j \leq m, \mathcal{E} \vdash P_j \qquad \mathcal{E} \vdash P\end{array}}{\begin{array}{c}\mathcal{E} \vdash \mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat} \\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P\end{array}} \quad \text{(TFind)}$$

Figure 3: Typing rules

- $\mathcal{E} \vdash M : T$ means that term M has type T in environment $\mathcal{E}$.
- $\mathcal{E} \vdash P$ and $\mathcal{E} \vdash Q$ mean that the output process P and the input process Q are well-typed in environment $\mathcal{E}$, respectively.

In $x[M_1, \ldots, M_m]$, $M_1, \ldots, M_m$ must be of the suitable interval type. When $f(M_1, \ldots, M_m)$ is called and $f : T_1 \times \ldots \times T_m \to T$, M_j must be of type T_j, and $f(M_1, \ldots, M_m)$ is then of type T. The type system requires each subterm to be well-typed. Furthermore, in let $x : T = M$ in P, M must be of type T. In

$$\mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$$

M_j is of type *bool* for all $j \leq m$. In $!^{i \leq n} Q$, i is of type $[1, n]$ in Q. For new $x[\widetilde{i}] : T$, T must be a fixed-length type.

We say that an occurrence of a term M in a process Q is of type T when $\mathcal{E} \vdash M : T$ where $\mathcal{E}$ is the type environment of Q extended with $i \mapsto [1, n]$ for each replication $!^{i \leq n}$ above M in Q.

B Formal Semantics

B.1 Definition of the Semantics

The formal semantics of our calculus is presented in Figures 4 and 5. In this figure and in the rest of the appendix, we use $\uplus$ for multiset union. When S is a multiset, $S(x)$ is the number of elements of S equal to x. A semantic configuration is a quadruple $E, P, \mathcal{Q}, \mathcal{C}$, where E is an environment mapping array cells to bitstrings or $\perp$, P is the output process currently scheduled, $\mathcal{Q}$ is the multiset of input processes running in parallel with P, $\mathcal{C}$ is the set of channels already created. The semantics is defined by reduction rules of the form $E, P, \mathcal{Q}, \mathcal{C} \xrightarrow{p}_{\eta, t} E', P', \mathcal{Q}', \mathcal{C}'$ meaning that $E, P, \mathcal{Q}, \mathcal{C}$ reduces to $E', P', \mathcal{Q}', \mathcal{C}'$ with probability p, when the security parameter is η. The value of the security parameter is often omitted to lighten the notation. The index t just serves in distinguishing reductions that yield the same configuration with the same probability in different ways, so that the probability of a certain reduction can be computed correctly:

$$\Pr[E, P, \mathcal{Q}, \mathcal{C} \to_\eta E', P', \mathcal{Q}', \mathcal{C}'] = \sum_{E, P, \mathcal{Q}, \mathcal{C} \xrightarrow{p}_{\eta, t} E', P', \mathcal{Q}', \mathcal{C}'} p$$

The probability of a trace is computed as follows:

$$\Pr[E_1, P_1, \mathcal{Q}_1, \mathcal{C}_1 \to_\eta \ldots \to_\eta E'_m, P'_m, \mathcal{Q}'_m, \mathcal{C}'_m] = \prod_{j=1}^{m-1} \Pr[E_j, P_j, \mathcal{Q}_j, \mathcal{C}_j \to_\eta E'_{j+1}, P'_{j+1}, \mathcal{Q}'_{j+1}, \mathcal{C}'_{j+1}]$$

We define an auxiliary relation for evaluating terms: $E, M \Downarrow_\eta a$, or simply $E, M \Downarrow a$, means that the term M evaluates to the bitstring a in environment E. Rule (Cst) simply evaluates constants to themselves. This rule serves for replication indices, which are substituted with constant values when reducing the replication. Rule (Var) looks for the value of the array

Terms and find conditions:

$$E, a \Downarrow a \qquad \text{(Cst)}$$

$$\frac{\forall j \leq m, E, M_j \Downarrow a_j \qquad x[a_1,\dots,a_m] \in \mathrm{Dom}(E)}{E, x[M_1,\dots,M_m] \Downarrow E(x[a_1,\dots,a_m])} \qquad \text{(Var)}$$

$$\frac{\begin{array}{c}\forall j \leq m, E, M_j \Downarrow a_j \qquad f : T_1 \times \dots \times T_m \to T\\ \forall j \leq m, a_j \in I_\eta(T_j)\end{array}}{E, f(M_1,\dots,M_m) \Downarrow I_\eta(f)(a_1,\dots,a_m)} \qquad \text{(Fun)}$$

$$\frac{\neg \forall k \leq l, \exists a_k, E, M_k \Downarrow a_k}{E, (\mathsf{defined}(M_1,\dots,M_l) \wedge M) \Downarrow \mathrm{false}} \qquad \text{(Def1)}$$

$$\frac{\forall k \leq l, \exists a_k, E, M_k \Downarrow a_k \qquad E, M \Downarrow a \qquad a \in \{\mathrm{false}, \mathrm{true}\}}{E, (\mathsf{defined}(M_1,\dots,M_l) \wedge M) \Downarrow a} \qquad \text{(Def2)}$$

Input processes:

$$E, \{0\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \mathcal{Q}, \mathcal{C} \qquad \text{(Nil)}$$

$$E, \{Q_1 \mid Q_2\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \{Q_1, Q_2\} \uplus \mathcal{Q}, \mathcal{C} \qquad \text{(Par)}$$

$$E, \{!^{i \leq n} Q\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \{Q\{a/i\} \mid a \in [1, I_\eta(n)]\} \uplus \mathcal{Q}, \mathcal{C} \qquad \text{(Repl)}$$

$$\frac{c' \notin \mathcal{C}}{\begin{array}{c}E, \{\mathsf{newChannel}\ c; Q\} \uplus \mathcal{Q}, \mathcal{C}\\ \rightsquigarrow E, \{Q\{c'/c\}\} \uplus \mathcal{Q}, \mathcal{C} \cup \{c'\}\end{array}} \qquad \text{(NewChannel)}$$

$$\frac{\forall j \leq l, E, M_j \Downarrow a_j}{\begin{array}{c}E, \{c[M_1,\dots,M_l](x_1[\widetilde{a'}] : T_1, \dots, x_k[\widetilde{a'}] : T_k); P\} \uplus \mathcal{Q}, \mathcal{C}\\ \rightsquigarrow E, \{c[a_1,\dots,a_l](x_1[\widetilde{a'}] : T_1, \dots, x_k[\widetilde{a'}] : T_k); P\} \uplus \mathcal{Q}, \mathcal{C}\end{array}} \qquad \text{(Input)}$$

$\mathrm{reduce}(E, \mathcal{Q}, \mathcal{C})$ is the normal form of $E, \mathcal{Q}, \mathcal{C}$ by $\rightsquigarrow$

Figure 4: Semantics (1)

Output processes:

$$\frac{T \text{ fixed-length type} \qquad a \in I_\eta(T)}{E, \mathsf{new}\ x[\widetilde{a'}] : T; P, \mathcal{Q}, \mathcal{C} \xrightarrow{\frac{1}{|I_\eta(T)|}}_{N(a)} E[x[\widetilde{a'}] \mapsto a], P, \mathcal{Q}, \mathcal{C}} \qquad \text{(New)}$$

$$\frac{E, M \Downarrow a \qquad a \in I_\eta(T)}{E, \mathsf{let}\ x[\widetilde{a'}] : T = M \ \mathsf{in}\ P, \mathcal{Q}, \mathcal{C} \xrightarrow{1}_{L} E[x[\widetilde{a'}] \mapsto a], P, \mathcal{Q}, \mathcal{C}} \qquad \text{(Let)}$$

$$\frac{\begin{array}{c}\forall j \leq m, \forall \widetilde{v} \leq \widetilde{n_j}, E[\widetilde{u_j}[\widetilde{a'}] \mapsto \widetilde{v}], (D_j \wedge M_j) \Downarrow a_{j,\widetilde{v}}\\ S = \{j, \widetilde{v} \mid a_{j,\widetilde{v}} = \mathrm{true}\} \qquad a_{j_0,\widetilde{v_0}} = \mathrm{true}\\ E_{j_0,\widetilde{v_0}} = E[\widetilde{u_{j_0}}[\widetilde{a'}] \mapsto \widetilde{v_0}]\end{array}}{\begin{array}{c}E, \mathsf{find}\ (\bigoplus_{j=1}^m \widetilde{u_j}[\widetilde{a'}] \leq \widetilde{n_j}\ \mathsf{suchthat}\ D_j \wedge M_j\ \mathsf{then}\ P_j)\\ \mathsf{else}\ P, \mathcal{Q}, \mathcal{C} \xrightarrow{\mathrm{among}(S)}_{F1(j_0,\widetilde{v_0})} E_{j_0,\widetilde{v_0}}, P_{j_0}, \mathcal{Q}, \mathcal{C}\end{array}} \qquad \text{(Find1)}$$

$$\frac{\forall j \leq m, \forall \widetilde{v} \leq \widetilde{n_j}, E[\widetilde{u_j}[\widetilde{a'}] \mapsto \widetilde{v}], (D_j \wedge M_j) \Downarrow \mathrm{false}}{\begin{array}{c}E, \mathsf{find}\ (\bigoplus_{j=1}^m \widetilde{u_j}[\widetilde{a'}] \leq \widetilde{n_j}\ \mathsf{suchthat}\ D_j \wedge M_j\ \mathsf{then}\ P_j)\\ \mathsf{else}\ P, \mathcal{Q}, \mathcal{C} \xrightarrow{1}_{F2} E, P, \mathcal{Q}, \mathcal{C}\end{array}} \qquad \text{(Find2)}$$

$$\frac{\begin{array}{c}\forall j \leq l, E, M_j \Downarrow a_j \qquad \forall j \leq k, E, N_j \Downarrow b_j\\ E, \mathcal{Q}', \mathcal{C}' = \mathrm{reduce}(E, \{Q''\}, \mathcal{C})\\ S = \{Q \in \mathcal{Q} \mid \text{for some } x'_1, \dots, x'_k, \widetilde{a''}, T'_1, \dots, T'_k, P',\\ Q = c[a_1,\dots,a_l](x'_1[\widetilde{a''}] : T'_1, \dots, x'_k[\widetilde{a''}] : T'_k).P'\}\\ Q_0 = c[a_1,\dots,a_l](x_1[\widetilde{a'}] : T_1, \dots, x_k[\widetilde{a'}] : T_k).P \in S\\ \forall j \leq k, b'_j = b_j \& (2^{\mathrm{maxlen}_\eta(c)} - 1) \in I_\eta(T_j)\end{array}}{\begin{array}{c}E, \overline{c[M_1,\dots,M_l]}\langle N_1,\dots,N_k\rangle.Q'', \mathcal{Q}, \mathcal{C}\\ \xrightarrow{S(Q_0) \times \mathrm{among}(S)}_{O(Q_0)}\\ E[x_1[\widetilde{a'}] \mapsto b'_1, \dots, x_k[\widetilde{a'}] \mapsto b'_k], P, \mathcal{Q} \uplus \mathcal{Q}' \setminus \{Q_0\}, \mathcal{C}'\end{array}} \qquad \text{(Output)}$$

Figure 5: Semantics (2)

variable in the environment. Rule (Fun) evaluates the function call. Rules (Def1) and (Def2) evaluate conditions of find: When some M_k is not defined, $\mathsf{defined}(M_1,\dots,M_l) \wedge M$ returns false by (Def1). Otherwise, it returns the boolean value of M by (Def2).

We use an auxiliary reduction relation $\rightsquigarrow_\eta$, or simply $\rightsquigarrow$, for reducing input processes. This relation transforms configurations of the form $E, \mathcal{Q}, \mathcal{C}$. Rule (Nil) removes nil processes. Rules (Par) and (Repl) expand parallel compositions and replications, respectively. Rule (NewChannel) creates a new channel and adds it to $\mathcal{C}$. Semantic configurations are considered equivalent modulo renaming of channels in $\mathcal{C}$, so that a single semantic configuration is obtained after applying (NewChannel). Rule (Input) evaluates the terms in the input channel. The input itself is not executed: the communication is done by the (Output) rule. The relation $\rightsquigarrow$ is convergent (confluent and terminating), so it has normal forms. Since processes in $\mathcal{Q}$ in configurations $E, P, \mathcal{Q}, \mathcal{C}$ are in normal form by $\rightsquigarrow$, they always start with an input.

Rules (New) to (Find2) simply reduce the scheduled process. As explained in the footnote page 275, we use an approximately uniform probability distribution for choosing an element among a set S when $m = |S|$ is not a power of 2. Let k be the smallest integer such that $2^k \geq m$. We choose a random integer r uniformly among $[0, 2^{k+f(\eta)} - 1]$ for a certain function f. When r is in $[0, (2^{k+f(\eta)} \operatorname{div} m \times m) - 1]$, $r \mod m$ returns a random integer in $[0, m-1]$, with the same probability for all elements of $[0, m-1]$. When r is in $[2^{k+f(\eta)} \operatorname{div} m \times m, 2^{k+f(\eta)} - 1]$, we can do anything; we choose to block. The probability of being in this case is $(2^{k+f(\eta)} \mod m)/2^{k+f(\eta)} \leq m/2^{k+f(\eta)} \leq 1/2^{f(\eta)}$, so it can be made as small as we wish by choosing $f(\eta)$ large enough. We choose $f(\eta) \geq \alpha\eta$ for some $\alpha > 0$, so that it is negligible. The probability of choosing each element of S is then $\mathrm{among}(S) = \frac{2^{k+f(\eta)} \operatorname{div} m}{2^{k+f(\eta)}}$. Then $\mathrm{among}(S)$ approximates $1/m$. Rules (Find1) and (Find2) evaluate a find. They compute the value of all conditions $D_j \wedge M_j$ of this find for all possible values $\widetilde{v}$ of the indices $\widetilde{u_j}[\widetilde{a'}]$. When all these conditions are false, rule (Find2) executes the else branch of the find. When at least one of these conditions is true, rule (Find1) chooses one such true case (for $j = j_0$ and $\widetilde{v} = \widetilde{v_0}$) with approximately uniform probability, and executes the corresponding then branch of the find.

Rule (Output) performs communications: it evaluates the terms in the channel and the sent messages, selects an input on the desired channel randomly, and immediately executes the communication. The scheduled process after this rule is the receiving process. (The process blocks if no suitable input is available.)

The initial configuration for running process Q_0 is $\mathrm{initConfig}(Q_0) = \emptyset, \overline{start}\langle\rangle, \mathcal{Q}, \mathcal{C}$ where $\emptyset, \mathcal{Q}, \mathcal{C} = \mathrm{reduce}(\emptyset, \{Q_0\}, \mathrm{fc}(Q_0))$.

Definition 6 Let c be a channel name and a be a bitstring. We say that $E, P, \mathcal{Q}, \mathcal{C}$ *executes* $\overline{c}\langle a\rangle$ *immediately* when $P = \overline{c}\langle M\rangle.Q$ and $E, M \Downarrow a$ for some Q and M.

The probability that Q executes $\overline{c}\langle a\rangle$ is denoted $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a\rangle]$. When $c \in \mathrm{fc}(Q)$, $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \sum_{\mathcal{T}\in\mathbb{T}} \Pr[\mathcal{T}]$ where $\mathbb{T}$ is the set of traces $\mathrm{initConfig}(Q) \rightarrow_\eta \ldots \rightarrow_\eta E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ such that $E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ executes $\overline{c}\langle a\rangle$ immediately and for all $j < m$, $E_j, P_j, \mathcal{Q}_j, \mathcal{C}_j$ does not execute $\overline{c}\langle a\rangle$ immediately. When $c \notin \mathrm{fc}(Q)$, $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a\rangle] = 0$.

B.2 Each Variable is Defined at Most Once

In this section, we show that Invariant 1 implies that each array cell is assigned at most once during the execution of a process.

When S and S' are multisets, $\max(S, S')$ is the multiset such that $\max(S, S')(x) = \max(S(x), S'(x))$. We define the multiset of variable accesses that may be defined by a process as follows:

$$\begin{aligned}
&\mathit{Defined}(0) = \emptyset\\
&\mathit{Defined}(Q_1 \mid Q_2) = \mathit{Defined}(Q_1) \uplus \mathit{Defined}(Q_2)\\
&\mathit{Defined}(!^{i\leq n}Q) = \biguplus_{a\in[1,I_\eta(n)]} \mathit{Defined}(Q\{a/i\})\\
&\mathit{Defined}(\mathsf{newChannel}\ c; Q) = \mathit{Defined}(Q)\\
&\mathit{Defined}(c[M_1,\ldots,M_l](x_1[\widetilde{a}] : T_1,\ldots,x_k[\widetilde{a}] : T_k); P) =\\
&\quad \{x_j[\widetilde{a}] \mid j \leq k\} \uplus \mathit{Defined}(P)\\
&\mathit{Defined}(\overline{c[M_1,\ldots,M_l]}\langle N_1,\ldots,N_k\rangle; Q) = \mathit{Defined}(Q)\\
&\mathit{Defined}(\mathsf{new}\ x[\widetilde{a}] : T; P) = \{x[\widetilde{a}]\} \uplus \mathit{Defined}(P)\\
&\mathit{Defined}(\mathsf{let}\ x[\widetilde{a}] : T = M\ \mathsf{in}\ P) = \{x[\widetilde{a}]\} \uplus \mathit{Defined}(P)\\
&\mathit{Defined}(\mathsf{find}\ (\textstyle\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{a}] \leq \widetilde{n_j}\ \mathsf{suchthat defined}(M_{j1},\\
&\quad \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P) =\\
&\qquad \max(\max_{j=1}^{m}\{\widetilde{u_j}[\widetilde{a}]\} \uplus \mathit{Defined}(P_j), \mathit{Defined}(P))
\end{aligned}$$

We define $\mathit{Defined}(E) = \mathrm{Dom}(E)$, $\mathit{Defined}(E, P, \mathcal{Q}, \mathcal{C}) = \mathit{Defined}(E) \uplus \mathit{Defined}(P) \uplus \biguplus_{Q\in\mathcal{Q}} \mathit{Defined}(Q)$.

Invariant 4 (Single definition, for executing games) The semantic configuration $E, P, \mathcal{Q}, \mathcal{C}$ satisfies Invariant 4 if and only if $\mathit{Defined}(E, P, \mathcal{Q}, \mathcal{C})$ does not contain duplicate elements.

Lemma 3 *If* Q_0 *satisfies Invariant 1, then* $\mathrm{initConfig}(Q_0)$ *satisfies Invariant 4.*

Lemma 4 *If* $E, P, \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}'$ *with* $p > 0$ *and* $E, P, \mathcal{Q}, \mathcal{C}$ *satisfies Invariant 4, then so does* $E', P', \mathcal{Q}', \mathcal{C}'$.

Proof sketch We show by cases following the definition of $\xrightarrow{p}_t$ that if $E, P, \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}'$ then $\mathit{Defined}(E, P, \mathcal{Q}, \mathcal{C}) \subseteq \mathit{Defined}(E', P', \mathcal{Q}', \mathcal{C}')$. The result follows. □

Therefore, if Q_0 satisfies Invariant 1, then each variable is defined at most once for each value of its array indices in a trace of Q_0. Indeed, by Invariant 4, just before executing a definition of $x[\widetilde{a}]$, $\mathit{Defined}(E, P, \mathcal{Q}, \mathcal{C})$ does not contain duplicate elements, so $x[\widetilde{a}] \notin \mathrm{Dom}(E)$ since $x[\widetilde{a}] \in \mathit{Defined}(P) \uplus \mathit{Defined}(\mathcal{Q})$.

B.3 Variables are Defined Before Being Used

In this section, we show that Invariant 2 implies that all variables are defined before being used. In order to show this property, we use the following invariant:

Invariant 5 (Defined variables, for executing games) The semantic configuration $E, P, \mathcal{Q}, \mathcal{C}$ satisfies Invariant 5 if and only if every occurrence of a variable access $x[M_1,\ldots,M_m]$ in P or $\mathcal{Q}$ is either

- present in $\mathrm{Dom}(E)$: for all $j \leq m$, $E, M_j \Downarrow a_j$ and $x[a_1,\ldots,a_m] \in \mathrm{Dom}(E)$;
- or syntactically under the definition of $x[M_1,\ldots,M_m]$ (in which case for all $j \leq m$, M_j is a constant or variable replication index);
- or in a defined condition in a find process;
- or in M'_j or P_j in a process of the form find $(\bigoplus_{j=1}^{m''} \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$ suchthat defined$(M'_{j1},\ldots,M'_{jl_j}) \wedge M'_j$ then P_j else P where for some $k \leq l_j$, $x[M_1,\ldots,M_m]$ is a subterm of M'_{jk}.

Lemma 5 *If* Q_0 *satisfies Invariant 2, then* $\mathrm{initConfig}(Q_0)$ *satisfies Invariant 5.*

Lemma 6 *If* $E, P, \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}'$ *with* $p > 0$ *and* $E, P, \mathcal{Q}, \mathcal{C}$ *satisfies Invariant 5, then so does* $E', P', \mathcal{Q}', \mathcal{C}'$.

Proof sketch If $x[M_1,\ldots,M_m]$ is in the second case of Invariant 5, and we execute the definition of $x[M_1,\ldots,M_m]$, then for all $j \leq m$, M_j is a constant replication index and $x[M_1,\ldots,M_m]$ is added to $\mathrm{Dom}(E)$ by rules (New), (Let), (Find1), or (Output), so it moves to the first case of Invariant 5.

If $x[M_1,\ldots,M_m]$ is in the third case of Invariant 5, and we execute the corresponding find, this access to x simply disappears.

If $x[M_1,\ldots,M_m]$ is in the last case of Invariant 5, and we execute the find selecting branch j, then $x[M_1,\ldots,M_m]$ is a subterm of M'_{jk} for $k \leq l_j$. We show by induction on M that, if $E, M \Downarrow a$, then for all subterms $x[M_1,\ldots,M_m]$ of M, for all $j' \leq m$, $E, M_{j'} \Downarrow a_{j'}$ and $x[a_1,\ldots,a_m]$ is in $\mathrm{Dom}(E)$. Therefore, by hypothesis of the semantic rule for find, for all $j' \leq m$, $E, M_{j'} \Downarrow a_{j'}$ and $x[a_1,\ldots,a_m]$ is in $\mathrm{Dom}(E)$. So $x[M_1,\ldots,M_m]$ also moves to the first case of Invariant 5.

In all other cases, the situation remains unchanged. □

Therefore, if Q_0 satisfies Invariant 2, then in traces of Q_0, the test $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$ in rule (Var) always succeeds, except when the considered term occurs in a defined condition of a find.

Indeed, consider an application of rule (Var), where the array access $x[M_1, \dots, M_m]$ is not in a defined condition of a find. Then, this array access is not under any variable definition or find, so for all $j \leq m$, $E, M_j \Downarrow a_j$ and $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$. Hence, the test $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$ succeeds.

B.4 Typing

In this section, we show that our type system is compatible with the semantics of the calculus, that is, we define a notion of typing for semantic configurations and show that typing is preserved by reduction (subject reduction). Finally, the property that semantic configurations are well-typed shows that certain conditions in the semantics always hold.

We say that $\mathcal{E} \vdash_\eta E$ if and only if $E(x[a_1, \dots, a_m]) = a$ implies $\mathcal{E}(x) = T_1 \times \dots \times T_m \to T$ with for all $j \leq m$, $a_j \in I_\eta(T_j)$ and $a \in I_\eta(T)$. We define $\mathcal{E} \vdash_\eta P$ as $\mathcal{E} \vdash P$, $\mathcal{E} \vdash_\eta Q$ as $\mathcal{E} \vdash Q$, and $\mathcal{E} \vdash_\eta M : T$ as $\mathcal{E} \vdash M : T$, with the additional rule $\mathcal{E} \vdash_\eta a : T$ if and only if $a \in I_\eta(T)$. (This rule is useful to type constant replication indices. In the formulas giving the typing rules, replication indices i may then also be constants a.) We say that $\mathcal{E} \vdash_\eta E, P, \mathcal{Q}, \mathcal{C}$ if and only if $\mathcal{E} \vdash_\eta E$, $\mathcal{E} \vdash_\eta P$, and for all $Q \in \mathcal{Q}$, $\mathcal{E} \vdash_\eta Q$. Similarly, $\mathcal{E} \vdash_\eta E, \mathcal{Q}, \mathcal{C}$ if and only if $\mathcal{E} \vdash_\eta E$ and for all $Q \in \mathcal{Q}$, $\mathcal{E} \vdash_\eta Q$.

Lemma 7 *If $\mathcal{E} \vdash_\eta E$, $\mathcal{E} \vdash_\eta M : T$, and $E, M \Downarrow a$, then $\mathcal{E} \vdash_\eta a : T$*

Proof sketch By induction on the derivation of $E, M \Downarrow a$. □

Lemma 8 *If $\mathcal{E} \vdash_\eta E, \mathcal{Q}, \mathcal{C}$ and $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$, then $\mathcal{E} \vdash_\eta E', \mathcal{Q}', \mathcal{C}'$.*
So, if $\mathcal{E} \vdash_\eta E, \mathcal{Q}, \mathcal{C}$, then $\mathcal{E} \vdash_\eta \mathrm{reduce}(E, \mathcal{Q}, \mathcal{C})$.

Proof sketch By cases on the derivation of $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$. In the case of the replication, we use a substitution lemma, noticing that $a \in I_\eta([1, n])$, so $\mathcal{E} \vdash_\eta a : [1, n]$. In the case of the input, we use Lemma 7. □

Lemma 9 *If $\mathcal{E} \vdash Q_0$, then $\mathcal{E} \vdash_\eta \mathrm{initConfig}(Q_0)$.*

Proof sketch By Lemma 8 and the previous definitions. □

Lemma 10 (Subject reduction) *If $\mathcal{E} \vdash_\eta E, P, \mathcal{Q}, \mathcal{C}$ and $E, P, \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}'$ with $p > 0$, then $\mathcal{E} \vdash_\eta E', P', \mathcal{Q}', \mathcal{C}'$.*

Proof sketch By cases on the derivation of $E, P, \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E', P', \mathcal{Q}', \mathcal{C}'$, using Lemmas 7 and 8. □

As an immediate consequence of Lemmas 9, 10, and 7, we obtain: if Q_0 satisfies Invariant 3, then in traces of Q_0, the tests T *fixed-length type* in rule (New), $a \in I_\eta(T)$ in rule (Let), $\forall j \leq m, a_j \in I_\eta(T_j)$ in rule (Fun), and the test $a \in \{\mathrm{false}, \mathrm{true}\}$ in rule (Def2) always succeed.

B.5 Runtime

Proposition 6 *For each process Q, there exists a probabilistic polynomial time Turing machine that simulates Q.*

Proof We give a very brief sketch of this proof here. We refer the reader to [39] for a more detailed proof for a different calculus; their proof could be adapted to our calculus.

The length of all bitstrings manipulated by processes is polynomial in the security parameter η. Indeed, by hypothesis, the length of received messages is limited by maxlen_η, so polynomial in the security parameter η. The length of random bitstrings is also polynomial in the security parameter by hypothesis on the types. Function symbols correspond to functions that run in polynomial time, so they output bitstrings of size polynomial in the size of their inputs, so also polynomial in the security parameter.

Since the number of copies generated by each replication is polynomial in the security parameter, the total number of executed instructions is polynomial in the security parameter. Moreover, it is easy to see that each instruction runs in polynomial time since bitstrings are of polynomial length. Therefore, processes run in polynomial time. □

C Simplification

In this section, we define the transformation **Simplify**, which is used to simplify games. The simplification proceeds as follows. It uses information from several sources: equations and rewrite rules given by user, that come in particular from algebraic properties of cryptographic primitives; facts that hold at certain points in the game due to the form of the game; dependency information obtained by two dependency analyses. (The global dependency analysis tracks which variables depend on any element of the array x at any program point. The local dependency analysis tracks which terms depend on the current cell of the array x, $x[\tilde{i}]$, at each program point.) The simplification algorithm uses this information in order to infer equalities using a Knuth-Bendix-like equational prover. The obtained equalities are used to simplify the game, by replacing a term with an equal term or by simplifying find when the system proves that some branches cannot be taken.

C.1 User-defined Rewrite Rules

The user can give two kinds of information:

- claims of the form $\forall x_1 : T_1, \dots, \forall x_m : T_m, M$ which mean that for all environments E, if for all $j \leq m$, $E(x_j) \in I_\eta(T_j)$, then $E, M \Downarrow \mathrm{true}$.

  Such claims must be well-typed, that is, $\{x_1 \mapsto T_1, \dots, x_m \mapsto T_m\} \vdash M : bool$.

  They are translated into rewrite rules as follows:

  – If M is of the form $M_1 = M_2$ and $\mathrm{var}(M_2) \subseteq \mathrm{var}(M_1)$, we generate the rewrite rule $\forall x_1 : T_1, \dots, \forall x_m : T_m, M_1 \to M_2$.

– If M is of the form $M_1 \neq M_2$, we generate the rewrite rules $\forall x_1 : T_1, \ldots, \forall x_m : T_m, (M_1 = M_2) \rightarrow \text{false}$, $\forall x_1 : T_1, \ldots, \forall x_m : T_m, (M_1 \neq M_2) \rightarrow \text{true}$. (Such rules are used for instance to express that different constants are different.)
– Otherwise, we generate the rewrite rule $\forall x_1 : T_1, \ldots, \forall x_m : T_m, M \rightarrow \text{true}$.

- claims of the form $\mathsf{new}\ y_1 : T'_1, \ldots, \mathsf{new}\ y_l : T'_l, \forall x_1 : T_1, \ldots, \forall x_m : T_m, M_1 \approx M_2$ with $\text{var}(M_2) \subseteq \text{var}(M_1)$. Informally, these claims mean that M_1 and M_2 evaluate to the same bitstring except in cases of negligible probability, provided that $y_1, \ldots, y_l$ are chosen randomly with uniform probability among $T'_1, \ldots, T'_l$ respectively, and that $x_1, \ldots, x_m$ are of type $T_1, \ldots, T_m$. ($x_1, \ldots, x_m$ may depend on $y_1, \ldots, y_l$.) Formally, a first approach is to define these claims as: for all polynomials q, there exists a negligible $p(\eta)$ such that

$$\begin{aligned}\max_{\mathcal{A}} \Pr[&E(y_1) \stackrel{R}{\leftarrow} I_\eta(T'_1); \ldots E(y_l) \stackrel{R}{\leftarrow} I_\eta(T'_l);\\ &(E(x_1), \ldots, E(x_m)) \leftarrow \mathcal{A}(E(y_1), \ldots, E(y_l));\\ &E, M_1 \Downarrow a; E, M_2 \Downarrow a' : a \neq a'] \leq p(\eta)\end{aligned}$$

where $\mathcal{A}$ is a probabilistic Turing machine running in time $q(\eta)$. However, this phrasing requires checking that the restrictions that create $y_1, \ldots, y_l$ are pairwise distinct, which is sometimes delicate. (It may depend on the value of array indices.) So we prefer the following definition, in which the substitution σ allows us to rename $y_1, \ldots, y_l$ to possibly equal variables $y'_1, \ldots, y'_{l'}$:

> The claim $\mathsf{new}\ y_1 : T'_1, \ldots, \mathsf{new}\ y_l : T'_l, \forall x_1 : T_1, \ldots, \forall x_m : T_m, M_1 \approx M_2$ means that for all polynomials q, there exists a negligible $p(\eta)$ such that, for all substitutions σ that map $y_1, \ldots, y_l$ to variables $y'_1, \ldots, y'_{l'}$ such that $\sigma\{y_1, \ldots, y_l\} = \{y'_1, \ldots, y'_{l'}\}$ and for all $j \leq l$, if $\sigma y_j = y'_{j'}$ then $T''_{j'} = T'_j$, we have
>
> $$\begin{aligned}\max_{\mathcal{A}} \Pr[&E(y'_1) \stackrel{R}{\leftarrow} I_\eta(T''_1); \ldots E(y'_{l'}) \stackrel{R}{\leftarrow} I_\eta(T''_{l'});\\ &(E(x_1), \ldots, E(x_m)) \leftarrow \mathcal{A}(E(y'_1), \ldots, E(y'_{l'}));\\ &E, \sigma M_1 \Downarrow a; E, \sigma M_2 \Downarrow a' : a \neq a'] \leq p(\eta)\end{aligned}$$
>
> where $\mathcal{A}$ is a probabilistic Turing machine running in time $q(\eta)$.

The claims need to be adapted to this definition. For instance, we write $\mathsf{new}\ x : T; \mathsf{new}\ y : T; \text{pkgen}(x) = \text{pkgen}(y) \approx x = y$ rather than $\mathsf{new}\ x : T; \mathsf{new}\ y : T; \text{pkgen}(x) = \text{pkgen}(y) \approx \text{false}$, since we may have $\text{pkgen}(x) = \text{pkgen}(y)$ with probability 1 when x and y are in fact the same variable.

The above claim must be well-typed, that is, $\{x_1 \mapsto T_1, \ldots, x_m \mapsto T_m, y_1 \mapsto T'_1, \ldots, y_l \mapsto T'_l\} \vdash M_1 = M_2$.

This claim is translated into the rewrite rule $\mathsf{new}\ y_1 : T'_1, \ldots, \mathsf{new}\ y_l : T'_l, \forall x_1 : T_1, \ldots, \forall x_m : T_m, M_1 \rightarrow M_2$.

The term M reduces into M' by the rewrite rule $\mathsf{new}\ y_1 : T'_1, \ldots, \mathsf{new}\ y_l : T'_l, \forall x_1 : T_1, \ldots, \forall x_m : T_m, M_1 \rightarrow M_2$ if and only if $M = C[\sigma M_1]$, $M' = C[\sigma M_2]$, where C is a term context and σ is a substitution that maps x_j to any term of type T_j for all $j \leq m$ and y_j to terms to the form $x[\widetilde{M}]$ where x is defined only by restrictions $\mathsf{new}\ x : T'_j$ for all $j \leq l$.

The prover has built-in rewrite rules for defining boolean functions:

$$\neg\text{true} \rightarrow \text{false} \qquad \neg\text{false} \rightarrow \text{true} \qquad \forall x : \mathit{bool}, \neg(\neg x) \rightarrow x$$
$$\forall x : T, \forall y : T, \neg(x = y) \rightarrow x \neq y$$
$$\forall x : T, \forall y : T, \neg(x \neq y) \rightarrow x = y$$
$$\forall x : T, x = x \rightarrow \text{true} \qquad \forall x : T, x \neq x \rightarrow \text{false}$$
$$\forall x : \mathit{bool}, \forall y : \mathit{bool}, \neg(x \wedge y) \rightarrow (\neg x) \vee (\neg y)$$
$$\forall x : \mathit{bool}, \forall y : \mathit{bool}, \neg(x \vee y) \rightarrow (\neg x) \wedge (\neg y)$$
$$\forall x : \mathit{bool}, x \wedge \text{true} \rightarrow x \qquad \forall x : \mathit{bool}, x \wedge \text{false} \rightarrow \text{false}$$
$$\forall x : \mathit{bool}, x \vee \text{true} \rightarrow \text{true} \qquad \forall x : \mathit{bool}, x \vee \text{false} \rightarrow x$$

The prover also has support for commutative function symbols, that is, binary function symbols $f : T \times T \rightarrow T'$ such that for all $x, y \in I_\eta(T)$, $I_\eta(f)(x, y) = I_\eta(f)(y, x)$. For such symbols, all equality and matching tests are performed modulo commutativity. The functions $\wedge$, $\vee$, $=$, and $\neq$ are commutative. So, for instance, the last four rewrite rules above may also be used to rewrite $\text{true} \wedge M$ into M, $\text{false} \wedge M$ into false, $\text{true} \vee M$ into true, and $\text{false} \vee M$ into M. Used-defined functions may also be declared commutative; xor is an example of such a commutative function.

C.2 Collecting True Facts from a Game

We use *facts* to represent properties that hold at certain program points in processes. We consider two kinds of facts: $\mathsf{defined}(M)$ means that M is defined, and a term M means that M is true (the boolean term M evaluates to true). In this section, we show how to compute a set of facts $\mathcal{F}_P$ that are guaranteed to hold at the program point P of the game.

The function collectFacts collects facts that hold at each program point of the game. More precisely, for each occurrence P of a subprocess of the game, it computes a set $\mathcal{F}_P$ of facts that hold at that occurrence. (It is important that P is an occurrence and not a process: processes at several occurrences may be equal and must be distinguished from one another here.) The function collectFacts also computes a set $\mathcal{D}$ containing pairs $(x[\widetilde{i}], P)$ where $x[\widetilde{i}]$ has been defined just above process P. (If there are several definitions of x, there is one such pair for each definition of x.) Finally, for output processes P, $\text{collectFacts}(P)$ returns a set of facts that will hold when the next output is executed and stores this set in $\mathcal{F}_P^{\text{Fut}}$. (The superscript Fut stands for *future*, since these facts do not hold yet at P, but will hold in the future.)

The function collectFacts is defined in Figure 6. It is initially called by $\text{collectFacts}(Q_0)$. It takes into account that $x[\widetilde{i}]$ may be defined by an input, a restriction, a let, or a find and updates $\mathcal{D}$ accordingly. Furthermore, when we execute $\mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ P'$, $x[\widetilde{i}] = M$ holds in P' and $x[\widetilde{i}]$ is defined in P'. When we execute $\mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P'$, M_j

collectFacts(Q) =

 if $Q = Q_1 \mid Q_2$ then collectFacts(Q_1); collectFacts(Q_2)

 if $Q = !^{i \le n} Q'$ then collectFacts(Q')

 if $Q = \mathsf{newChannel}\ c; Q'$ then collectFacts(Q')

 if $Q = c[M_1, \dots, M_l](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k); P$ then

  $\mathcal{F}_P = \{\mathsf{defined}(x_j[\widetilde{i}]) \mid j \le k\}; \mathcal{F}_P^{\mathrm{Fut}} = \mathrm{collectFacts}(P)$

  $\mathcal{D} = \mathcal{D} \cup \{(x_j[\widetilde{i}], P) \mid j \le k\}$

collectFacts(P) =

 if $P = \overline{c[M_1, \dots, M_l]}\langle N_1, \dots, N_k\rangle; Q$ then

  collectFacts(Q); return $\emptyset$

 if $P = \mathsf{new}\ x[\widetilde{i}] : T; P'$ then

  $\mathcal{F}_{P'} = \{\mathsf{defined}(x[\widetilde{i}])\}; \mathcal{F}_{P'}^{\mathrm{Fut}} = \mathrm{collectFacts}(P')$

  $\mathcal{D} = \mathcal{D} \cup \{(x[\widetilde{i}], P')\}$; return $\mathcal{F}_{P'} \cup \mathcal{F}_{P'}^{\mathrm{Fut}}$

 if $P = \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ P'$ then

  $\mathcal{F}_{P'} = \{\mathsf{defined}(x[\widetilde{i}]), x[\widetilde{i}] = M\}$

  $\mathcal{F}_{P'}^{\mathrm{Fut}} = \mathrm{collectFacts}(P')$

  $\mathcal{D} = \mathcal{D} \cup \{(x[\widetilde{i}], P')\}$; return $\mathcal{F}_{P'} \cup \mathcal{F}_{P'}^{\mathrm{Fut}}$

 if $P = \mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \le n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \le n_{jm_j}$ suchthat $\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P' then

  for each $j \le m$,

   $\mathcal{F}_{P_j} = \{\mathsf{defined}(u_{j1}[\widetilde{i'}]), \dots, \mathsf{defined}(u_{jm_j}[\widetilde{i'}]),$
    $\mathsf{defined}(M_{j1}), \dots, \mathsf{defined}(M_{jl_j}), M_j\}$

   $\mathcal{F}_{P_j}^{\mathrm{Fut}} = \mathrm{collectFacts}(P_j)$;

   $\mathcal{D} = \mathcal{D} \cup \{(u_{j1}[\widetilde{i'}], P_j), \dots, (u_{jm_j}[\widetilde{i'}], P_j)\}$

  $\mathcal{F}_{P'} = \{\neg M_j \mid m_j = l_j = 0\}; \mathcal{F}_{P'}^{\mathrm{Fut}} = \mathrm{collectFacts}(P')$

  return $(\mathcal{F}_{P'} \cup \mathcal{F}_{P'}^{\mathrm{Fut}}) \cap \bigcap_{j=1}^{m} (\mathcal{F}_{P_j} \cup \mathcal{F}_{P_j}^{\mathrm{Fut}})$

Figure 6: The function collectFacts

holds in P_j, $M_{j1}, \dots, M_{jl_j}, u_{j1}[\widetilde{i}], \dots, u_{jm_j}[\widetilde{i}]$ are defined in P_j, and $\neg M_j$ holds in P' when $m_j = l_j = 0$.

After calling collectFacts(Q_0), we complete the computed sets $\mathcal{F}_P$ (where P may be an input or output process) by adding facts that come from processes above P:

$$\mathcal{F}_P \leftarrow \mathcal{F}_P \cup \mathcal{F}_{P'} \text{ if } P \text{ is immediately under } P'$$

We also add facts that we can deduce from facts defined(M). Precisely, if defined$(M) \in \mathcal{F}_P$ and $x[M_1, \dots, M_m]$ is a subterm of M, then we take into account facts that are known to be true at the definitions of x by adding them to $\mathcal{F}_P$ as follows:

$$\mathcal{F}_P \leftarrow \mathcal{F}_P \cup \left(\bigcap_{(x[i_1,\dots,i_m],P') \in \mathcal{D}} \begin{cases} \sigma(\mathcal{F}_{P'} \cup (\mathcal{F}_{P'}^{\mathrm{Fut}} \cap \mathcal{F}_P)) & \text{if } P \text{ is under } P' \\ \sigma(\mathcal{F}_{P'} \cup \mathcal{F}_{P'}^{\mathrm{Fut}}) & \text{otherwise} \end{cases} \right)$$

where $\sigma = \{M_1/i_1, \dots, M_m/i_m\}$. Indeed, if defined$(M) \in \mathcal{F}_P$ and $x[M_1, \dots, M_m]$ is a subterm of M, then $x[M_1, \dots, M_m]$ is defined at P, so some definition of $x[M_1, \dots, M_m]$, just above the process P', must have been executed before reaching P, so the facts that hold at P' also hold at P, with a suitable substitution of indices: we have $\sigma\mathcal{F}_{P'}$, that is, $\mathcal{F}_{P'}\{M_1/i_1, \dots, M_m/i_m\}$. Moreover, if the occurrence P is not syntactically under the occurrence P', then the code of P' must have been executed until the next output before yielding control to some other code and reaching P, so in fact $\sigma(\mathcal{F}_{P'} \cup \mathcal{F}_{P'}^{\mathrm{Fut}})$ hold. If P is syntactically under P', it is possible that the code of P' has been executed until reaching P instead of until reaching the next output, so we have only $\sigma(\mathcal{F}_{P'} \cup (\mathcal{F}_{P'}^{\mathrm{Fut}} \cap \mathcal{F}_P))$. If there are several definitions of x, we do not know which one has been executed, so we only add to $\mathcal{F}_P$ the facts that hold in all cases, by taking the intersection on all definitions of x.

This operation may add new defined facts to $\mathcal{F}_P$, so it is executed until a fixpoint is reached, except that, in order to avoid infinite loops, we do not execute this step for definitions defined(M) in which M contains nested occurrences of the same symbol (such as $x[\dots x[\dots] \dots]$).

We also consider an additional fact that serves in expressing that the condition part of a find failed. Precisely, the fact *elsefind*$((u_1 \le n_1, \dots, u_m \le n_m), (M_1, \dots, M_l), M)$ means that for all $u_1 \in [1, n_1], \dots, u_m \in [1, n_m]$, the terms $M_1, \dots, M_l$ are not all defined or M is false. The function collectElseFind described in Figure 7 collects *elsefind* facts that hold at each occurrence. The function collectElseFind$(P, \mathcal{F})$ is called when $\mathcal{F}$ is the set of true *elsefind* facts at occurrence P. It sets the value of $\mathcal{F}_P^{\mathrm{ElseFind}}$ to $\mathcal{F}$.

- In the case of restrictions, assignments, and then branches of find, it takes into account that a variable x or $u_{j1}, \dots, u_{jm_j}$ is newly defined. Hence *elsefind* facts that claim that one of these variables is not defined are removed.

- In the case of the else branch of a find, it adds the new *elsefind* facts that hold when the conditions of the find fail. These conditions express that each then branch of the find fails by a *elsefind* fact. To construct this fact, we replace (by applying σ_j) the terms $u_{j1}[\widetilde{i}], \dots, u_{jm_j}[\widetilde{i}]$ with fresh variables $u_1, \dots, u_{m_j}$, respectively.

$\mathrm{collectElseFind}(Q) =$
  if $Q = Q_1 \mid Q_2$ then
    $\mathrm{collectElseFind}(Q_1); \mathrm{collectElseFind}(Q_2)$
  if $Q = !^{i \leq n} Q'$ then $\mathrm{collectElseFind}(Q')$
  if $Q = \mathsf{newChannel}\ c; Q'$ then $\mathrm{collectElseFind}(Q')$
  if $Q = c[M_1, \dots, M_l](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k); P$ then
    $\mathrm{collectElseFind}(P, \emptyset)$

$\mathrm{collectElseFind}(P, \mathcal{F}) =$
  $\mathcal{F}_P^{\mathrm{ElseFind}} = \mathcal{F}$
  if $P = \overline{c[M_1, \dots, M_l]}\langle N_1, \dots, N_k \rangle; Q$ then
    $\mathrm{collectElseFind}(Q)$
  if $P = \mathsf{new}\ x[\widetilde{i}] : T; P'$
  or $P = \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ P'$ then
    $\mathcal{F}' = \{ \mathit{elsefind}((\widetilde{u} \leq \widetilde{n}), (M_1, \dots, M_l), M) \in \mathcal{F} \mid$
        x does not occur in $M_1, \dots, M_l \}$
    $\mathrm{collectElseFind}(P', \mathcal{F}')$
  if $P = \mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$
  $\mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P'$
  then
    for each $j \leq m$,
      $\mathcal{F}'_j = \{ \mathit{elsefind}((\widetilde{u} \leq \widetilde{n}), (M_1, \dots, M_l), M) \in \mathcal{F}$
          $\mid u_{j1}, \dots, u_{jm_j}$ do not occur in $M_1, \dots, M_l \}$
      $\mathrm{collectElseFind}(P_j, \mathcal{F}'_j)$
    $\sigma_j = \{ u_1 / u_{j1}[\widetilde{i}], \dots, u_{m_j} / u_{jm_j}[\widetilde{i}] \}$
    $\mathrm{collectElseFind}(P', \mathcal{F} \cup$
      $\{ \mathit{elsefind}((u_1 \leq n_{j1}, \dots, u_{m_j} \leq n_{jm_j}),$
      $\sigma_j(M_{j1}, \dots, M_{jl_j}), \sigma_j M_j) \mid j \in \{1, \dots, m\} \})$

Figure 7: The function collectElseFind

- In the case of an output, any code may be executed before the input processes under it, so any variable may be defined by that code, and all *elsefind* facts are removed. That is why the function collectElseFind for input processes has no $\mathcal{F}$ argument (this argument would always be empty) and calls $\mathrm{collectElseFind}(P, \emptyset)$ for processes P that follow an input.

The *elsefind* facts can be used to add new facts to the facts $\mathcal{F}_P$. Indeed, if $\mathcal{F}_P$ implies that $M_1, \dots, M_l$ are defined for some values of $u_1, \dots, u_m$, then the fact $\mathit{elsefind}((u_1 \leq n_1, \dots, u_m \leq n_m), (M_1, \dots, M_l), M)$ implies that M is false for these values of $u_1, \dots, u_m$. Precisely, we execute:

$$\mathcal{F}_P \leftarrow \mathcal{F}_P \cup \{ \neg \sigma M \mid \mathit{elsefind}((u_1 \leq n_1, \dots, u_m \leq n_m), (M_1, \dots, M_l), M) \in \mathcal{F}_P^{\mathrm{ElseFind}}, \mathrm{Dom}(\sigma) = \{u_1, \dots, u_m\}, \text{for each } j \in \{1, \dots, l\}, \sigma M_j \text{ is a subterm of } M'_j \text{ and } \mathsf{defined}(M'_j) \in \mathcal{F}_P \}$$

The possible images of σ are found by exploring the set of defined facts in $\mathcal{F}_P$.

Furthermore, when the previous update of $\mathcal{F}_P$ adds facts, we again complete the computed sets $\mathcal{F}_P$ by adding facts that come from processes above P:

$$\mathcal{F}_P \leftarrow \mathcal{F}_P \cup \mathcal{F}_{P'} \text{ if } P \text{ is immediately under } P'$$

We could also iterate the addition of consequences of defined facts. (However, for simplicity, the current implementation does not perform such an iteration.)

C.3 Global Dependency Analysis

For each variable x, the global dependency analysis tries to find a set of variables S such that only variables in S depend on x. In particular, when the global dependency analysis succeeds, the control flow and the view of the adversary do not depend on x, except in cases of negligible probability.

Let x be a variable defined only by restrictions $\mathsf{new}\ x : T$ where T is a large type. Let S_{def} be a set of variables defined only by assignments. Let S_{dep} be a set of variables containing x. (Intuitively, S_{dep} will be a superset of variables that depend on x.)

We say that a function $f : T \to T'$ is *uniform* when each element of $I_\eta(T')$ has at most $|I_\eta(T)|/|I_\eta(T')|$ antecedents by f. In particular, this is true in the following two cases:

- f is such that $f(x)$ is uniformly distributed in $I_\eta(T')$ if x is uniformly distributed in $I_\eta(T)$.
- f is the restriction to the image of f' of an inverse of f', where f' is a poly-injective function. (We consider that $f(x)$ is undefined when x is not in the image of f'. Here, in contrast to the rest of the paper, we allow $f : T \to T'$ to be defined only on a subset of $I_\eta(T)$.) Precisely, when $x_k \in S_{\mathrm{def}}$ is defined by a pattern-matching $\mathsf{let}\ f'(x_1, \dots, x_n) = M\ \mathsf{in}\ P\ \mathsf{else}\ P'$, we have $x_k = {f'}_k^{-1}(M)$, but furthermore when x_k is defined we know that the value of M is in the image of f', so we have $x_k = f(M)$ where $f = {f'}_k^{-1}{}_{|\mathrm{im}\ f'}$.

We say that M *characterizes a part of* x with S_{def},S_{dep} when for all M_0 obtained from M by substituting variables of S_{def} with their definition (when there is a dependency cycle among variables of S_{def}, we do not substitute a variable inside its definition), $\alpha M_0 = M_0$ implies $f_1(\dots f_k((\alpha x)[\widetilde{M'}])) = f_1(\dots f_k(x[\widetilde{M}]))$ for some uniform functions $f_1, \dots, f_k$ and for some $\widetilde{M}$ and $\widetilde{M'}$, where α is a renaming of variables of S_{dep} to fresh variables, $x[\widetilde{M}]$ is a subterm of M_0, $(\alpha x)[\widetilde{M'}]$ is a subterm of αM_0, the variables in S_{dep} do not occur in $\widetilde{M}$ or $\widetilde{M'}$, T is the type of the result of f_1 (or of x when $k = 0$), and T is a large type. In that case, the value of M uniquely determines the value of $f_1(\dots f_k(x[\widetilde{M}]))$.

We use a simple rewriting prover to determine that. We consider the set of terms $\mathcal{M}_0 = \{\alpha M_0 = M_0\}$, and we rewrite elements of $\mathcal{M}_0$ using the first kind of user-defined rewrite rules mentioned in the first point of this section and the rule $\{M_1 \wedge M_2\} \cup \mathcal{M}' \to \{M_1, M_2\} \cup \mathcal{M}'$.

When $\mathcal{M}_0$ can be rewritten to a set that contains an equality of the form $f_1(\dots f_k(x[\widetilde{M}])) = f_1(\dots f_k((\alpha x)[\widetilde{M'}]))$ or $f_1(\dots f_k((\alpha x)[\widetilde{M'}])) = f_1(\dots f_k(x[\widetilde{M}]))$ for some $\widetilde{M}$ and $\widetilde{M'}$ such that the variables in S_{dep} do not occur in $\widetilde{M}$ or $\widetilde{M'}$, we have that M characterizes a part of x with $S_{\mathrm{def}}, S_{\mathrm{dep}}$.

We say that M *characterizes a part of* x when M characterizes a part of x with $\emptyset, S'$ where S' is $\{x\}$ union the set of all variables except those defined by restrictions. (We know that variables different from x and defined by restrictions do not depend on x, so in the absence of more precise information, we can set $S_{\mathrm{dep}} = S'$.)

We say that $\mathrm{only_dep}(x) = S$ when intuitively, only variables in S depend on x, and the adversary cannot see the value of x. Formally, $\mathrm{only_dep}(x) = S$ when

- $S \cap V = \emptyset$.
- Variables of S do not occur in input or output channels or messages, that is, they do not occur in the terms $M_1, \dots, M_m, N_1, \dots, N_k$ in the input $c[M_1, \dots, M_m](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k)$ or in the output $\overline{c[M_1, \dots, M_m]}\langle N_1, \dots, N_k\rangle$.
- Variables of S except x are defined only by assignments.
- If a variable $y \in S$ occurs in M in let $z : T = M$ in P, then $z \in S$.
- Variables in S may occur in defined conditions of find but only at the root of them.
- All terms M_j in processes find $(\bigoplus_{j=1}^{m} \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$ suchthat defined$(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P' are combinations by $\wedge$, $\vee$, or $\neg$ of terms that either do not contain variables in S or are of the form $M_1 = M_2$ or $M_1 \neq M_2$ where M_1 characterizes a part of x with $S \setminus \{x\}, S$ and no variable of S occurs in M_2, or M_2 characterizes a part of x with $S \setminus \{x\}, S$ and no variable of S occurs in M_1.

The last item implies that the result of tests does not depend on the values of variables in S, except in cases of negligible probability. Indeed, the tests $M_1 = M_2$ with M_1 characterizes a part of x with $S \setminus \{x\}, S$ and M_2 does not depend on variables in S are false except in cases of negligible probability, since the value of M_1 uniquely determines the value of $f_1(\dots f_k(x[\widetilde{M}]))$ and M_2 does not depend on $f_1(\dots f_k(x[\widetilde{M}]))$, so the equality $M_1 = M_2$ happens for a single value of $f_1(\dots f_k(x[\widetilde{M}]))$, which yields a negligible probability because $f_1, \dots, f_k$ are uniform, x is chosen with uniform probability, and the type of the result of f_1 is large. Similarly, the tests $M_1 \neq M_2$ are true except in cases of negligible probability.

In checking the conditions of $\mathrm{only_dep}(x) = S$, we do not consider the parts of the code that are unreachable due to tests whose result is known by the conditions above.

The set S is computed by a fixpoint iteration, starting from $\{x\}$ and adding variables defined by assignments that depend on variables already in S.

C.4 Local Dependency Analysis

For each program point P and each variable x, the local dependency analysis tries to find which variables and terms depend on $x[\widetilde{i}]$ at program point P, where $\widetilde{i}$ denotes the current replication indices at the definition of x. It simplifies the game on-the-fly when possible.

For each occurrence of a process P and each variable x such that a restriction new $x : T$ occurs above P and T is a large type, we compute a set of terms $\mathrm{indep}_P(x)$ that are independent of $x[\widetilde{i}]$ where $\widetilde{i}$ denotes the current replication indices at the definition of x.

For each occurrence of a process P and each variable x such that a restriction new $x : T$ occurs above P and T is a large type, we also compute $\mathrm{depend}_P(x)$ which can be either $\top$ (I don't know) or a set of pairs (y, M) where $y[\widetilde{i}]$ depends on $x[\widetilde{i}]$ by assignments, and M is a term defining $y[\widetilde{i}]$ as a function of $x[\widetilde{i}]$. (The tuple $\widetilde{i}$ denotes the current replication indices at the definition of x and of y.)

We define "M *characterizes a part of* $x[\widetilde{i}]$ at P" as follows. Let α be defined by $\alpha(f(M_1, \dots, M_m)) = f(\alpha M_1, \dots, \alpha M_m)$; $\alpha(i) = i$ where i is a replication index; $\alpha(M') = M'$ when $M' \in \mathrm{indep}_P(x)$; $\alpha(y[M_1, \dots, M_{m'}]) = y[\alpha M_1, \dots, \alpha M_{m'}]$ when $y \neq x$ and y either is defined only by restrictions or $\mathrm{depend}_P(x) \neq \top$ and $(y, M') \notin \mathrm{depend}_P(x)$ for any M'; $\alpha(y[M_1, \dots, M_{m'}]) = y'[\alpha M_1, \dots, \alpha M_{m'}]$ where y' is a fresh variable, otherwise. We write $y' = \alpha y$ in this case. We say that M *characterizes a part of* $x[\widetilde{i}]$ at P when $\alpha M = M$ implies $f_1(\dots f_k((\alpha x)[\widetilde{i}])) = f_1(\dots f_k(x[\widetilde{i}]))$ for some uniform functions $f_1, \dots, f_k$, where $x[\widetilde{i}]$ is a subterm of M, $(\alpha x)[\widetilde{i}]$ is a subterm of αM, T' is the type of the result of f_1 (or of x when $k = 0$), and T' is a large type. In that case, the value of M uniquely determines the value of $f_1(\dots f_k(x[\widetilde{i}]))$. This property is shown by a simple rewriting prover, as in the global dependency analysis.

We denote by $\mathrm{subterms}(M)$ the set of subterms of the term M.

We say that M does not depend on x at P when M is built by function applications from terms in $\mathrm{indep}_P(x)$, replications indices, and terms $y[M_1, \dots, M_m]$ such that $M_1, \dots, M_m$ do not depend on x at P, $y \neq x$, and either y is defined only by restrictions or $\mathrm{depend}_P(x) \neq \top$ and $y \neq y'$ for all

$\mathrm{depAnal}(Q, \mathrm{indep}) =$
$\quad \forall y, \mathrm{depend}_Q(y) = \top; \mathrm{indep}_Q = \mathrm{indep}$
$\quad$ if $Q = Q_1 \mid Q_2$ then
$\quad\quad \mathrm{depAnal}(Q_1, \mathrm{indep}); \mathrm{depAnal}(Q_2, \mathrm{indep})$
$\quad$ if $Q = !^{i \le n} Q'$ then $\mathrm{depAnal}(Q', \mathrm{indep})$
$\quad$ if $Q = \mathsf{newChannel}\ c; Q'$ then $\mathrm{depAnal}(Q', \mathrm{indep})$
$\quad$ if $Q = c[M_1, \dots, M_l](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k); P$ then
$\quad\quad \mathrm{depAnal}(P, \{\forall y, y \mapsto \top\}, \mathrm{indep})$

Figure 8: Local dependency analysis (1)

$(y', M') \in \mathrm{depend}_P(x)$. Since terms in $\mathrm{indep}_P(x)$ do not depend on $x[\widetilde{i}]$ and when $\mathrm{depend}_P(x) \neq \top$, variables not in the first component of $\mathrm{depend}_P(x)$ do not depend on $x[\widetilde{i}]$, the conditions above guarantee that M does not depend on $x[\widetilde{i}]$, where $\widetilde{i}$ are the current replication indices at the definition of x.

When $\mathrm{depend} \neq \top$, we denote by $M\mathrm{depend}$ the term obtained from M by replacing $y[\widetilde{i}]$ with M' for each $(y, M') \in \mathrm{depend}$, where $\widetilde{i}$ denotes the replication indices at the definition of y.

We define simplifyTerm such that $\mathrm{simplifyTerm}(M, P)$ is a simplified version of M, equal to M except in cases of negligible probability. The term $\mathrm{simplifyTerm}(M, P)$ is defined as follows:

- Case 1: M is $M_1 = M_2$. For each x, we proceed as follows. If $\mathrm{depend}_P(x) = \top$, let $M_0 = M_1$; otherwise, let $M_0 = M_1\mathrm{depend}_P(x)$. Let M_0' and M_2' be obtained respectively from M_0 and M_2 by replacing all array indices that depend on x at P with fresh replication indices. If M_0' characterizes a part of $x[\widetilde{i}]$ at P, and M_2' does not depend on x at P, then $\mathrm{simplifyTerm}(M, P) = \mathrm{false}$. Indeed, M is equal to false up to negligible probability in this case. We have similar cases swapping M_1 and M_2 or when M is $M_1 \neq M_2$. (In the latter case, $\mathrm{simplifyTerm}(M, P) = \mathrm{true}$.)
- Case 2: M is $M_1 \wedge M_2$. Let $M_1' = \mathrm{simplifyTerm}(M_1, P)$ and $M_2' = \mathrm{simplifyTerm}(M_2, P)$. If M_1' or M_2' are false, we return false. If M_1' is true, we return M_2'. If M_2' is true, we return M_1'. Otherwise, we return $M_1' \wedge M_2'$. We have similar cases when M is $M_1 \vee M_2$ or $\neg M_1$.
- In all other cases, $\mathrm{simplifyTerm}(M, P) = M$.

The local dependency analysis is defined in Figures 8 and 9. The function depAnal is initially called with $\mathrm{depAnal}(Q_0, \emptyset)$ where $\emptyset$ designates the function defined nowhere.

- For input processes, depAnal sets $\mathrm{depend}_Q(y)$ to $\top$, so that depend_Q gives no information, and propagates indep. Indeed, when $y[\widetilde{i'}]$ is set in some output process P_0, the value of $y[\widetilde{i'}]$ may be output by P_0 or read by find in other output processes executed after P_0, so as soon as P_0 passes control to another process by the first output after the definition of y, we lose track of exactly which variables depend on $y[\widetilde{i'}]$. However, variables already defined before

$\mathrm{depAnal}(P, \mathrm{depend}, \mathrm{indep}) =$
$\quad \mathrm{depend}_P = \mathrm{depend}; \mathrm{indep}_P = \mathrm{indep}$
$\quad$ if $P = \overline{c[M_1, \dots, M_l]}\langle N_1, \dots, N_k\rangle; Q$ then
$\quad\quad \mathrm{depAnal}(Q, \mathrm{indep})$
$\quad$ if $P = \mathsf{new}\ x[\widetilde{i}] : T; P'$ then
$\quad\quad$ if T is a large type then
$\quad\quad\quad \mathrm{depend}'(x) = \emptyset$
$\quad\quad\quad \mathrm{indep}'(x) = \bigcup_{\mathsf{defined}(M) \in \mathcal{F}_P} \mathrm{subterms}(M)$
$\quad\quad \forall y \neq x, \mathrm{depend}'(y) = \mathrm{depend}(y),$
$\quad\quad\quad \mathrm{indep}'(y) = \mathrm{indep}(y) \cup \{x[\widetilde{i}]\}$
$\quad\quad \mathrm{depAnal}(P', \mathrm{depend}', \mathrm{indep}')$
$\quad$ if $P = \mathsf{let}\ x[\widetilde{i}] : T = M\ \mathsf{in}\ P'$ then
$\quad\quad \forall y,$ if M does not depend on y at P then
$\quad\quad\quad\quad \mathrm{depend}'(y) = \mathrm{depend}(y)$
$\quad\quad\quad\quad \mathrm{indep}'(y) = \{x[\widetilde{i}]\} \cup \mathrm{indep}(y)$
$\quad\quad\quad$ else
$\quad\quad\quad\quad$ if $\mathrm{depend}(y) \neq \top$ then
$\quad\quad\quad\quad\quad \mathrm{depend}'(y) = \mathrm{depend}(y) \cup \{(x, M\mathrm{depend}(y))\}$
$\quad\quad\quad\quad$ else
$\quad\quad\quad\quad\quad \mathrm{depend}'(y) = \top$
$\quad\quad\quad\quad \mathrm{indep}'(y) = \mathrm{indep}(y)$
$\quad\quad \mathrm{depAnal}(P', \mathrm{depend}', \mathrm{indep}')$
$\quad$ if $P = \mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \le n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \le n_{jm_j}$
$\quad \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P'$
$\quad$ then
$\quad\quad$ for each $j \le m, M_j' = \mathrm{simplifyTerm}(M_j, P)$
$\quad\quad\quad$ replace M_j with M_j'
$\quad\quad\quad$ if $M_j' = \mathrm{false}$ then remove the j-th branch
$\quad\quad\quad$ if $M_j' = \mathrm{true}$ and $l_j = 0$ then replace P' with $\overline{yield}\langle\rangle$
$\quad\quad$ if $m = 0$ then
$\quad\quad\quad$ replace P with P'; $\mathrm{depAnal}(P', \mathrm{depend}, \mathrm{indep})$
$\quad\quad$ else if $m = 1, m_1 = l_1 = 0$, and $M_1 = \mathrm{true}$ then
$\quad\quad\quad$ replace P with P_1; $\mathrm{depAnal}(P_1, \mathrm{depend}, \mathrm{indep})$
$\quad\quad$ else
$\quad\quad \forall y,$ if $\forall j, k$, M_{jk} and M_j' do not depend on y at P then
$\quad\quad\quad\quad \mathrm{depend}'(y) = \mathrm{depend}(y)$
$\quad\quad\quad\quad$ for each $j \le m, \mathrm{indep}_j(y) = \mathrm{indep}(y) \cup \{M' \mid$
$\quad\quad\quad\quad\quad M' \in \mathrm{subterms}(M)$ for some $\mathsf{defined}(M) \in \mathcal{F}_{P_j}$,
$\quad\quad\quad\quad\quad M'$ does not depend on y at $P\}$
$\quad\quad\quad$ else
$\quad\quad\quad\quad \mathrm{depend}'(y) = \top$
$\quad\quad\quad\quad$ for each $j \le m, \mathrm{indep}_j(y) = \mathrm{indep}(y)$
$\quad\quad$ for each $j \le m, \mathrm{depAnal}(P_j, \mathrm{depend}', \mathrm{indep}_j)$
$\quad\quad \mathrm{depAnal}(P', \mathrm{depend}', \mathrm{indep})$

Figure 9: Local dependency analysis (2)

P_0 passes control to another process and proved to be independent of $y[\widetilde{i'}]$ remain independent of $y[\widetilde{i'}]$, so we can propagate indep in all subprocesses of P_0.

- In the case of an output, $\mathrm{depAnal}$ forgets the information in depend_P as mentioned above.

- In the case of a restriction new $x[\widetilde{i}] : T$, if T is a large type, we create the dependency information for the newly defined variable x: no variable depends on $x[\widetilde{i}]$, and all terms already defined before the restriction are independent of $x[\widetilde{i}]$. We also note that $x[\widetilde{i}]$ is independent of $y[\widetilde{i'}]$ for other variables y by adding $x[\widetilde{i}]$ to $\mathrm{indep}(y)$.

- In the case of an assignment let $x[\widetilde{i}] : T = M$, if M depends on $y[\widetilde{i'}]$ for some variable y, then $x[\widetilde{i}]$ depends on $y[\widetilde{i'}]$, so x is added to $\mathrm{depend}(y)$ (if it is not $\top$); otherwise, $x[\widetilde{i}]$ does not depend on $y[\widetilde{i'}]$ so it is added to $\mathrm{indep}(y)$.

- In the case of a find, we first simplify each condition of the find, remove branches when we can prove that they are taken with negligible probability, and remove the find itself when we know which branch is taken and this branch of the find does not define variables. Furthermore, if some condition of find depends on $y[\widetilde{i}]$ for some variable y, $\mathrm{depend}'(y)$ is set to $\top$: the control flow depends on $y[\widetilde{i}]$ so future assignments in fact depend on $y[\widetilde{i}]$ even if the assigned expression itself does not, so we can no longer keep track precisely of which variables depend on $y[\widetilde{i}]$. Otherwise, we add all terms that are guaranteed to be defined and independent of $y[\widetilde{i}]$ to $\mathrm{indep}(y)$.

C.5 Equational Prover

We use an algorithm inspired by the Knuth-Bendix completion algorithm [29], with differences detailed below.

The prover manipulates pairs $\mathcal{F}, \mathcal{R}$ where $\mathcal{F}$ is a set of facts (M or defined(M)) and $\mathcal{R}$ is a set of rewrite rules $M_1 \rightarrow M_2$. We say that M reduces into M' by $M_1 \rightarrow M_2$ when $M = C[M_1]$ and $M' = C[M_2]$ for some term context C. (That is, all variables in rewrite rules of $\mathcal{R}$ are considered as constants.) The prover starts with a certain set of facts $\mathcal{F}$ and $\mathcal{R} = \emptyset$. Then the prover transforms the pairs $(\mathcal{F}, \mathcal{R})$ by the following rules (the rule $\frac{\mathcal{F},\mathcal{R}}{\mathcal{F}',\mathcal{R}'}$ means that $\mathcal{F}, \mathcal{R}$ is transformed into $\mathcal{F}', \mathcal{R}'$):

$$\frac{\mathcal{F} \cup \{F\}, \mathcal{R}}{\mathcal{F} \cup \{F'\}, \mathcal{R}} \quad \text{if } F \text{ reduces into } F' \text{ by a rule of } \mathcal{R} \text{ or a user-defined rewrite rule} \tag{1}$$

$$\frac{\mathcal{F} \cup \{M_1 \wedge M_2\}, \mathcal{R}}{\mathcal{F} \cup \{M_1, M_2\}, \mathcal{R}} \tag{2}$$

$$\frac{\mathcal{F} \cup \{x[M_1, \ldots, M_m] = x[M'_1, \ldots, M'_m]\}, \mathcal{R}}{\mathcal{F} \cup \{M_1 = M'_1, \ldots, M_m = M'_m\}, \mathcal{R}} \quad \text{when } x \text{ is defined only by restrictions new } x : T \text{ and } T \text{ is a large type} \tag{3}$$

$$\frac{\mathcal{F} \cup \{M_1 = M_2\}, \mathcal{R}}{\{\mathrm{false}\}, \mathcal{R}} \quad \text{when one of the following conditions holds:}$$

- denoting by M'_1 the term obtained from M_1 by replacing all array indices that are not replication indices with fresh replication indices, we have the following properties: x occurs in M'_1, x is defined only by restrictions new $x : T$, T is a large type, M'_1 characterizes a part of x, and M_2 is obtained by optionally applying function symbols to terms of the form $y[\widetilde{M}]$ where y is defined only by restrictions and $y \neq x$;

- x occurs in M_1, x is defined only by restrictions new $x : T$, T is a large type, M_1 characterizes a part of x, $\mathrm{only_dep}(x) = S$, and no variable of S occurs in M_2;

- $\mathrm{simplifyTerm}(M_1 = M_2, P) = \mathrm{false}$, where P is the current program point.

(4)

$$\frac{\mathcal{F} \cup \{M = M'\}, \mathcal{R}}{\mathcal{F}, \mathcal{R} \cup \{M \rightarrow M'\}} \text{ if } M > M' \tag{5}$$

$$\frac{\mathcal{F}, \mathcal{R} \cup \{M_1 \rightarrow M_2\}}{\mathcal{F} \cup \{M_1 = M'_2\}, \mathcal{R}} \quad \text{if } M_2 \text{ reduces into } M'_2 \text{ by a rule of } \mathcal{R} \text{ or a user-defined rewrite rule} \tag{6}$$

$$\frac{\mathcal{F}, \mathcal{R} \cup \{M_1 \rightarrow M_2\}}{\mathcal{F} \cup \{M'_1 = M_2\}, \mathcal{R}} \quad \text{if } M_1 \text{ reduces into } M'_1 \text{ by a rule of } \mathcal{R} \tag{7}$$

We also use the symmetrics of Rules (4) and (5) obtained by swapping the two sides of the equality.

Rule (1) simplifies facts using rewrite rules. Rule (2) decomposes conjunctions of facts. Rules (3) and (4) exploit the elimination of collisions between random values. Rule (3) takes into account that, when x is defined by a restriction of a large type, two different cells of x have a negligible probability of containing the same value. So when two cells of x contain the same value, we can conclude up to negligible probability that they are the same cell. Rule (4) expresses that M_1 and M_2 have a negligible probability of being equal when x is defined by a restriction of a large type, M_1 characterizes a part of x, and M_2 does not depend of x. The first item of (4) establishes these properties without further dependency analysis; the second item exploits the global dependency analysis; and the third item exploits the local dependency analysis.

Rule (5) is applied only when Rules (1) to (4) cannot be applied. Rule (5) transforms equations into rewrite rules by orienting them. We say that $M > M'$ when either M is the form $x[\widetilde{M}]$, x does not occur in M', and x is not defined only by restrictions, or $M = x[M_1, \ldots, M_m]$, $M' = x[M'_1, \ldots, M'_m]$, and for all $j \leq m$, $M_j > M'_j$. Intuitively, our goal is to replace M with M' when M' defines the content of the variable M. (Notice that this is not an ordering; the Knuth-Bendix algorithm normally uses a reduction ordering to orient equations. However, we tried some reduction orderings, namely the lexicographic path ordering and the Knuth-Bendix ordering, and obtained disappointing results: the prover fails to prove many equalities because too many equations are left unoriented. The simple heuristic given above succeeds more often, at the expense of a greater risk of non-termination, but that does not cause problems in practice on our examples. We believe that this

comes from the particular structure of equations, which come from let definitions and from conditions of find or if, and tend to define variables from other variables without creating dependency cycles.)

Rules (6) and (7) are systematically applied to simplify all rewrite rules of $\mathcal{R}$ after a new rewrite rule has been added by Rule (5). Since all terms in rewrite rules of $\mathcal{R}$ are considered as constants, Rule (7) in fact includes the deduction of equations from critical pairs done by the standard Knuth-Bendix completion algorithm.

We say that $\mathcal{F}$ *yields a contradiction* when the prover, starting from $(\mathcal{F}, \emptyset)$, derives false.

C.6 Game Simplification

We use the following transformations in order to simplify games. These transformations exploit the information collected as explained in the previous sections.

- Each term M in the game is replaced with a simplified term M' obtained by reducing M by user-defined rewrite rules (first point of this section) and the rewrite rules obtained from $\mathcal{F}_{P_M}$ by the above equational prover where P_M is the smallest process containing M. The replacement is performed only when at least one user-defined rewrite rule has been used, to avoid complicating the game by substituting all variables with their value.
- If $P = \mathsf{find}\ (\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ $\mathsf{suchthat\ defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ $\mathsf{then}\ P_j)$ $\mathsf{else}\ P'$, $u_{jk}[\widetilde{i}]$ reduces into M' by user-defined rewrite rules (first point of this section) and the rewrite rules obtained from $\mathcal{F}_{P_j}$, and u_{jk} does not occur in M', then u_{jk} is removed from the j-th branch of this find, $u_{jk}[\widetilde{i}]$ is replaced with M' in $M_{j1}, \dots, M_{jl_j}, M_j$ and P_j is replaced with $\mathsf{let}\ u_{jk}[\widetilde{i}] : [1, n_{jk}] = M'\ \mathsf{in}\ P_j$. (Intuitively, $u_{jk}[\widetilde{i}] = M'$, so the value of $u_{jk}[\widetilde{i}]$ can be computed by evaluating M' instead of performing an array lookup. We remove $u_{jk}[\widetilde{i}]$ from the variables looked up by find and replace $u_{jk}[\widetilde{i}]$ with its value M'.)
- Suppose that $P = \mathsf{find}\ (\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ $\mathsf{suchthat\ defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ $\mathsf{then}\ P_j)$ $\mathsf{else}\ P'$, $x[N_1, \dots, N_l]$ is a subterm of M_{jk}, and none of the following conditions holds: a) P is under a definition of x in Q_0; b) Q_0 contains $Q_1 \mid Q_2$ such that a definition of x occurs in Q_1 and P is under Q_2 or a definition of x occurs in Q_2 and P is under Q_1; c) Q_0 contains $lp + 1$ replications above a process Q that contains a definition of x and P, where lp is the length of the longest common prefix between $N_1, \dots, N_l$ and the current replication indices at the definitions of x. Then the j-th branch of the find is removed. (In this case, $x[N_1, \dots, N_l]$ cannot be defined at P, so the j-th branch of the find cannot be taken.)
- If $P = \mathsf{find}\ (\bigoplus_{j=1}^m \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$ $\mathsf{suchthat\ defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ $\mathsf{then}\ P_j)$ $\mathsf{else}\ P'$ and $\mathcal{F}_{P_j}$ yields a contradiction, then the j-th branch of the find is removed.
- If $P = \mathsf{find\ else}\ P'$, then P is replaced with P'.
- If $\mathsf{find}\ (\bigoplus_{j=1}^m \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$ $\mathsf{suchthat\ defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ $\mathsf{then}\ P_j)$ $\mathsf{else}\ P'$ and $\mathcal{F}_{P'}$ yields a contradiction, then P' is replaced with $\overline{yield}\langle\rangle$.
- If $P = \mathsf{find}\ \widetilde{u}[\widetilde{i}] \leq \widetilde{n}$ $\mathsf{suchthat}\ M$ $\mathsf{then}\ P_1$ $\mathsf{else}\ P'$, $\mathcal{F}_{P'}$ yields a contradiction, and the variables in $\widetilde{u}$ are not used outside P and are not in V, then P is replaced with P_1. (When the find defines variables $\widetilde{u}$ used elsewhere, we cannot remove it.)
- If $P = \mathsf{find}\ (\bigoplus_{j=1}^m \widetilde{u_j}[\widetilde{i}] \leq \widetilde{n_j}$ $\mathsf{suchthat\ defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j$ $\mathsf{then}\ \overline{yield}\langle\rangle)$ $\mathsf{else}\ \overline{yield}\langle\rangle$ and the variables in $\widetilde{u_j}$ are not used outside P and are not in V, then P is replaced with $\overline{yield}\langle\rangle$.
- The defined conditions of find are updated so that Invariant 2 is satisfied. (When such a defined condition guarantees that M is defined, $\mathsf{defined}(M)$ implies $\mathsf{defined}(M')$, and after simplification M' appears in the scope of this condition, then M' has to be added to this condition if it is not already present.)
- If $P = \mathsf{new}\ x : T; P'$ or $\mathsf{let}\ x : T = M\ \mathsf{in}\ P'$ and x is not used in the game and is not in V, then P is replaced with P'.

C.7 Further Simplifications

After applying the game simplifications described above, we further apply the following transformations:

MoveNew We move restrictions downwards in the code as much as possible, when they have no array access using find. A $\mathsf{new}\ x[\widetilde{i}] : T$ cannot be moved under a replication, or under a parallel composition when both sides use x, or a let $\mathsf{let}\ y[\widetilde{i}] : T = M\ \mathsf{in}\ \dots$, $\mathsf{input}\ c[M_1, \dots, M_l](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k)$, $\mathsf{output}\ \overline{c[M_1, \dots, M_l]}\langle N_1, \dots, N_k\rangle$ when x occurs in $M, M_1, \dots, M_l, N_1, \dots, N_k$, or a find when the conditions use x. It can be moved under the other constructs, duplicating it if necessary, when we move it under a find that uses x in several branches. Note that when the restriction $\mathsf{new}\ x[\widetilde{i}] : T$ cannot be moved under an input, a parallel composition, or a replication, it must be written above the output that is located above the considered input, parallel composition or replication, so that the syntax of processes is not violated.

When this transformation duplicates a $\mathsf{new}\ x[\widetilde{i}] : T$ by moving it under a find that uses x in several branches, a subsequent **SArename**(x) enables us to distinguish several cases depending in which branch x is created, which is useful in some proofs.

RemoveAssign(useless): As a particular case of the transformation **RemoveAssign**, we remove useless assignments, that is, assignments to x when x is unused and assignments $\mathsf{let}\ x[\widetilde{i}] : T = y[\widetilde{M}]$. Since removing such assignments may also remove uses of other variables, we repeat this removal until a fixpoint is reached.

SArename(auto): As a particular case of the transformation **SArename**, when x has $m > 1$ definitions and all variable accesses to x are of the form $x[i_1, \dots, i_l]$ under a definition of

$x[i_1, \dots, i_l]$, where $i_1, \dots, i_l$ are the current replication indices at this definition of x (that is, x has no array access using find), we rename x to $x_1, \dots, x_m$ with a different name for each definition.

D Applying the Definition of Security of Primitives

D.1 Formalization of the Transformation

In this appendix, we formalize the transformation performed by exploiting equivalences that come from the definition of security of cryptographic primitives. We require the following conditions for the equivalences $L \approx R$ that model cryptographic primitives:

H0. $[\![L]\!]$ and $[\![R]\!]$ satisfy Invariants 1, 2, and 3. Furthermore, the result of each function in R has the same type as the result of the corresponding function of L.

H1. In L, the functional processes FP are simply terms M; all their array accesses use the current replication indices. (Allowing let or find in L is difficult, because we need to recognize the terms M in a context and in a possibly syntactically modified form.)

H2. L and R have the same structure: same replications, same number of functions, same number of arguments with the same types for each function.

H3. The variables y_j defined by new and x_j defined by function inputs in L and R are distinct from other variables defined in R.

H4. Under $!^{i \leq n}$ with no restriction in L, one can have only a single function $(x_1 : T_1, \dots, x_l : T_l) \rightarrow FP$. (One can transform $!^{i \leq n}((\widetilde{x_1} : \widetilde{T_1}) \rightarrow FP_1, \dots, (\widetilde{x_m} : \widetilde{T_m}) \rightarrow FP_m, !^{i_1 \leq n_1} \dots, \dots, !^{i_{m'} \leq n_{m'}} \dots)$ into $(!^{i \leq n}(\widetilde{x_1} : \widetilde{T_1}) \rightarrow FP_1, \dots, !^{i \leq n}(\widetilde{x_m} : \widetilde{T_m}) \rightarrow FP_m, !^{i_1 \leq n'_1} \dots, \dots, !^{i_{m'} \leq n'_{m'}} \dots)$ in order to eliminate situations that do not satisfy this requirement.)

H5. Replications in L (resp. R) must have pairwise distinct bounds n. (This strengthens the typing: the typing then guarantees that, if several variables are accessed with the same array indices, then these variables are defined under the same replication.)

H6. For all restrictions new $y : T$ that occur above a term M in L, y occurs in M. (This guarantees that, in Hypothesis H′3.1 below, $z_{jk}[M_{j1}, \dots, M_{jq_j}]$ is defined for all $j \leq l$ and $k \leq m_j$. With Hypothesis H4, this guarantees that index_j is well-defined in Hypothesis H′3.1.3 below.)

H7. Finds in R are of the form

$$\text{find } (\textstyle\bigoplus_{j=1}^{m} \widetilde{u_j} \leq \widetilde{n_j} \text{ suchthat defined}(z_{j1}[\widetilde{u_{j1}}], \dots, z_{jl_j}[\widetilde{u_{jl_j}}]) \wedge M_j \text{ then } FP_j) \text{ else } FP'$$

where the following conditions are satisfied:

- For all $1 \leq k \leq l_j$, $\widetilde{u_{jk}}$ is the concatenation of a prefix of the current replication indices (the same prefix for all k) and a non-empty prefix of $\widetilde{u_j}$.
- When $\widetilde{u_j}$ is non-empty, at least one $\widetilde{u_{jk}}$ for $1 \leq k \leq l_j$ is the concatenation of a prefix of the current replication indices with the whole sequence $\widetilde{u_j}$.
- When $l_j \neq 0$, there exists $k \in \{1, \dots, l_j\}$ such that for all $k' \neq k$, $z_{jk'}$ is defined syntactically above all definitions of z_{jk} and $\widetilde{u_{jk'}}$ is a prefix of $\widetilde{u_{jk}}$. (This implies that the same find cannot access variables defined in different functions under the same replication in R.)
- Finally, variables z_{jk} are not defined by a find in R. (Otherwise, the transformation would be considerably more complicated.)

Such equivalences $L \approx R$ are used by the prover by replacing a process Q_0 observationally equivalent to $C[[\![L]\!]]$ with a process Q'_0 observationally equivalent to $C[[\![R]\!]]$, for some evaluation context C. We now give sufficient conditions for a process to be equivalent to $C[[\![L]\!]]$. These conditions essentially guarantee that all uses of certain secret variables of Q_0, in a set S, can be implemented by calling functions of L. These conditions are explained in more detail below.

We first define the function extract used in order to extract information from the left- or right-hand sides of the equivalence.

$$\begin{aligned}
&\mathrm{extract}((x_1 : T_1, \dots, x_l : T_l) \rightarrow M, ()) = \\
&\quad (x_1 : T_1, \dots, x_l : T_l) \rightarrow M \\
&\mathrm{extract}(!^{i \leq n}\text{new } y_1 : T_1; \dots; \text{new } y_l : T_l; (G_1, \dots, G_m), \\
&\quad (j_1, \dots, j_k)) = \\
&\quad (y_1 : T_1, \dots, y_l : T_l), \mathrm{extract}(G_{j_1}, (j_2, \dots, j_k)) \\
&\mathrm{extract}((G_1, \dots, G_m), (j_0, \dots, j_k)) = \\
&\quad \mathrm{extract}(G_{j_0}, (j_1, \dots, j_k))
\end{aligned}$$

We rename the variables of Q_0 such that variables of L and R do not occur in Q_0. Assume that there exist a set of variables S and a set $\mathcal{M}$ of occurrences of terms in Q_0 such that:

H′1. $S \cap V = \emptyset$.

H′2. No term in $\mathcal{M}$ occurs in the condition part of a find (defined$(M_1, \dots, M_l) \wedge M$) or in the channel of an input.

H′3. For each $M \in \mathcal{M}$, there exist a sequence $BL(M) = (j_0, \dots, j_l)$ such that $\mathrm{extract}(L, BL(M)) = (y_{11} : T_{11}, \dots, y_{1m_1} : T_{1m_1}), \dots, (y_{l1} : T_{l1}, \dots, y_{lm_l} : T_{lm_l}), (x_1 : T_1, \dots, x_m : T_m) \rightarrow N$ and a substitution σ such that $M = \sigma N$ (σ applies to the abbreviated form of N in which we write x instead of $x[\widetilde{i}]$) and the following conditions hold:

H′3.1. For all $j \leq l$ and $k \leq m_j$, σy_{jk} is a variable access $z_{jk}[M_{j1}, \dots, M_{jq_j}]$, with $z_{jk} \in S$. We define $z_{jk} = \mathrm{varImL}(y_{jk}, M)$.

H′3.1.1. All definitions of z_{jk} in Q_0 are of the form new $z_{jk}[\ldots] : T_{jk}$, and for all $k \leq m_j$, they occur under the same replications (but they may occur under different replications for different values of j).

H′3.1.2. When $j \neq j'$ or $k \neq k'$, $z_{jk} \neq z_{j'k'}$.

H′3.1.3. The sequence of array indices $M_{j1}, \ldots, M_{jq_j}$ is the same for all $k \leq m_j$ (but may depend on j). We denote by $\mathrm{index}_j(M)$ a substitution that maps the current replication indices at the definition of z_{jk} to $M_{j1}, \ldots, M_{jq_j}$ respectively. If $m_l = 0$, $\mathrm{index}_l(M)$ is not set by the previous definition, so we set $\mathrm{index}_l(M)$ to map the current replication indices at M to themselves. For each $j < l$, there exists a substitution $\rho_j(M)$ such that $\mathrm{index}_j(M) = \mathrm{index}_{j+1}(M) \circ \rho_j(M)$ and the image of $\rho_j(M)$ does not contain the current replication indices at M. We denote by $\mathrm{im}\ \mathrm{index}_j(M)$ the sequence image by $\mathrm{index}_j(M)$ of the sequence of current replication indices at the definition of z_{jk} (so, $\mathrm{im}\ \mathrm{index}_j(M) = (M_{j1}, \ldots, M_{jq_j})$). We define $\mathrm{im}\ \rho_j(M)$ similarly.

H′3.2. For all $j \leq m$, σx_j is a term of type T_j.

H′3.3. All occurrences in Q_0 of a variable in S are either as z_{jk} above or at the root of an argument of a defined test in a find process.

To make it precise which term M each element refers to, we add M as a subscript, writing $y_{jk,M}$ for y_{jk}, $z_{jk,M}$ for z_{jk}, $T_{jk,M}$ for T_{jk}, $x_{j,M}$ for x_j, $T_{j,M}$ for T_j, N_M for N, and σ_M for σ. We also define $\mathrm{nNew}_{j,M} = m_j$, $\mathrm{nNewSeq}_M = l$, and $\mathrm{nInput}_M = m$.

H′4. We say that two terms $M, M' \in \mathcal{M}$ share the first l' sequences of random variables when $y_{jk,M} = y_{jk,M'}$ and $z_{jk,M} = z_{jk,M'}$ for all $j \leq l'$ and $k \leq \mathrm{nNew}_{j,M} = \mathrm{nNew}_{j,M'} \neq 0$. Let l' be the greatest integer such that M and M' share the first l' sequences of random variables. Then we require:

H′4.1. The sets of variables $\{z_{jk,M} \mid j > l' \text{ and } k \leq \mathrm{nNew}_{j,M}\}$ and $\{z_{jk,M'} \mid j > l' \text{ and } k \leq \mathrm{nNew}_{j,M'}\}$ must be disjoint.

H′4.2. $\rho_j(M) = \rho_j(M')$ for all $j < l'$.

H′4.3. If $l' = \mathrm{nNewSeq}_M$ and $N_M = N_{M'}$, then there exists M_0 such that $M = (\mathrm{index}_{l'}(M))M_0$, $M' = (\mathrm{index}_{l'}(M'))M_0$, and M_0 does not contain the current replication indices at M or M'.

When these conditions are satisfied, there exists a context C such that $Q_0 \approx_0^V C[\![L]\!]$.

Terms in $\mathcal{M}$ must not occur in conditions of find (Hypothesis H′2) because such terms may refer to variables defined by find, and by the transformation, these variables might be moved outside their scope, thus violating Invariant 2. Terms in $\mathcal{M}$ must not occur in the channel of an input, because otherwise, after the transformation, the input process might need to perform computations by find or let, forbidden by the syntax. (This requirement is not a limitation in practice, since terms in channels of inputs are typically the current replication indices, so they do not contain cryptographic primitives.)

In Hypothesis H′3, the sequence $BL(M)$ indicates which branch of L corresponds to the term M.

Hypothesis H′3.2 checks that the values received by inputs in L are of the proper type. Hypothesis H′3.1.1 checks that variables $z_{jk,M}$ that correspond to variables defined by new in L are of the proper type. The variables y_{jk} defined by new in L are used only in terms N in L. Correspondingly, Hypothesis H′3.3 checks that the corresponding variables $z_{jk,M} \in S$ are not used elsewhere in Q_0 and Hypothesis H′1 checks that they cannot be used directly by the context.

In L, for distinct j, k, the variables y_{jk} correspond to independent random numbers. Correspondingly, Hypothesis H′3.1.2 requires that the variables $z_{jk,M}$ are created by different restrictions for distinct j, k. In L, the variables y_{jk} are accessed with the same indices for any k (but a fixed j). Correspondingly, Hypothesis H′3.1.3 requires that the variables $z_{jk,M}$ are accessed with the same indices $\mathrm{im}\ \mathrm{index}_j(M)$ for any k. When instances of N and N' both refer to y_{jk} with the same indices, then they also refer to $y_{j'k'}$ with the same indices when $j' \leq j$. Correspondingly, if M and M' refer to the same z_{jk}, by Hypothesis H′4.1, they also refer to the same $z_{j'k'}$ for $j' \leq j$. Moreover, if $\mathrm{index}_j(M)$ and $\mathrm{index}_j(M')$ evaluate to the same bitstrings, then $\mathrm{index}_{j'}(M)$ and $\mathrm{index}_{j'}(M')$ also evaluate to the same bitstrings, since $\mathrm{index}_{j'}(M) = \mathrm{index}_j(M) \circ \rho_{j-1}(M) \circ \ldots \circ \rho_{j'}(M)$ by Hypothesis H′3.1.3 and $\rho_k(M) = \rho_k(M')$ for $k < j$ by Hypothesis H′4.2. These conditions guarantee that we can establish a correspondence from the array cells of variables of S in Q_0 to the array cells of variables defined by new in L, and that this correspondence is an injective function, as required in Section 3.2.

Finally, a term N in L is evaluated at most once for each value of the indices of $y_{l1}, \ldots, y_{lm_l}$, so N is computed for a single value of the arguments $x_1, \ldots, x_m$. Correspondingly, by Hypothesis H′4.3, when M and M' share the $l = \mathrm{nNewSeq}_M$ sequences of random variables and $\mathrm{index}_l(M)$ and $\mathrm{index}_l(M')$ evaluate to the same bitstring, then M and M' evaluate to the same bitstring.

We compute the possible values of the sets S and $\mathcal{M}$ by fixpoint iteration. We start with $\mathcal{M} = \emptyset$ and S containing a single variable of Q_0 bound by a restriction. (We try all possible variables.) When a term M of Q_0 contains a variable in S, we try to find a function in L that corresponds to M, and if we succeed, we add M to $\mathcal{M}$, and add to S variables in M that correspond to variables bound by restrictions in L. (If we fail, the transformation is not possible.) We continue until a fixpoint is reached, in which case all occurrences of variables of S are in terms of $\mathcal{M}$.

We now describe how we construct a process Q'_0 such that $Q'_0 \approx_0^V C[\![R]\!]$.

1. We first move restrictions in the right-hand side of the equivalence, so that they occur above the reception of the arguments of functional processes instead of inside functional processes. As explained below, this is necessary for

the correctness of the subsequent transformation of Q_0, when restrictions appear in the corresponding part of the left-hand side. More precisely, we transform the right-hand side of the equivalence, R, as follows: for each $j_1, \dots, j_l$, if $\mathrm{extract}(L, (j_1, \dots, j_l)) = (y_{11} : T_{11}, \dots, y_{1m_1} : T_{1m_1}), \dots, (y_{l1} : T_{l1}, \dots, y_{lm_l} : T_{lm_l}), (x_1 : T_1, \dots, x_m : T_m) \rightarrow N$ with $m_l \neq 0$ and $\mathrm{extract}(R, (j_1, \dots, j_l)) = (y'_{11} : T'_{11}, \dots, y'_{1m'_1} : T'_{1m'_1}), \dots, (y'_{l1} : T'_{l1}, \dots, y'_{lm'_l} : T'_{lm'_l}), (x_1 : T_1, \dots, x_m : T_m) \rightarrow FP$, for each new $z : T$ in FP,

- we add $z : T$ in the sequence of random variables $y'_{l1} : T'_{l1}, \dots, y'_{lm'_l} : T'_{lm'_l}$;
- if z does not occur in defined conditions of find in R, we remove new $z : T$ from FP;
- otherwise, we replace new $z : T$ with let $z' : T = cst$ for some constant cst and add $z'[\widetilde{M}]$ to each defined condition of R that contains $z[\widetilde{M}]$.

This transformation is needed, because in the right-hand side, a new random number must be chosen exactly for each different call to the function $(x_1 : T_1, \dots, x_m : T_m) \rightarrow FP$. This would not be guaranteed without that transformation, because when the left-hand side N is evaluated at several occurrences with the same random numbers $y_{l1} : T_{l1}, \dots, y_{lm_l} : T_{lm_l}$ ($m_l \neq 0$), these occurrences all correspond to a single call to $(x_1 : T_1, \dots, x_m : T_m) \rightarrow N$, so a single call to $(x_1 : T_1, \dots, x_m : T_m) \rightarrow FP$, but we create a copy of FP for each occurrence. After the transformation, FP contains no choice of random numbers, so we can evaluate it several times without changing the result. When $m_l = 0$, evaluations of N at several occurrences can correspond to different calls to $(x_1 : T_1, \dots, x_m : T_m) \rightarrow N$, so the transformation is not necessary.

2. Next, we create fresh variables corresponding to variables of the right-hand side of the equivalence. For each $M \in \mathcal{M}$, let $\mathrm{extract}(R, BL(M)) = (y'_{11,M} : T'_{11,M}, \dots, y'_{1m'_1,M} : T'_{1m'_1,M}), \dots, (y'_{l1,M} : T'_{l1,M}, \dots, y'_{lm'_l,M} : T'_{lm'_l,M}), (x_{1,M} : T_{1,M}, \dots, x_{m,M} : T_{m,M}) \rightarrow FP_M$ with $l = \mathrm{nNewSeq}_M$, $m = \mathrm{nInput}_M$ and we define $\mathrm{nNew}'_{j,M} = m'_j$. We create fresh variables $z'_{jk,M} = \mathrm{varImR}(y'_{jk,M}, M)$ for each $j \leq \mathrm{nNewSeq}_M$, $k \leq \mathrm{nNew}'_{j,M}$, and $M \in \mathcal{M}$, such that if M and M' share the first l' sequences of random variables, then $z'_{jk,M} = z'_{jk,M'}$ for $j \leq l'$ and $k \leq \mathrm{nNew}'_{j,M}$. All variables $z'_{jk,M}$ are otherwise pairwise distinct.

   We also create a fresh variable $\mathrm{varImR}(x_{j,M}, M)$ for each $j \leq \mathrm{nInput}_M$ and each $M \in \mathcal{M}$, and a fresh variable $\mathrm{varImR}(z, M)$ for each variable z defined by let or new in FP_M and each $M \in \mathcal{M}$.

3. We update the defined conditions of finds, in order to preserve Invariant 2. More precisely, if a defined condition of a find contains $z_{j1,M}[M_1, \dots, M_{l'}]$ for some M, we add defined$(z'_{jk',M}[M_1, \dots, M_{l'}])$ for all $k' \leq \mathrm{nNew}'_{j,M}$ to this condition. (So that accesses to $z'_{jk',M}[M_1, \dots, M_{l'}]$ created when transforming term M satisfy Invariant 2, since accesses to $z_{j1,M}[M_1, \dots, M_{l'}]$ occur in M and satisfy Invariant 2.)

4. We update restrictions corresponding to restrictions of the left-hand side of the equivalence: we either remove them or replace them with restrictions corresponding to the right-hand side of the equivalence. More precisely, when $x \in S$ occurs at the root of a term M_k in a condition defined$(M_1, \dots, M_l)$, we replace its definition new $x : T; Q$ with let $x : T = cst$ in Q for some constant cst; when it does not occur in defined tests, we remove its definition. If $x = z_{j1,M}$ for some M, we add new $z'_{jk,M} : T'_{jk,M}$ for each $k \leq \mathrm{nNew}'_{j,M}$ where new $x : T$ was.

5. Finally, we transform the terms $M \in \mathcal{M}$ corresponding to functions of the left-hand side of the equivalence into their corresponding functional process in the right-hand side. For each term $M \in \mathcal{M}$, let $P_M = C_M[M]$ be the smallest process containing M. (Note that M never occurs in an input, so P_M is an output process.) Let $l = \mathrm{nNewSeq}_M$. We replace P_M with (new $z'_{lk,M} : T'_{lk,M};)_{k \leq \mathrm{nNew}'_{l,M}} P'_M$ if $\mathrm{nNew}_{l,M} = 0$ and $\mathrm{nNew}'_{l,M} > 0$, and with P'_M otherwise, where

   – $P'_M =$ (let $\mathrm{varImR}(x_{k,M}, M) : T_{k,M} = \sigma_M x_{k,M}$ in$)_{k \leq \mathrm{nInput}_M} \mathrm{transf}_{\phi_0, C_M}(FP_M)$.

   – ϕ_0 is defined as follows:

$$\phi_0(x_{j,M}[i_1, \dots, i_l]) = \mathrm{varImR}(x_{j,M}, M)[i'_1, \dots, i'_{l'}]$$
$$\phi_0(z[i_1, \dots, i_l]) = \mathrm{varImR}(z, M)[i'_1, \dots, i'_{l'}]$$
$$\phi_0(y'_{jk,M}[i_1, \dots, i_j]) = \mathrm{varImR}(y'_{jk,M}, M)[\mathrm{im\ index}_j(M)]$$

   where $i_1, \dots, i_l$ are the current replication indices at the definition of $x_{j,M}$ in R, $i'_1, \dots, i'_{l'}$ are the current replication indices at M in Q_0, and z is a variable defined by let or new in FP_M.

   – A function ϕ from array accesses to array accesses is extended to terms as a substitution, by $\phi(f(M_1, \dots, M_m)) = f(\phi(M_1), \dots, \phi(M_m))$.

   – $\mathrm{transf}_{\phi, C_M}(FP)$ is defined recursively as follows:

$$\mathrm{transf}_{\phi,C_M}(M') = C_M[\phi(M')]$$
$$\mathrm{transf}_{\phi,C_M}(\mathsf{new}\ z : T; FP') = \mathsf{new}\ \mathrm{varImR}(z, M) : T; \mathrm{transf}_{\phi,C_M}(FP')$$
$$\mathrm{transf}_{\phi,C_M}(\mathsf{let}\ z : T = M'\ \mathsf{in}\ FP') = \mathsf{let}\ \mathrm{varImR}(z, M) : T = \phi(M')\ \mathsf{in}\ \mathrm{transf}_{\phi,C_M}(FP')$$
$$\mathrm{transf}_{\phi,C_M}(\mathsf{find}(\bigoplus_{j=1}^m FB_j)\ \mathsf{else}\ FP') = \mathsf{find}(\bigoplus_{j=1}^m \mathrm{transf}_{\phi,C_M}(FB_j))\ \mathsf{else}\ \mathrm{transf}_{\phi,C_M}(FP')$$

   and for find branches FB, $\mathrm{transf}_{\phi,C_M}(FB)$ is defined as follows:

$$\mathrm{transf}_{\phi,C_M}(\mathsf{suchthat}\ M'\ \mathsf{then}\ FP') = \mathsf{suchthat}\ \phi(M')\ \mathsf{then}\ \mathrm{transf}_{\phi,C_M}(FP')$$

$\text{transf}_{\phi,C_M}(\widetilde{u} \leq \widetilde{n}$ suchthat
defined$(z_k[M_{k1}, \dots, M_{kl'_k}]_{1 \leq k \leq l}) \wedge M_1$ then $FP') =$
$\bigoplus_{M' \in \mathcal{M}'} \widetilde{u'} \leq \widetilde{n'}$ suchthat
defined$(\phi'(z_k[M_{k1}, \dots, M_{kl'_k}])_{1 \leq k \leq l}) \wedge$
$\text{im index}_{j_1}(M')\{\widetilde{u'}/\widetilde{i'}\} = \text{im index}_{j_1}(M) \wedge$
$\phi'(M_1)$ then $\text{transf}_{\phi',C_M}(FP')$

where $l \neq 0$; j_1 is the length of the prefix of the current replication indices that occurs in $M_{k1}, \dots, M_{kl'_k}$ (by Hypothesis H7); $\mathcal{M}'$ is the set of $M' \in \mathcal{M}$ such that $\text{varImR}(z_k, M')$ is defined for $k \leq l$ and M' and M share the first j_1 sequences of random variables; $\widetilde{i'}$ is the sequence of current replication indices at M'; $\widetilde{u'}$ is a sequence formed with a fresh variable for each variable in $\widetilde{i'}$; $\widetilde{n'}$ is the sequence of bounds of replications above M'; ϕ' is an extension of ϕ with $\phi'(z_k[M_{k1}, \dots, M_{kl'_k}]) = \text{varImR}(z_k, M')[\text{im index}_j(M')\{\widetilde{u'}/\widetilde{i'}\}]$ if $z_k = y'_{jk',M'}$ for some k', and $\phi'(z_k[M_{k1}, \dots, M_{kl'_k}]) = \text{varImR}(z_k, M')[\widetilde{u'}]$ if z_k is defined by let or by a function input.

The two essential parts of the transformation are the last two ones, numbered 4 and 5. In step 4, we add the restrictions to create random variables that correspond to random variables of R. We create the variables $z'_{jk,M}$ at the place where $z_{j1,M}$ was created in the initial game (We could have chosen $z_{jk',M}$ for any k'.), or when there is no $z_{j1,M}$, we have $j = \text{nNewSeq}_M$ and we create $z'_{jk,M}$ just before evaluating M. In step 5, we transform the term M itself into the corresponding functional process of R, FP_M. The only delicate part for evaluating FP_M is the case of find: instead of looking up arrays of R, we look up the corresponding arrays of Q'_0 given by the mapping ϕ.

D.2 Extension

We introduce a small extension to the equivalences $(G_1, \dots, G_m) \approx (G'_1, \dots, G'_m)$ described in Section 3.2. These equivalences become $(G_1\ mode_1, \dots, G_m\ mode_m) \approx (G'_1, \dots, G'_m)$, where $mode_j$ is either empty or $[all]$. The mode $[all]$ is an indication for the prover, to guide the application of the equivalence without changing its semantics. When $mode_j = [all]$, $\mathcal{M}$ must contain all occurrences in the initial game Q of the root function symbols of terms M inside G_j. When $mode_j$ is empty, at least one variable defined by new in G_j must correspond to a variable in S.

The following hypotheses guarantee the good usage of modes:

H8. At most one $mode_j$ can be empty. (Otherwise, when several sets of random variables can be chosen for each G_j, there are many possible combinations for applying the transformation.)

H9. If G_j is of the form $!^{i \leq n}(x_1 : T_1, \dots, x_l : T_l) \rightarrow FP$ without any restriction, then $mode_j = [all]$. (A restriction is needed in the definition of empty mode.)

D.3 Modeling other Primitives

This appendix gives the definition of a number of cryptographic primitives in our prover.

D.3.1 Super-Pseudo-Random Permutations (SPRP)

T_r large, fixed length; T large, fixed length
$e, d : T \times T_k \rightarrow T$
$\text{kgen} : T_r \rightarrow T_k$

$\forall m : T, \forall r : T_r, d(e(m, \text{kgen}(r)), \text{kgen}(r)) = m$
$\forall m : T, \forall r : T_r, e(d(m, \text{kgen}(r)), \text{kgen}(r)) = m$

$!^{i'' \leq n''}$ new $r : T_r$; (
  $!^{i \leq n}(x : T) \rightarrow e(x, \text{kgen}(r))$,
  $!^{i' \leq n'}(m : T) \rightarrow d(m, \text{kgen}(r)))$
$\approx$
$!^{i'' \leq n''}$ new $r : T_r$; (
  $!^{i \leq n}(x : T) \rightarrow$
    find $u \leq n$ suchthat defined$(x[u], r'[u]) \wedge$
      $x = x[u]$ then $r'[u]$
    $\oplus\ u \leq n'$ suchthat defined$(r''[u], m[u]) \wedge$
      $x = r''[u]$ then $m[u]$
    else new $r' : T; r'$,
  $!^{i' \leq n'}(m : T) \rightarrow$
    find $u \leq n$ suchthat defined$(r'[u], x[u]) \wedge$
      $m = r'[u]$ then $x[u]$
    $\oplus\ u \leq n'$ suchthat defined$(m[u], r''[u]) \wedge$
      $m = m[u]$ then $r''[u]$
    else new $r'' : T; r'')$

This equivalence expresses that the encryption and decryption oracles can be replaced with inverse random permutations. These random permutations are built as follows for the encryption oracle: when we receive an argument x already passed to the encryption oracle, we return the previous result; when we receive the result of a previous call to the decryption oracle, we return the argument of the decryption oracle in that call; otherwise, we return a fresh random number. (Collisions between random numbers in T_r have negligible probability, so we obtain permutations except in cases of negligible probability.) The construction is similar for the decryption oracle.

D.3.2 Public-Key Cryptography

UF-CMA Signature

T_r large, fixed length; T'_r fixed length
$s, s' : T \times T_{sk} \times T'_r \rightarrow T_s$
$c, c' : T \times T_{pk} \times T_s \rightarrow bool$
$\text{skgen}, \text{skgen}' : T_r \rightarrow T_{sk}$
$\text{pkgen}, \text{pkgen}' : T_r \rightarrow T_{pk}$

$$\forall m : T, \forall r : T_r, \forall r' : T'_r,$$
$$c(m, \text{pkgen}(r), s(m, \text{skgen}(r), r')) = \text{true}$$
$$\forall m : T, \forall r : T_r, \forall r' : T'_r,$$
$$c'(m, \text{pkgen}'(r), s'(m, \text{skgen}'(r), r')) = \text{true}$$
$$\mathsf{new}\ x : T_r; \mathsf{new}\ y : T_r; f(x) = f(y) \approx x = y$$
$$\text{for } f \in \{\text{pkgen}, \text{skgen}, \text{pkgen}', \text{skgen}'\}$$

$$!^{i\le n}\mathsf{new}\ r : T_r; ($$
$$() \to \text{pkgen}(r),$$
$$!^{i'\le n'}\mathsf{new}\ r' : T'_r; (x : T) \to s(x, \text{skgen}(r), r')),$$
$$!^{i''\le n''}(m : T, y : T_{pk}, si : T_s) \to c(m, y, si)\ [all]$$
$$\approx$$

1. $!^{i\le n}\mathsf{new}\ r : T_r; ($
2. $\quad () \to \text{pkgen}'(r),$
3. $\quad !^{i'\le n'}\mathsf{new}\ r' : T'_r; (x : T) \to s'(x, \text{skgen}'(r), r')),$
4. $!^{i''\le n''}(m : T, y : T_{pk}, si : T_s) \to$
5. $\quad \mathsf{find}\ u \le n, u' \le n'\ \mathsf{suchthat\ defined}(r[u], x[u, u'])$
6. $\qquad \wedge\, y = \text{pkgen}'(r[u]) \wedge m = x[u, u']$
7. $\qquad \wedge\, c'(m, y, si)\ \mathsf{then}\ \text{true}\ \mathsf{else}$
8. $\quad \mathsf{find}\ u \le n\ \mathsf{suchthat\ defined}(r[u])$
9. $\qquad \wedge\, y = \text{pkgen}'(r[u])\ \mathsf{then}\ \text{false}\ \mathsf{else}$
10. $\quad c(m, y, si)$

The first three lines of each side of the equivalence express that the generation of public keys and the computation of the signature are left unchanged in the transformation. The verification of a signature $c(m, y, si)$ is replaced with a lookup in the previously computed signatures: if the signature is checked using one of the keys $\text{pkgen}'(r[u])$ (that is, if $y = \text{pkgen}'(r[u])$), then it can be valid only when it has been computed by the signature oracle $s'(x, \text{skgen}'(r[u]), r')$, that is, when $m = x[u, u']$ for some u'. Lines 5-7 of the right-hand side of the equivalence try to find such a u' and return true when they succeed. Lines 8-9 of the right-hand side returns false when no such u' is found in lines 5-7, but $y = \text{pkgen}'(r[u])$ for some u. The last line handles the case when the key y is not $\text{pkgen}'(r[u])$. In this case, we check the signature as before. (Using c and not c' in the last line of the transformation allows to reapply this transformation with another value of r.)

We can model deterministic signatures in a similar way, by removing the third argument of s.

IND-CCA2 Public-Key Encryption

T_r large, fixed length; T'_r fixed length
$$\text{enc}, \text{enc}' : T \times T_{pk} \times T'_r \to T_e$$
$$\text{dec}, \text{dec}' : T_e \times T_{sk} \to T_\bot$$
$$\text{skgen}, \text{skgen}' : T_r \to T_{sk}$$
$$\text{pkgen}, \text{pkgen}' : T_r \to T_{pk}$$
$\text{i}_\bot : T \to T_\bot$ (poly-injective)
$$\text{Z}_T : T$$

$$\forall m : T, \forall r : T_r, \forall r' : T'_r,$$
$$\text{dec}(\text{enc}(m, \text{pkgen}(r), r'), \text{skgen}(r)) = \text{i}_\bot(m)$$
$$\forall m : T, \forall r : T_r, \forall r' : T'_r,$$
$$\text{dec}'(\text{enc}'(m, \text{pkgen}'(r), r'), \text{skgen}'(r)) = \text{i}_\bot(m)$$
$$\mathsf{new}\ x : T_r; \mathsf{new}\ y : T_r; f(x) = f(y) \approx x = y$$
$$\text{for } f \in \{\text{pkgen}, \text{pkgen}', \text{skgen}, \text{skgen}'\}$$

$$!^{i\le n}\mathsf{new}\ r : T_r; ($$
$$() \to \text{pkgen}(r),$$
$$!^{i'\le n'}(m : T_e) \to \text{dec}(m, \text{skgen}(r))),$$
$$!^{i''\le n''}\mathsf{new}\ r' : T'_r; (x : T, y : T_{pk}) \to \text{enc}(x, y, r')\ [all]$$
$$\approx$$
$$!^{i\le n}\mathsf{new}\ r : T_r; ($$
$$!^{i\le n}() \to \text{pkgen}'(r),$$
$$!^{i'\le n'}(m : T_e) \to \mathsf{find}\ u \le n''\ \mathsf{suchthat}$$
$$\mathsf{defined}(m'[u], x[u], y[u]) \wedge y[u] = \text{pkgen}'(r)$$
$$\wedge\, m = m'[u]\ \mathsf{then}\ \text{i}_\bot(x[u])\ \mathsf{else}\ \text{dec}'(m, \text{skgen}'(r))),$$
$$!^{i''\le n''}(x : T, y : T_{pk}) \to$$
$$\mathsf{find}\ u' \le n\ \mathsf{suchthat\ defined}(r[u']) \wedge y = \text{pkgen}'(r[u'])$$
$$\mathsf{then\ new}\ r' : T'_r;$$
$$\mathsf{let}\ m' : T_e = \text{enc}'(\text{Z}_T, \text{pkgen}'(r[u']), r')\ \mathsf{in}\ m'$$
$$\mathsf{else\ new}\ r'' : T'_r; \text{enc}(x, y, r'')$$

When no decryption is present, this transformation reduces to IND-CPA public key encryption, described below.

IND-CPA Public-Key Encryption

T_r large, fixed length; T'_r fixed length
$$\text{enc}, \text{enc}' : T \times T_{pk} \times T'_r \to T_e$$
$$\text{dec} : T_e \times T_{sk} \to T_\bot$$
$$\text{skgen} : T_r \to T_{sk}$$
$$\text{pkgen}, \text{pkgen}' : T_r \to T_{pk}$$
$\text{i}_\bot : T \to T_\bot$ (poly-injective)
$$Z_T : T$$

$$\forall m : T, \forall r : T_r, \forall r' : T'_r,$$
$$\text{dec}(\text{enc}(m, \text{pkgen}(r), r'), \text{skgen}(r)) = \text{i}_\bot(m)$$
$$\mathsf{new}\ x : T_r; \mathsf{new}\ y : T_r; f(x) = f(y) \approx x = y$$
$$\text{for } f \in \{\text{pkgen}, \text{skgen}, \text{skgen}'\}$$

$$!^{i\le n}\mathsf{new}\ r : T_r; () \to \text{pkgen}(r),$$
$$!^{i'\le n'}\mathsf{new}\ r' : T'_r; (x : T, y : T_{pk}) \to \text{enc}(x, y, r')\ [all]$$
$$\approx$$
$$!^{i\le n}\mathsf{new}\ r : T_r; () \to \text{pkgen}'(r),$$
$$!^{i'\le n'}(x : T, y : T_{pk}) \to$$
$$\mathsf{find}\ u \le n\ \mathsf{suchthat\ defined}(r[u]) \wedge y = \text{pkgen}'(r[u])$$
$$\mathsf{then\ new}\ r' : T'_r; \text{enc}'(Z_T, \text{pkgen}'(r[u]), r')$$
$$\mathsf{else\ new}\ r'' : T'_r; \text{enc}(x, y, r'')$$

D.3.3 Hash Functions

Collision Resistant Hash Function

T_k fixed length
$h : T_k \times bitstring \rightarrow T$
$\mathsf{new}\ k : T_k; \forall x : bitstring, \forall y : bitstring,$
$h(k, x) = h(k, y) \approx x = y$

Hash Function in the Random Oracle Model

T fixed length
$h : bitstring \rightarrow T$
$!^{i \leq n}(x : bitstring) \rightarrow h(x)\ [all]$
$\approx_0$
$!^{i \leq n}(x : bitstring) \rightarrow$
$\mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[u], r[u]) \wedge x = x[u]$
$\mathsf{then}\ r[u]$
$\mathsf{else}\ \mathsf{new}\ r : T; r$

Note that the game must include, in parallel with the protocol to verify, the process $!^{i \leq n} c(x : bitstring); \overline{c}\langle h(x) \rangle$. Otherwise, the prover would incorrectly assume that the adversary cannot compute the hash function. This particularity is related to the fact that a random oracle is unimplementable: otherwise, the adversary could implement it without being explicitly given access to it.

D.3.4 Xor

$\mathrm{xor} : T \times T \rightarrow T$ (commutative)
$\forall x : T, y : T, \mathrm{xor}(x, \mathrm{xor}(x, y)) = y.$
$\forall x : T, y : T, z : T, (\mathrm{xor}(x, z) = \mathrm{xor}(y, z)) = (x = y).$
$!^{i \leq n}\mathsf{new}\ k : T; (x : T) \rightarrow \mathrm{xor}(x, k)$
$\approx_0$
$!^{i \leq n}\mathsf{new}\ k : T; (x : T) \rightarrow k$

This modeling of xor could be improved by taking into account more equations, in particular associativity.

E Proofs

E.1 Proof of Proposition 1

The proof that Q'_0 satisfies Invariants 1, 2, and 3 is in general easy, and the proof of $Q_0 \approx_0^V Q'_0$ relies on a correspondence between traces of $C[Q_0]$ and traces of $C[Q'_0]$, with the same probability and such that a configuration of the trace of $C[Q_0]$ executes $\overline{c}\langle a \rangle$ immediately if and only if the corresponding configuration of the corresponding trace of $C[Q'_0]$ executes $\overline{c}\langle a \rangle$ immediately. This correspondence is obtained by replacing some internal actions of Q_0 with corresponding internal actions of Q'_0. We sketch the proof only for the cases of **SArename**(x) and **Simplify**, and leave the case of **RemoveAssign**(x) to the reader.

Proof sketch of Proposition 1 for SArename(x) The process Q'_0 satisfies Invariant 1 because definitions of variables duplicated by **SArename** all occur in a different branch of a find.

For Invariant 2, each variable access $x_j[M_1, \ldots, M_l]$ in Q'_0 comes from a variable access $x[M_1, \ldots, M_l]$ in Q_0. Since Q_0 satisfies Invariant 2, either this access is under its definition, in which case **SArename**(x) has replaced this definition of x with a definition of x_j, so $x_j[M_1, \ldots, M_l]$ is under its definition in Q'_0; or this access is in a defined test, in which case it is also in a defined test in Q'_0; or this access is in a branch of find with a condition $\mathsf{defined}(N_1, \ldots, N_{l'})$ such that $x[M_1, \ldots, M_l]$ is a subterm of N_j for some $j \leq l'$, in which case $x[M_1, \ldots, M_l]$ has been substituted with $x_j[M_1, \ldots, M_l]$ in this branch of find, so $x_j[M_1, \ldots, M_l]$ is under a suitable defined condition. Therefore, Q'_0 satisfies Invariant 2.

For Invariant 3, the type environment $\mathcal{E}'$ for Q'_0 is obtained from the type environment $\mathcal{E}$ for Q_0, by setting $\mathcal{E}'(x_1) = \ldots = \mathcal{E}'(x_m) = \mathcal{E}(x)$ and $\mathcal{E}'(x)$ is not defined. (Indeed, all definitions of x in Q_0 have the same type $\mathcal{E}(x)$, which is therefore the type of the definitions of $x_j, j \leq m$ in Q'_0.) The proof of $\mathcal{E}' \vdash Q'_0$ is obtained from the proof of $\mathcal{E} \vdash Q_0$, by replacing requests to $\mathcal{E}(x)$ with requests to $\mathcal{E}(x_j)$ for some $j \leq m$, and duplicating parts of the proof of $\mathcal{E} \vdash Q_0$ that correspond to duplicated branches of find.

Finally, let us prove that $Q_0 \approx_0^V Q'_0$. We denote by $SArename(x, Q)$ the process obtained by applying **SArename**(x) to Q. Let j be a partial function from l-tuples of indices $a_1, \ldots, a_l$ to subscripts $1, \ldots, m$ of variable x. Informally, j is such that $x[a_1, \ldots, a_l]$ in a trace of Q_0 corresponds to $x_{j(a_1,\ldots,a_l)}[a_1, \ldots, a_l]$ in the corresponding trace of Q'_0. We define a function $SArename_j$ that relates configurations in a trace of Q_0 to configurations in a trace of the renamed process Q'_0. Below, we will show that this function maps traces of Q_0 to traces of Q'_0 of the same probability, which will show the desired equivalence $Q_0 \approx_0^V Q'_0$.

- We define $SArename_j$ for terms so that $SArename_j(x, E, M)$ replaces occurrences of x in M with the appropriate x_j. More precisely,

$$SArename_j(x, E, x[M_1, \ldots, M_l]) = x_{j(a_1,\ldots,a_l)}[SArename_j(x, E, M_1), \ldots, SArename_j(x, E, M_l)]$$
when $E, M_k \Downarrow a_k$ for $k \leq l$ and $x[a_1, \ldots, a_l] \in \mathrm{Dom}(E)$;
$$SArename_j(x, E, y[M_1, \ldots, M_l]) = y[SArename_j(x, E, M_1), \ldots, SArename_j(x, E, M_l)]$$
when $y \neq x$;
$$SArename_j(x, E, f(M_1, \ldots, M_l)) = f(SArename_j(x, E, M_1), \ldots, SArename_j(x, E, M_l));$$
$$SArename_j(x, E, i) = i$$

- We define $SArename_j$ for (input and output) processes as follows: $SArename_j(x, E, P_1)$ first computes $SArename(x, P_1) = P_2$. More precisely, it renames each definition of x to the name used when renaming the whole

process Q_0; it replaces variable accesses to x with variable accesses to x_j when the definition of x that caused this replacement in Q_0 also occurs in P_1; it duplicates branches of find as $SArename(x, Q_0)$, renaming variable accesses to x into variable accesses to x_j when the find that caused this replacement in Q_0 also occurs in P_1. (When a variable access to x is under both a definition of x and find, or under several nested finds that guarantee that it is defined, it is important to follow exactly the renaming procedure that happened in Q_0. Formally, this can be done by annotating each construct in processes with a distinct occurrence symbol and by reducing annotated processes. When we perform $SArename(x, Q_0)$, we can then remember the occurrence symbols of the constructs that cause each variable renaming.) Finally, $SArename_j$ replaces each term M in P_2 with $SArename_j(x, E, M)$.

- We also define $SArename_j$ for environments: $E' = SArename_j(x, E)$ if and only if $E'(x_{j(a_1,\dots,a_l)}[a_1, \dots, a_l]) = E(x[a_1, \dots, a_l])$ when $x[a_1, \dots, a_l] \in \mathrm{Dom}(E)$, $E'(y[a_1, \dots, a_l]) = E(y[a_1, \dots, a_l])$ when $y \neq x$ and $y[a_1, \dots, a_l] \in \mathrm{Dom}(E)$, and $E'(y[a_1, \dots, a_l])$ is undefined in all other cases.
- We extend $SArename_j$ to semantic configurations:

$$\begin{aligned}&SArename_j(x, (E, P, \mathcal{Q}, \mathcal{C})) =\\ &\quad (SArename_j(x, E), SArename_j(x, E, P),\\ &\quad \{SArename_j(x, E, Q_1) \mid Q_1 \in \mathcal{Q}\}, \mathcal{C})\end{aligned}$$

We also define $SArename_j(x, (E, \mathcal{Q}, \mathcal{C}))$ in the same way.

We first show that if $E, M \Downarrow a$, then

$$SArename_j(x, E), SArename_j(x, E, M) \Downarrow a$$

The proof proceeds by induction on M. The only interesting case is $M = x[M_1, \dots, M_l]$. Since $E, M \Downarrow a$ has been derived by (Var), $E, M_k \Downarrow a_k$ for all $k \leq l$ and $a = E(x[a_1, \dots, a_l])$. By induction hypothesis, $SArename_j(x, E), SArename_j(x, E, M_k) \Downarrow a_k$ for all $k \leq l$. Moreover,

$$\begin{aligned}&SArename_j(x, E, x[M_1, \dots, M_l]) =\\ &\quad x_{j(a_1,\dots,a_l)}[SArename_j(x, E, M_1), \dots,\\ &\quad\quad SArename_j(x, E, M_l)]\end{aligned}$$

and

$$\begin{aligned}&SArename_j(x, E)(x_{j(a_1,\dots,a_l)}[a_1, \dots, a_l]) =\\ &\quad E(x[a_1, \dots, a_l]) = a\end{aligned}$$

so $SArename_j(x, E), SArename_j(x, E, M) \Downarrow a$.

Next, we can show by cases on the reduction $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$ that, if $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$, then

$$SArename_j(x, (E, \mathcal{Q}, \mathcal{C})) \rightsquigarrow SArename_j(x, (E', \mathcal{Q}', \mathcal{C}')).$$

Hence

$$\begin{aligned}&SArename_j(x, \mathrm{reduce}(E, \mathcal{Q}, \mathcal{C})) =\\ &\quad \mathrm{reduce}(SArename_j(x, (E, \mathcal{Q}, \mathcal{C})))\end{aligned}$$

Let C be any evaluation context acceptable for Q_0, Q'_0, V. We show that for each trace $\mathrm{initConfig}(C[Q_0]) \to_\eta \dots \to_\eta E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$, there exists a trace $\mathrm{initConfig}(C[Q'_0]) \to_\eta \dots \to_\eta E'_m, P'_m, \mathcal{Q}'_m, \mathcal{C}_m$ with the same probability, and a function j_m such that $E'_m, P'_m \mathcal{Q}'_m, \mathcal{C}_m = SArename_{j_m}(x, (E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m))$. The proof proceeds by induction on the length m of the trace. For the induction step, we distinguish cases depending on the last reduction step of the trace.

- Initial case $m = 0$: $\mathrm{fc}(C[Q_0]) = \mathrm{fc}(C[Q'_0])$ since the transformation **SArename** does not modify channels. Let j_0 be the function defined nowhere. We have, $C[Q'_0] = SArename_{j_0}(x, \emptyset, C[Q_0])$. Indeed, since $x \notin V$, $x \notin \mathrm{var}(C)$, so

  $$\begin{aligned}&SArename_{j_0}(x, \emptyset, C[Q_0]) = SArename(x, C[Q_0]) =\\ &\quad C[SArename(x, Q_0)] = C[Q'_0]\end{aligned}$$

  Therefore,

  $$\begin{aligned}&SArename_{j_0}(x, (\emptyset, \{C[Q_0]\}, \mathrm{fc}(C[Q_0]))) =\\ &\quad (\emptyset, \{C[Q'_0]\}, \mathrm{fc}(C[Q'_0]))\end{aligned}$$

  Hence we have

  $$\begin{aligned}&SArename_{j_0}(x, \mathrm{reduce}(\emptyset, \{C[Q_0]\}, \mathrm{fc}(C[Q_0]))) =\\ &\quad \mathrm{reduce}(\emptyset, \{C[Q'_0]\}, \mathrm{fc}(C[Q'_0]))\end{aligned}$$

  Thus,

  $$\begin{aligned}&SArename_{j_0}(x, \mathrm{initConfig}(C[Q_0])) =\\ &\quad \mathrm{initConfig}(C[Q'_0])\end{aligned}$$

- The last step of the trace is a definition of $x[a_1, \dots, a_l]$: By induction hypothesis, we have a trace of length $m - 1$, with an associated function j_{m-1}. Since $C[Q_0]$ satisfies Invariant 1, the configuration $E_{m-1}, P_{m-1}, \mathcal{Q}_{m-1}, \mathcal{C}_{m-1}$ satisfies Invariant 4, so $x[a_1, \dots, a_l] \notin \mathrm{Dom}(E_{m-1})$. Since $P'_{m-1} = SArename_{j_{m-1}}(x, E_{m-1}, P_{m-1})$, the first instruction of P'_{m-1} is a definition of $x_k[a_1, \dots, a_l]$ for some k (using the property "if $E, M \Downarrow a$, then $SArename_j(x, E), SArename_j(x, E, M) \Downarrow a$" shown above to prove that the indices of x, resp. x_k, are the same in the execution of P_{m-1} and of P'_{m-1}). We define $j_m = j_{m-1}[(a_1, \dots, a_l) \mapsto k]$, and show that we obtain a suitable trace of length m with this function j_m.
- The last step of the trace is a find whose defined condition refers to x: By induction hypothesis, we have a trace of length $m - 1$, with an associated function j_{m-1}. If a branch FB of the find in P_{m-1} succeeds for certain values of the variables defined by find, exactly one of its copies succeeds in P'_{m-1}, the copy whose defined condition refers to $x_{j_{m-1}(a_1,\dots,a_l)}[a_1, \dots, a_l]$ when the defined condition of the branch FB in P_{m-1} refers to $x[a_1, \dots, a_l]$. If a branch of the find fails in P_{m-1}, all its copies fail in P'_{m-1}. Therefore, the number $|S|$ of successful choices of the find is the same in P_{m-1} and in P'_{m-1}. Hence, the probability that each successful branch is taken is the same. When P_{m-1}

executes a successful branch, we build the corresponding trace of P'_{m-1} by executing the successful copy of this branch. When P_{m-1} executes the else branch, P'_{m-1} also executes the else branch. So we obtain a suitable trace of length m with associated function $j_m = j_{m-1}$ (except when the find also defines $x[a'_1, \dots, a'_l]$, in which case the previous item of the proof must also be applied).

- All other cases are easy: they execute in the same way in P_{m-1} and in P'_{m-1}.

We also show the converse property, that for each trace $\mathrm{initConfig}(C[Q'_0]) \to_\eta \dots \to_\eta E'_m, P'_m, \mathcal{Q}'_m, \mathcal{C}_m$, there exists a trace $\mathrm{initConfig}(C[Q_0]) \to_\eta \dots \to_\eta E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ with the same probability and

$$E'_m, P'_m \mathcal{Q}'_m, \mathcal{C}_m = \mathit{SArename}_{j_m}(x, (E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m)).$$

The proof is similar to the proof above.

If $E'_m, P'_m \mathcal{Q}'_m, \mathcal{C}_m = \mathit{SArename}_{j_m}(x, (E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m))$, then for all channels c and bitstrings a, $E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ executes $\overline{c}\langle a\rangle$ immediately if and only if $E'_m, P'_m, \mathcal{Q}'_m, \mathcal{C}_m$ executes $\overline{c}\langle a\rangle$ immediately. So $\Pr[C[Q_0] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C[Q'_0] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$. Therefore, $Q_0 \approx_0^V Q'_0$. □

Proof sketch of Proposition 1 for Simplify The proof of Invariants 1, 2, and 3 is relatively easy, so we focus on the proof of $Q_0 \approx^V Q'_0$.

Let C be any evaluation context acceptable for Q_0, Q'_0, V. Let $q(\eta)$ be the maximum runtime of $C[Q_0]$, where q is a polynomial. We denote by $\mathbb{C}_0$ the initial configuration of $C[Q_0]$, $\mathrm{initConfig}(C[Q_0])$.

We define $p_{\max}(\eta) = \max(\{\frac{1}{|I_\eta(T)|} \mid T \text{ is a large type}\} \cup \{p(\eta)$ associated to user-defined rewrite rules, for an adversary of runtime $q(\eta)\})$. The probability $p_{\max}(\eta)$ is negligible, since it is the maximum of a constant number of negligible functions. We shall prove below that the probability that a desired fact does not hold is at most $q'(\eta)p_{\max}(\eta)$, where q' is a polynomial, so it is negligible.

The proof follows the structure of the simplification algorithm: we prove the correctness of each component of the algorithm separately.

Correctness of the collection of true facts. We consider a slightly modified semantics for our calculus, in which each process is accompanied with a substitution that defines the values of the replication indices in that process. For example, the rule (Repl) becomes in this semantics:

$$E, \{(\sigma, !^{i\le n}Q)\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow \\ E, \{(\sigma[i \mapsto a], Q) \mid a \in [1, I_\eta(n)]\} \uplus \mathcal{Q}, \mathcal{C}$$

When evaluating a term M in a process with substitution (σ, Q) or (σ, P), we now use $E, \sigma, M \Downarrow a$ instead of $E, M \Downarrow a$, with the rule $E, \sigma, i \Downarrow \sigma i$ instead of (Cst), and the other rules modified accordingly.

The judgment $E, \sigma \vdash F$ means that a fact F holds in environment E and substitution σ. It is defined by $E, \sigma \vdash M$ if and only if $E, \sigma, M \Downarrow \mathrm{true}$; $E, \sigma \vdash \mathsf{defined}(M)$ if and only if $E, \sigma, M \Downarrow a$ for some a; $E, \sigma \vdash \mathit{elsefind}((u_1 \le n_1, \dots, u_m \le n_m), (M_1, \dots, M_l), M)$ if and only if for all $x_1 \in [1, I_\eta(n_1)], \dots, x_m \in [1, I_\eta(n_m)]$, we have $E, \sigma', (\mathsf{defined}(M_1, \dots, M_l) \wedge M) \Downarrow \mathrm{false}$ where $\sigma' = \sigma[u_1 \mapsto x_1, \dots, u_m \mapsto x_m]$. We extend this definition to sets of facts naturally. We say that $\mathcal{F}_P$ *is correct for all* P when $\mathbb{C}_0 \xrightarrow{p}_t \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ implies $E, \sigma \vdash \mathcal{F}_P$. Our goal is to show that $\mathcal{F}_P$ is indeed correct for all P.

For occurrences of processes P, Q in C and in the process $\overline{start}\langle\rangle; 0$ used in the initial configuration, we let $\mathcal{F}_P = \mathcal{F}_Q = \mathcal{F}_P^{\mathrm{Fut}} = \mathcal{F}_P^{\mathrm{ElseFind}} = \mathcal{F}_Q^{\mathrm{ElseFind}} = \emptyset$.

We show **S0**: immediately after calling collectFacts, if $E_1, (\sigma_1, P_1), \mathcal{Q}_1, \mathcal{C}_1 \xrightarrow{p}_t E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ then $E, \sigma \vdash \mathcal{F}_P$. If the reduced process is in C, the result is obvious since $\mathcal{F}_P = \emptyset$. Otherwise, we proceed by cases on the reduction $E_1, (\sigma_1, P_1), \mathcal{Q}_1, \mathcal{C}_1 \xrightarrow{p}_t E, (\sigma, P), \mathcal{Q}, \mathcal{C}$. For example, in the case (Let), $E_1, \sigma, M \Downarrow a$, $a \in I_\eta(T)$, and $E_1, (\sigma, \mathsf{let}\ x[\widetilde{i}] : T = M \text{ in } P), \mathcal{Q}, \mathcal{C} \xrightarrow{1}_L E = E_1[x[\sigma\widetilde{i}] \mapsto a], (\sigma, P), \mathcal{Q}, \mathcal{C}$. We have $\mathcal{F}_P = \{\mathsf{defined}(x[\widetilde{i}]), x[\widetilde{i}] = M\}$. Since $E, \sigma, x[\widetilde{i}] \Downarrow a$, we have $E, \sigma \vdash \mathsf{defined}(x[\widetilde{i}])$. We also have $E, \sigma, M \Downarrow a$, so $E, \sigma \vdash x[\widetilde{i}] = M$, so $E, \sigma \vdash \mathcal{F}_P$. We proceed in a similar way for the other cases.

We show that, immediately after calling collectFacts, $\mathcal{F}_P$ is correct for all P, that is, if $\mathbb{C}_0 \xrightarrow{p}_t \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ then $E, \sigma \vdash \mathcal{F}_P$. For the initial configuration, the property is obvious since $\mathcal{F}_P = \emptyset$. For the other configurations, we conclude by (S0).

We show the invariant **S1**: $\mathcal{F}_{C[Q_0]} = \emptyset$ and if Q is an input process and P is the input or output process just above Q, then $\mathcal{F}_Q \subseteq \mathcal{F}_P$. This property is obvious after collectFacts since $\mathcal{F}_Q = \emptyset$, and it is preserved by all updates to $\mathcal{F}_Q$ (provided the consequences of defined facts are not added in Q before they are added in P, which we can easily satisfy).

We prove **S2**: if $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$ and for all $(\sigma, Q) \in \mathcal{Q}$, $E, \sigma \vdash \mathcal{F}_Q$, then for all $(\sigma, Q) \in \mathcal{Q}'$, $E', \sigma \vdash \mathcal{F}_Q$. The proof is easy by cases on the derivation of $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E', \mathcal{Q}', \mathcal{C}'$, using (S1). Therefore, we have **S2'**: if $E', \mathcal{Q}', \mathcal{C}' = \mathrm{reduce}(E, \mathcal{Q}, \mathcal{C})$ and for all $(\sigma, Q) \in \mathcal{Q}$, $E, \sigma \vdash \mathcal{F}_Q$, then for all $(\sigma, Q) \in \mathcal{Q}'$, $E', \sigma \vdash \mathcal{F}_Q$.

Next, we prove that if $\mathcal{F}_P$ is correct for all P, then $\mathcal{F}'_P$ obtained by

$$\mathcal{F}'_P = \mathcal{F}_P \cup \mathcal{F}_{P'} \text{ if } P \text{ is immediately under } P'$$

is correct for all P. We show that, if $\mathbb{C}_0 \xrightarrow{p}_t \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ then for all $(\sigma', P') \in \{(\sigma, P)\} \uplus \mathcal{Q}$, $E, \sigma' \vdash \mathcal{F}'_{P'}$. The proof proceeds by induction on the length of the trace. For the initial configuration, $\mathcal{F}_{C[Q_0]} = \emptyset$ by (S1), so $\emptyset, \emptyset \vdash \mathcal{F}_{C[Q_0]}$, and $\emptyset, \emptyset \vdash \mathcal{F}_{\overline{start}\langle\rangle}$, so the property follows immediately from (S2'). For the inductive step, if the last reduction of the trace is (Output), we have $E_1, (\sigma_1, P_1), \{(\sigma, Q)\} \uplus \mathcal{Q}_1, \mathcal{C}_1 \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ with $P_1 = \overline{c[M_1, \dots, M_l]}\langle N_1, \dots, N_k\rangle.Q_1$, $Q = c[M'_1, \dots, M'_l](x_1[\widetilde{i}] : T_1, \dots, x_k[\widetilde{i}] : T_k).P$, $E = E_1[x_1[\sigma\widetilde{i}] \mapsto \dots, \dots, x_k[\sigma\widetilde{i}] \mapsto \dots]$, $\mathcal{Q} = \mathcal{Q}_1 \uplus \mathcal{Q}_2$, and $E_1, \mathcal{Q}_2, \mathcal{C} = \mathrm{reduce}(E_1, \{(\sigma_1, Q_1)\}, \mathcal{C}_1)$. If P is in C, $\mathcal{F}'_P = \emptyset$, so $E, \sigma \vdash \mathcal{F}'_P$. Otherwise, $E, \sigma \vdash \mathcal{F}'_Q$ by induction hypothesis. Moreover $E, \sigma \vdash \mathcal{F}_P$ since $\mathcal{F}_P$ is correct for all P, so

$E, \sigma \vdash \mathcal{F}'_P$ since $\mathcal{F}'_P = \mathcal{F}_Q \cup \mathcal{F}_P \subseteq \mathcal{F}'_Q \cup \mathcal{F}_P$. By induction hypothesis, for all $(\sigma', Q') \in \mathcal{Q}_1$, $E_1, \sigma' \vdash \mathcal{F}'_{Q'}$. Also by induction hypothesis, $E_1, \sigma_1 \vdash \mathcal{F}'_{P_1}$, so $E_1, \sigma_1 \vdash \mathcal{F}'_{Q_1} \subseteq \mathcal{F}'_{P_1}$ by (S1). By (S2'), for all $(\sigma', Q') \in \mathcal{Q}_2$, $E_1, \sigma' \vdash \mathcal{F}'_{Q'}$. So for all $(\sigma', Q') \in \mathcal{Q} = \mathcal{Q}_1 \uplus \mathcal{Q}_2$, $E_1, \sigma' \vdash \mathcal{F}'_{Q'}$, so $E, \sigma' \vdash \mathcal{F}'_{Q'}$ since E is an extension of E_1. If the last reduction is not (Output), it is of the form $E_1, (\sigma, P'), \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ where E is an extension of E_1. By induction hypothesis, for all $(\sigma', Q') \in \mathcal{Q}$, $E_1, \sigma' \vdash \mathcal{F}'_{Q'}$, so for all $(\sigma', Q') \in \mathcal{Q}$, $E, \sigma', \vdash \mathcal{F}'_{Q'}$. Since $\mathcal{F}_P$ is correct for all P, $E, \sigma \vdash \mathcal{F}_P$ and $E_1, \sigma \vdash \mathcal{F}_{P'}$, so $E, \sigma \vdash \mathcal{F}_{P'}$, so $E, \sigma \vdash \mathcal{F}'_P = \mathcal{F}_P \cup \mathcal{F}_{P'}$.

We show **S3**: if $E, (\sigma, P), \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t \dots \xrightarrow{p'}_{t'} E', (\sigma', P'), \mathcal{Q}', \mathcal{C}'$ where P' is an output and no process before P' in this trace is an output, then $E', \sigma' \vdash \mathcal{F}^{\mathrm{Fut}}_P$. Since no process before P' in this trace is an output, this trace does not contain the reduction rule (Output). The proof proceeds by induction on P. If P is an output, the result is obvious since $\mathcal{F}^{\mathrm{Fut}}_P = \mathrm{collectFacts}(P) = \emptyset$. Otherwise, let $P_1, \dots, P_m$ be the immediate subprocesses of P. We have $E, (\sigma, P), \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E_1, (\sigma, P_j), \mathcal{Q}, \mathcal{C}$ for some extension E_1 of E and some $j \in \{1, \dots, m\}$. Moreover, by definition of collectFacts, $\mathcal{F}^{\mathrm{Fut}}_P = \mathrm{collectFacts}(P) = \bigcap_{j=1}^m (\mathcal{F}_{P_j} \cup \mathcal{F}^{\mathrm{Fut}}_{P_j})$, where the value of $\mathcal{F}_{P_j}$ is considered immediately after calling collectFacts. By (S0), $E_1, \sigma \vdash \mathcal{F}_{P_j}$, so $E', \sigma' \vdash \mathcal{F}_{P_j}$ since E' is an extension of E_1 and $\sigma' = \sigma$ since no (Output) reduction occurs in this trace. By induction hypothesis, $E', \sigma' \vdash \mathcal{F}^{\mathrm{Fut}}_{P_j}$, so $E', \sigma' \vdash \mathcal{F}_{P_j} \cup \mathcal{F}^{\mathrm{Fut}}_{P_j}$ for some $j \in \{1, \dots, m\}$. Therefore, $E', \sigma' \vdash \mathcal{F}^{\mathrm{Fut}}_P$.

We now show that if $\mathcal{F}_P$ is correct for all P, and $\mathcal{F}'_P$ is obtained by

$$\mathcal{F}'_P = \mathcal{F}_P \cup \left(\bigcap_{(x[i_1,\dots,i_m],P') \in \mathcal{D}} \begin{cases} \sigma'(\mathcal{F}_{P'} \cup (\mathcal{F}^{\mathrm{Fut}}_{P'} \cap \mathcal{F}_P)) & \text{if } P \text{ is under } P' \\ \sigma'(\mathcal{F}_{P'} \cup \mathcal{F}^{\mathrm{Fut}}_{P'}) & \text{otherwise} \end{cases} \right)$$

where $\sigma' = \{M_1/i_1, \dots, M_m/i_m\}$, when $\mathsf{defined}(M) \in \mathcal{F}_P$ and $x[M_1, \dots, M_m]$ is a subterm of M, and $\mathcal{F}'_P = \mathcal{F}_P$ otherwise, then $\mathcal{F}'_P$ is also correct for all P. We assume that $\mathbb{C}_0 \xrightarrow{p}_t \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ and show that $E, \sigma \vdash \mathcal{F}'_P$. Since $\mathcal{F}_P$ is correct for all P, $E, \sigma \vdash \mathcal{F}_P$. Since $E, \sigma \vdash \mathsf{defined}(M)$, $E, \sigma, M_j \Downarrow a_j$ for all $j \leq m$ and $x[a_1, \dots, a_m] \in \mathrm{Dom}(E)$. Therefore, some definition of $x[a_1, \dots, a_m]$ has been executed in the considered trace. Next, we show that, for some $(x[i_1, \dots, i_m], P') \in \mathcal{D}$, we have $E, \sigma_1 \vdash \mathcal{F}_{P'}$; if P is under P' then $E, \sigma_1 \vdash \mathcal{F}^{\mathrm{Fut}}_{P'} \cap \mathcal{F}_P$; and if P is not under P' then $E, \sigma_1 \vdash \mathcal{F}^{\mathrm{Fut}}_{P'}$, where $\sigma_1(i_1) = a_1, \dots, \sigma_1(i_m) = a_m$. The desired result follows. Let $E_1, (\sigma_1, P_1), \mathcal{Q}_1, \mathcal{C}_1 \xrightarrow{p_1}_{t_1} E_2, (\sigma_1, P_2), \mathcal{Q}_2, \mathcal{C}_2$ be the reduction that defines $x[a_1, \dots, a_m]$ in the considered trace. We have $E_2, \sigma_1 \vdash \mathcal{F}_{P_2}$ since $\mathcal{F}_P$ is correct for all P. So $E, \sigma_1 \vdash \mathcal{F}_{P_2}$ since E is an extension of E_2 so all facts that hold in E_2 also hold in E. We have $(x[i_1, \dots, i_m], P_2) \in \mathcal{D}$. If P is not under P_2, the trace $E_2, (\sigma_1, P_2), \mathcal{Q}_2, \mathcal{C}_2 \xrightarrow{p_2}_{t_2} \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ must execute an output, so by (S3), $E_3, \sigma_3 \vdash \mathcal{F}^{\mathrm{Fut}}_{P_2}$ where the configuration in which the first output after $E_2, (\sigma_1, P_2), \mathcal{Q}_2, \mathcal{C}_2$ is executed is $E_3, (\sigma_3, P_3), \mathcal{Q}_3, \mathcal{C}_3$, so $E, \sigma_1 \vdash \mathcal{F}^{\mathrm{Fut}}_{P_2}$. (We have $\sigma_3 = \sigma_1$, since the substitution σ is changed only when executing a communication.) If P is under P_2, two cases can happen. Either the trace $E_2, (\sigma_1, P_2), \mathcal{Q}_2, \mathcal{C}_2 \xrightarrow{p_2}_{t_2} \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ executes an output, and we have $E, \sigma_1 \vdash \mathcal{F}^{\mathrm{Fut}}_{P_2}$ as above, or $E_2, (\sigma_1, P_2), \mathcal{Q}_2, \mathcal{C}_2 \xrightarrow{p_2}_{t_2} \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ executes no output, so $\sigma = \sigma_1$. (The substitution σ is changed only when executing a communication.) Since $\mathcal{F}_P$ is correct for all P, $E, \sigma \vdash \mathcal{F}_P$, hence $E, \sigma_1 \vdash \mathcal{F}_P$. Then, in both cases, $E, \sigma_1 \vdash \mathcal{F}^{\mathrm{Fut}}_{P_2} \cap \mathcal{F}_P$.

Next, we show **S4**: if $\mathbb{C}_0 \xrightarrow{p}_t \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ then $E, \sigma \vdash \mathcal{F}^{\mathrm{ElseFind}}_P$. The proof proceeds by induction on the length of the trace. For the initial configuration, the result is obvious since $\mathcal{F}^{\mathrm{ElseFind}}_P = \emptyset$. For the inductive step, if the reduced process is in C, the result is obvious since $\mathcal{F}^{\mathrm{ElseFind}}_P = \emptyset$. Otherwise, we proceed by cases on the last reduction of the trace. In the (Output) case, the result is obvious since $\mathcal{F}^{\mathrm{ElseFind}}_P = \emptyset$. In the (New), (Let), and (Find1) cases, σ is unchanged, E is extended with definitions for some variables, and *elsefind* facts that claim that these variables are not defined are removed from $\mathcal{F}^{\mathrm{ElseFind}}_P$, so we still have $E, \sigma \vdash \mathcal{F}^{\mathrm{ElseFind}}_P$. In the (Find2) case for $P' = \mathsf{find}\ (\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ $\mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P$, E and σ are unchanged and since (Find2) is executed, $\forall j \leq m$, $\forall a_1 \in [1, I_\eta(n_{j1})]$, …, $\forall a_{m_j} \in [1, I_\eta(n_{jm_j})]$, $E[u_{j1}[\widetilde{\sigma i}] \mapsto a_1, \dots, u_{jm_j}[\widetilde{\sigma i}] \mapsto a_{m_j}], \sigma, (\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j) \Downarrow$ false. $\mathcal{F}^{\mathrm{ElseFind}}_P = \mathcal{F}^{\mathrm{ElseFind}}_{P'} \cup \{\mathit{elsefind}((u_1 \leq n_{j1}, \dots, u_{m_j} \leq n_{jm_j}), \sigma_j(M_{j1}, \dots, M_{jl_j}), \sigma_j M_j) \mid j \in \{1, \dots, m\}\}$ where $\sigma_j = \{u_1/u_{j1}[\widetilde{i}], \dots, u_{m_j}/u_{jm_j}[\widetilde{i}]\}$. By induction hypothesis $E, \sigma \vdash \mathcal{F}^{\mathrm{ElseFind}}_{P'}$. Moreover, $E, \sigma \vdash \mathit{elsefind}((u_1 \leq n_{j1}, \dots, u_{m_j} \leq n_{jm_j}), \sigma_j(M_{j1}, \dots, M_{jl_j}), \sigma_j M_j)$ for $j \in \{1, \dots, m\}$, so $E, \sigma \vdash \mathcal{F}^{\mathrm{ElseFind}}_P$.

We now show that if $\mathcal{F}_P$ is correct for all P, then $\mathcal{F}'_P$ obtained by

$$\begin{aligned}\mathcal{F}'_P = \mathcal{F}_P \cup \{\neg\sigma' M \mid\ & \mathit{elsefind}((u_1 \leq n_1, \dots, u_m \leq n_m), \\ & (M_1, \dots, M_l), M) \in \mathcal{F}^{\mathrm{ElseFind}}_P, \mathrm{Dom}(\sigma') = \{u_1, \dots, u_m\}, \\ & \text{for each } j \in \{1, \dots, l\}, \sigma' M_j \text{ is a subterm of } M'_j \text{ and} \\ & \mathsf{defined}(M'_j) \in \mathcal{F}_P\}\end{aligned}$$

is also correct for all P. Assuming that $\mathbb{C}_0 \xrightarrow{p}_t \dots \xrightarrow{p'}_{t'} E, (\sigma, P), \mathcal{Q}, \mathcal{C}$, we show that $E, \sigma \vdash \mathcal{F}'_P$. Since $\mathcal{F}_P$ is correct for all P, $E, \sigma \vdash \mathcal{F}_P$. By (S4), $E, \sigma \vdash \mathcal{F}^{\mathrm{ElseFind}}_P$. Assume $\mathit{elsefind}((u_1 \leq n_1, \dots, u_m \leq n_m), (M_1, \dots, M_l), M) \in \mathcal{F}^{\mathrm{ElseFind}}_P$ and for each $j \in \{1, \dots, l\}$, $\sigma' M_j$ is a subterm of M'_j and $\mathsf{defined}(M'_j) \in \mathcal{F}_P$. Let a_k be such that $E, \sigma, \sigma' u_k \Downarrow a_k$ for each $k \in \{1, \dots, m\}$. Let $\sigma'' = \sigma[u_1 \mapsto a_1, \dots, u_m \mapsto a_m]$. Since $E, \sigma \vdash \mathsf{defined}(M'_j)$, we have $E, \sigma, M'_j \Downarrow a'_j$ for some a'_j so $E, \sigma, \sigma' M_j \Downarrow a''_j$ for some a''_j, so $E, \sigma'', M_j \Downarrow a''_j$. (This is proved by induction on M_j.) By definition of *elsefind* facts, $E, \sigma'', (\mathsf{defined}(M_1, \dots, M_l) \wedge M) \Downarrow$ false so $E, \sigma'', M \Downarrow$ false, that is, $E, \sigma, \sigma' M \Downarrow$ false, so $E, \sigma \vdash \neg\sigma' M$. So $E, \sigma \vdash \mathcal{F}'_P$.

Therefore, we conclude that at the end of the computation, $\mathcal{F}_P$ is correct for all P.

Correctness of the local dependency analysis. As above in the correctness of the collection of true facts, we denote by P

an occurrence of a process, so that we can distinguish identical subprocesses that occur at several occurrences in a process.

We first show the soundness of the local dependency analysis ignoring modifications in the game performed by $\mathrm{depAnal}$. Then we will show the soundness of the game modifications, that is, that these modifications change the behavior of the game only with negligible probability. Since the game modifications do not change the part of the computation of depend and indep performed before the modification, the $\mathrm{depAnal}$ procedure is equivalent to performing a full dependency analysis without game modification, performing game modification, redoing the whole dependency analysis analysis on the modified game, and so on, until a fixpoint is reached. Therefore, the separate proof of the dependency analysis and the game modifications outlined above is sufficient to prove the correctness of the $\mathrm{depAnal}$ procedure.

We have **S5**: if y is defined only by restrictions and $y \neq x$, then there exists no M such that $(y, M) \in \mathrm{depend}_P(x)$. This property is obvious since the only case in which an element (y, M) is added in $\mathrm{depend}_P(x)$ is in the assignment $\mathsf{let}\ y[\widetilde{i}] : T = M'$ in P', so such an addition cannot happen when y is defined only by restrictions.

For each σ, depend, indep, we define an equivalence relation $\sim_{\sigma,\mathrm{depend},\mathrm{indep}}$ on environments by $E \sim_{\sigma,\mathrm{depend},\mathrm{indep}} E'$ if and only if

- for all $M \in \mathrm{indep}$, for all b, $E, \sigma, M \Downarrow b$ if and only if $E', \sigma, M \Downarrow b$;
- if $\mathrm{depend} \neq \top$, then for all $z[\widetilde{a}]$ such that $z[\widetilde{a}] \neq x[\widetilde{\sigma i}]$ and there exists no $(y, M) \in \mathrm{depend}$ such that $z[\widetilde{a}] = y[\widetilde{\sigma i}]$, $E(z[\widetilde{a}])$ is defined if and only if $E'(z[\widetilde{a}])$ is defined and when they are defined, $E(z[\widetilde{a}]) = E'(z[\widetilde{a}])$ ($\widetilde{i}$ denotes the current replication indices at definition of x);
- and for all y such that $y \neq x$ and y is defined only by restrictions, for all $\widetilde{a}$, $E(y[\widetilde{a}])$ is defined if and only if $E'(y[\widetilde{a}])$ is defined and when they are defined, $E(y[\widetilde{a}]) = E'(y[\widetilde{a}])$.

When $E \sim_{\sigma,\mathrm{depend},\mathrm{indep}} E'$, the environments E and E' differ only by variables that depend on $x[\widetilde{\sigma i}]$, according to the information contained in depend and indep. That is, terms in indep have the same value in E and E' (first item); when $\mathrm{depend} \neq \top$, variables not in depend have the same value in E and E' (second item); variables defined only by restrictions have the same value in E and E' (third item). We abbreviate $\sim_{\sigma,\mathrm{depend}_P(x),\mathrm{indep}_P(x)}$ by $\sim_{\sigma,P}$.

We show **S6**: if M' does not depend on x at P and $E \sim_{\sigma,P} E'$, then $E, \sigma, M' \Downarrow b$ if and only if $E', \sigma, M' \Downarrow b$. This property expresses the correctness of the definition of "M' does not depend on x at P". We prove that if $E, \sigma, M' \Downarrow b$ then $E', \sigma, M' \Downarrow b$, by induction on the derivation that M' does not depend on x at P. The converse follows immediately by swapping the roles of E and E'.

- Case $M' = f(M'_1, \dots, M'_m)$ and for all $j \leq m$, M'_j does not depend on x at P. Since $E, \sigma, M' \Downarrow b$, $E, \sigma, M'_j \Downarrow b_j$ and $I_\eta(f)(b_1, \dots, b_m) = b$ for some $b_1, \dots, b_m$. Hence by induction hypothesis, $E', \sigma, M'_j \Downarrow b_j$, so $E', \sigma, M' \Downarrow b$.
- Case $M' \in \mathrm{indep}_P(x)$. The result comes from the definition of $\sim_{\sigma,P}$.
- Case M' is a replication index. We have $E, \sigma, M' \Downarrow \sigma M'$ and $E', \sigma, M' \Downarrow \sigma M'$, so the result holds.
- Case $M' = y[M'_1, \dots, M'_m]$, $M'_1, \dots, M'_m$ do not depend on x at P', $y \neq x$, and either y is defined only by restrictions or $\mathrm{depend}_P(x) \neq \top$ and $y \neq y'$ for all $(y', M'') \in \mathrm{depend}_P(x)$. Since $E, \sigma, M' \Downarrow b$, $E, \sigma, M'_j \Downarrow b_j$ and $E(y[b_1, \dots, b_k]) = b$ for some $b_1, \dots, b_k$. Hence by induction hypothesis, $E', \sigma, M'_j \Downarrow b_j$. By definition of $\sim_{\sigma,P}$, $E'(y[b_1, \dots, b_k]) = E(y[b_1, \dots, b_k]) = b$, so $E', \sigma, M' \Downarrow b$.

Let us consider the following property **L0**:

1. If $\Pr[\mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C}] > 0$, $\mathrm{depend}_P(x) \neq \top$, and $(y, M) \in \mathrm{depend}_P(x)$, then $E, \sigma, M \Downarrow E(y[\widetilde{\sigma i}])$ where $\widetilde{i}$ denotes the current replication indices at P;
2. If $\Pr[\mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C}] > 0$ and $M \in \mathrm{indep}_P(x)$, then $E, \sigma, M \Downarrow a$ for some a;
3. For each $b \in I_\eta(T)$, for each σ, for each E_0, $\Pr[\exists E, \exists \mathcal{Q}, \exists \mathcal{C}, \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E \sim_{\sigma,P} E_0 \wedge E(x[\widetilde{\sigma i}]) = b] \leq \frac{1}{|I_\eta(T)|} \Pr[\exists E, \exists \mathcal{Q}, \exists \mathcal{C}, \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E \sim_{\sigma,P} E_0]$ where $\widetilde{i}$ denotes the current replication indices at the definition of x.

We will show that if $\mathrm{depend}_P(x) \neq \top$, then (L0) holds at P. This property expresses the correctness of the local dependency analysis at P, when $\mathrm{depend}_P(x) \neq \top$. (We will consider the general case below, Property L1.) Item 1 says that, when $(y, M) \in \mathrm{depend}_P(x)$, M evaluates to the contents of y. Item 2 says that, when $M \in \mathrm{indep}_P(x)$, the value of M is always defined at P. Finally, the last item is the most important one: it expresses the independence properties. Essentially, the traces that differ by the value of $x[\widetilde{\sigma i}]$ all have the same probability, and differ only by the values of variables that depend on $x[\widetilde{\sigma i}]$, collected in $\mathrm{depend}_P(x)$, so their environments are related by $\sim_{\sigma,P}$. When the value of $x[\widetilde{\sigma i}]$ is fixed to b, the probability of reaching an environment related to E_0 by $\sim_{\sigma,P}$ is then $\frac{1}{|I_\eta(T)|}$ times the probability of reaching such an environment for any value of $x[\widetilde{\sigma i}]$.

We first show **S7**: if (L0) holds at P with indep instead of $\mathrm{indep}_P(x)$, for all E, σ such that $\Pr[\mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C}] > 0$, $E, \sigma, M' \Downarrow a$ for some a, and M' does not depend on x at P with indep instead of $\mathrm{indep}_P(x)$, then (L0) also holds at P with $\mathrm{indep} \cup \{M'\}$ instead of $\mathrm{indep}_P(x)$. Essentially, this property means that M' can be added to $\mathrm{indep}_P(x)$ when M' does not depend on x at P. Items 1 and 2 of (L0) hold by hypothesis. If $E \sim_{\sigma,\mathrm{depend}_P(x),\mathrm{indep}} E'$, by (S6), $E, \sigma, M' \Downarrow b$ if and only if $E', \sigma, M' \Downarrow b$, so $E \sim_{\sigma,\mathrm{depend}_P(x),\mathrm{indep}\cup\{M'\}} E'$. Conversely, we have obviously: if $E \sim_{\sigma,\mathrm{depend}_P(x),\mathrm{indep}\cup\{M'\}} E'$, then $E \sim_{\sigma,\mathrm{depend}_P(x),\mathrm{indep}} E'$, so $\sim_{\sigma,\mathrm{depend}_P(x),\mathrm{indep}} = \sim_{\sigma,\mathrm{depend}_P(x),\mathrm{indep}\cup\{M'\}}$. This proves Item 3 of (L0), and concludes the proof of (L0).

Next, we prove **S8**: if $\mathrm{depend}_P(x) \neq \top$ then (L0) holds at P, by decreasing induction on the process P. The only cases in which $\mathrm{depend}_P(x) \neq \top$ are as follows:

- P occurs in $P' = \mathsf{new}\ x[\widetilde{i}] : T; P$ where T is a large type. We have $\mathrm{depend}_P(x) = \emptyset$ and $\mathrm{indep}_P(x) = \bigcup_{\mathsf{defined}(M) \in \mathcal{F}_{P'}} \mathrm{subterms}(M)$. Item 1 of (L0) holds trivially. For all traces of non-zero probability that reach P, the last reduction reduces P' by (New), so these traces are all of the form $\mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \to E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ where $E = E'[x[\sigma\widetilde{i}] \mapsto a']$ for some $a' \in I_\eta(T)$. Since $\mathcal{F}_P$ is correct for all P, $E', \sigma \vdash \mathcal{F}_{P'}$, so for all $M' \in \mathrm{subterms}(M)$ such that $\mathsf{defined}(M) \in \mathcal{F}_{P'}$, $E', \sigma, M' \Downarrow a$ for some a, hence $E, \sigma, M' \Downarrow a$ since E is an extension of E', which proves Item 2 of (L0). By the semantic rule (New), for all $b \in I_\eta(T)$, $\Pr[\exists E, \exists \mathcal{Q}, \exists \mathcal{C}, \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E \sim_{\sigma,P} E_0 \wedge E(x[\sigma\widetilde{i}]) = b] = \frac{1}{|I_\eta(T)|} \Pr[\exists E, \exists \mathcal{Q}, \exists \mathcal{C}, \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E \sim_{\sigma,P} E_0]$ since the condition $E \sim_{\sigma,P} E_0$ does not use the value of $E(x[\sigma\widetilde{i}])$. (The first item of $E \sim_{\sigma,P} E_0$ does not use the value of $E(x[\sigma\widetilde{i}])$ because the elements of $\mathrm{indep}_P(x)$ are all defined in E' and $E'(x[\sigma\widetilde{i}])$ is not defined. The other two items never use $E(x[\sigma\widetilde{i}])$.) Therefore, we obtain Item 3 of (L0).

- P occurs in $P' = \mathsf{new}\ y[\widetilde{i}] : T'; P$ with $y \neq x$. We have $\mathrm{depend}_P(x) = \mathrm{depend}_{P'}(x)$ and $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x) \cup \{y[\widetilde{i}]\}$. For all traces of non-zero probability that reach P, the last reduction reduces P' by (New), so these traces are all of the form $\mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \to E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ where $E = E'[y[\sigma\widetilde{i}] \mapsto a']$ for some $a' \in I_\eta(T')$. Item 1 of (L0) comes from the induction hypothesis (at P') and the fact that E is an extension of E'. Item 2 of (L0) comes from the induction hypothesis (at P'), the fact that E is an extension of E', and the fact that $E(y[\sigma\widetilde{i}])$ is defined. Let $E_0' = E_{0|\overline{y[\sigma\widetilde{i}]}}$ be the environment E_0 restricted to the variables defined at P'.

$$
\begin{aligned}
&\Pr\begin{bmatrix}\exists(E, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \\ \wedge E \sim_{\sigma,P} E_0 \wedge E(x[\sigma\widetilde{i}]) = b\end{bmatrix} \\
&= \frac{1}{|I_\eta(T')|} \Pr\begin{bmatrix}\exists(E', \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \\ \wedge E' \sim_{\sigma,P'} E_0' \wedge E(x[\sigma\widetilde{i}]) = b\end{bmatrix} \\
&\leq \frac{1}{|I_\eta(T')|} \frac{1}{|I_\eta(T)|} \Pr\begin{bmatrix}\exists(E', \mathcal{Q}, \mathcal{C}), \\ \mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \\ \wedge E' \sim_{\sigma,P'} E_0'\end{bmatrix} \\
&\leq \frac{1}{|I_\eta(T)|} \Pr\begin{bmatrix}\exists(E, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \\ \wedge E \sim_{\sigma,P} E_0\end{bmatrix}
\end{aligned}
$$

The first step is by the semantic rule (New), the second step by induction hypothesis, and the last step by the semantic rule (New) again. Therefore, we obtain Item 3 of (L0).

- P occurs in $P' = \mathsf{let}\ y[\widetilde{i}] : T' = M\ \mathsf{in}\ P$ with $y \neq x$. For all traces of non-zero probability that reach P, the last reduction reduces P' by (Let), so these traces are all of the form $\mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \to E, (\sigma, P), \mathcal{Q}, \mathcal{C}$ where $E', \sigma, M \Downarrow a'$ and $E = E'[y[\sigma\widetilde{i}] \mapsto a']$. Let $E_0' = E_{0|\overline{y[\sigma\widetilde{i}]}}$.

  If M does not depend on x at P', we have $\mathrm{depend}_P(x) = \mathrm{depend}_{P'}(x)$ and $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x) \cup \{y[\widetilde{i}]\}$. In this case, by (S6), $E', \sigma, M \Downarrow a'$ if and only if $E_0', \sigma, M \Downarrow a'$ (where $E' \sim_{\sigma,P'} E_0'$ are environments at P'). We can then show that (L0) holds at P using the induction hypothesis. (We have $E \sim_{\sigma,P} E_0$ if and only if $E' \sim_{\sigma,P'} E_0'$ and $E_0 = E_0'[y[\sigma\widetilde{i}] \mapsto a']$.)

  Otherwise, we have $\mathrm{depend}_P(x) = \mathrm{depend}_{P'}(x) \cup \{(y, M\mathrm{depend}_{P'}(x))\}$ and $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x)$. By induction hypothesis, for all $(y', M') \in \mathrm{depend}_{P'}(x)$, $E', \sigma, M' \Downarrow E'(y'[\sigma\widetilde{i}])$, so $E, \sigma, M' \Downarrow E(y'[\sigma\widetilde{i}])$, hence $E, \sigma, M\mathrm{depend}_{P'}(x) \Downarrow a' = E(y[\sigma\widetilde{i}])$, so we obtain Item 1 of (L0). Item 2 of (L0) follows immediately from the induction hypothesis. Item 3 of (L0) also follows from the induction hypothesis. (We have $E \sim_{\sigma,P} E_0$ if and only if $E' \sim_{\sigma,P'} E_0'$.)

- P occurs in $P' = \mathsf{find}\ (\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P''$, P is either P'' or P_j for some $j \leq m$, and for all j, k, M_{jk} and M_j' do not depend on x at P'. We have $\mathrm{depend}_P(x) = \mathrm{depend}_{P'}(x)$, $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x)$ if $P = P''$, and $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x) \cup \{M' \mid M' \in \mathrm{subterms}(M)$ for some $\mathsf{defined}(M) \in \mathcal{F}_{P_j}$, M' does not depend on x at $P'\}$ if $P = P_j$. By (S6), we can show that the same branch of the find is taken with the same probability for all E such that $E \sim_{\sigma,P'} E_0$ for the same E_0. Using the induction hypothesis, we can then show that (L0) holds at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$. This concludes the proof when $P = P''$. When $P = P_j$, let $M_1'', \ldots, M_l''$ be the terms M' such that $M' \in \mathrm{subterms}(M)$ for some $\mathsf{defined}(M) \in \mathcal{F}_{P_j}$ and M' does not depend on x at P'. Since $\mathcal{F}_{P_j}$ is correct and $\Pr[\mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C}] = p > 0$ then $E, \sigma \vdash \mathcal{F}_{P_j}$, so $E, \sigma, M_k'' \Downarrow a$ for some a. The term M_k'' does not depend on x at P with $\mathrm{indep}_{P'}(x) \cup \{M_1'', \ldots, M_{k-1}''\}$ instead of $\mathrm{indep}_P(x)$. By (S7) applied at P with $\mathrm{indep}_{P'}(x) \cup \{M_1'', \ldots, M_{k-1}''\}$ instead of $\mathrm{indep}_P(x)$, if (L0) holds at P with $\mathrm{indep}_{P'}(x) \cup \{M_1'', \ldots, M_{k-1}''\}$, then (L0) holds at P with $\mathrm{indep}_{P'}(x) \cup \{M_1'', \ldots, M_k''\}$. So (L0) holds at P with $\mathrm{indep}_{P'}(x) \cup \{M_1'', \ldots, M_l''\} = \mathrm{indep}_P(x)$.

For each σ, P, we define a special semantics of processes. This semantics executes the process $C[Q_0]$ normally until it reaches a configuration $E', (\sigma', P'), \mathcal{Q}, \mathcal{C}$ such that P' is the smallest superprocess of P such that $\mathrm{depend}_{P'}(x) \neq \top$ and $\sigma'(i) = \sigma(i)$ for all $i \in \mathrm{Dom}(\sigma')$. After reaching this configuration, it executes restrictions for all variables defined only by restrictions in $C[Q_0]$ that have not been assigned yet and executes the not-executed-yet restrictions and the assignments $P_1 = \mathsf{let}\ y : T = M\ \mathsf{in}\ P_2$ such that M does not depend on x at P_1 between P' and P. In the second part of the trace, a configuration is only $E'', (\sigma'', P'')$; σ'' is always set to be σ restricted to the current replication indices at P''. We write $\mathbb{C}_0 \to'^* E, (\sigma, P)$ to designate a trace in this special semantics. (When $\mathrm{depend}_P(x) \neq \top$, this semantics executes the process normally, and finally executes restrictions for all variables defined only by restrictions that have not been assigned yet.)

We will show the following property **L1**:

1. If $\Pr[\mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C}] > 0$, $\mathrm{depend}_P(x) \neq \top$, and $(y, M) \in \mathrm{depend}_P(x)$, then $E, \sigma, M \Downarrow E(y[\sigma\widetilde{i}])$ where $\widetilde{i}$

denotes the current replication indices at P;

2. If $\Pr[\mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C}] > 0$ and $M \in \mathrm{indep}_P(x)$, then $E, \sigma, M \Downarrow a$ for some a;

3. For each $b \in I_\eta(T)$, for each σ, for each E_0, $\Pr[\exists(E, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E \sim_{\sigma,P} E_0 \wedge E(x[\widetilde{\sigma i}]) = b] \leq \frac{1}{|I_\eta(T)|} \Pr[\exists E_1, \mathbb{C}_0 {\to'}^* E_1, (\sigma, P) \wedge E_{1|\mathrm{Dom}(E_0)} \sim_{\sigma,P} E_0]$ where $\widetilde{i}$ denotes the current replication indices at the definition of x.

Property (L1) expresses the correctness of the local dependency analysis at P. It differs from (L0) by the use of the special semantics $\to'$ in Item 3. This semantics is necessary when $\mathrm{depend}_P(x) = \top$, because in that case the control-flow may also depend on the value of $x[\widetilde{\sigma i}]$, so P may not be reachable for certain values of $x[\widetilde{\sigma i}]$, which breaks the inequality between probabilities of (L0), Item 3. In contrast, the special semantics $\to'$ computes $E_1, (\sigma, P)$ without taking into account the control-flow, so this problem is avoided.

Property (S7) also holds for (L1), with the same proof as for (L0).

We show **S9**: if (L0) holds at P, then (L1) holds at P. Let E_1 be E extended with values for all variables defined only by restrictions. If $E \sim_{\sigma,P} E_0$, the variables defined only by restrictions are defined for the same indices in E and in E_0, so $E_{1|\mathrm{Dom}(E_0)} = E$, hence $E_{1|\mathrm{Dom}(E_0)} \sim_{\sigma,P} E_0$. Therefore, $\Pr[\exists(E, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E \sim_{\sigma,P} E_0] \leq \Pr[\exists E_1, \mathbb{C}_0 {\to'}^* E_1, (\sigma, P) \wedge E_{1|\mathrm{Dom}(E_0)} \sim_{\sigma,P} E_0]$, which proves (L1).

We show **S9'**: if (L0) holds at P, then (L1) holds at P with $\sim_{\sigma,\top,\mathrm{indep}_P(x)}$ instead of $\sim_{\sigma,P}$. If $E \sim_{\sigma,P} E'$, then $E \sim_{\sigma,\top,\mathrm{indep}_P(x)} E'$. So each equivalence class of $\sim_{\sigma,\top,\mathrm{indep}_P(x)}$ is a union of equivalence classes of $\sim_{\sigma,P}$. So we obtain (L0) with $\sim_{\sigma,\top,\mathrm{indep}_P(x)}$ instead of $\sim_{\sigma,P}$ by adding probabilities. We conclude that (L1) holds at P with $\sim_{\sigma,\top,\mathrm{indep}_P(x)}$ instead of $\sim_{\sigma,P}$ using a proof similar to that of (S9).

We show **S10**: If P is an output process, P' is the smallest output process such that P is a strict subprocess of P', (L1) holds at P' with $\sim_{\sigma,\top,\mathrm{indep}_{P'}(x)}$ instead of $\sim_{\sigma,P'}$, and $\mathrm{depend}_P(x) = \top$, then (L1) holds at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$. The equivalence between environments for (L1) at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$ is also $\sim_{\sigma,\top,\mathrm{indep}_{P'}(x)}$, since $\mathrm{depend}_P(x) = \top$. Item 1 of (L1) holds trivially at P since $\mathrm{depend}_P(x) = \top$. For the proof of Item 3 of (L1), we let $p = \Pr[\exists(E, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E_0 \wedge E(x[\widetilde{\sigma i}]) = b]$.

- Case $P' = \mathsf{let}\ y[\widetilde{i}] : T' = M\ \mathsf{in}\ P$. In traces of non-zero probability that reach P, the last reduction of the trace reduces P' by (Let), so these traces are all of the form:

$$\mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \to E, (\sigma, P), \mathcal{Q}, \mathcal{C}$$

where $E', \sigma, M \Downarrow a$ and $E = E'[y[\widetilde{\sigma i}] \mapsto a]$ and the corresponding trace of $\to'$ is

$$\mathbb{C}_0 {\to'}^* E'_1, (\sigma, P') \to' E_1, (\sigma, P)$$

where $E'_1, \sigma, M \Downarrow a'$ and $E_1 = E'_1[y[\widetilde{\sigma i}] \mapsto a']$. Let $E'_0 = E_{0|\overline{y[\widetilde{\sigma i}]}}$ be the environment E_0 restricted to the variables defined at P'. For all $M' \in \mathrm{indep}_{P'}(x)$, $E_1, \sigma, M' \Downarrow b$ for some b since (L1) holds at P'. Then $E, \sigma, M' \Downarrow b$, so Item 2 of (L1) holds at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$. Since all elements of $\mathrm{indep}_{P'}(x)$ must be defined at P' (by Item 2 of (L1) at P'), $y[\widetilde{\sigma i}]$ is not defined at P', and y is not defined only by restrictions, the condition $E \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E_0$ in Item 3 of (L1) at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$ does not use the value of $E(y[\widetilde{\sigma i}])$, hence $E \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E_0$ if and only if $E' \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E'_0$, and $E_{1|\mathrm{Dom}(E_0)} \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E_0$ if and only if $E'_{1|\mathrm{Dom}(E'_0)} \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E'_0$, so the probabilities that occur in Item 3 of (L1) are the same for P' and for P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$. Therefore, Item 3 of (L1) holds at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$.

- Case $P' = \mathsf{new}\ y[\widetilde{i}] : T'; P$, where y is not defined only by restrictions. In traces of non-zero probability that reach P, the last reduction of the trace reduces P' by (New). This case is similar to the let case above.

- Case $P' = \mathsf{new}\ y[\widetilde{i}] : T'; P$, where y is defined only by restrictions. In traces of non-zero probability that reach P, the last reduction of the trace reduces P' by (New). Item 2 is proved as in the let case above. Let us consider Item 3. Let $E'_0 = E_{0|\overline{y[\widetilde{\sigma i}]}}$ be the environment E_0 restricted to the variables defined at P'. Let $\widetilde{i}'$ be the replication indices at the definition of x. ($\widetilde{i}'$ is a prefix of $\widetilde{i}$.)

$$\begin{aligned}
p &= \Pr\begin{bmatrix}\exists(E, E', \mathcal{Q}, \mathcal{C}),\\ \mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \to E, (\sigma, P), \mathcal{Q}, \mathcal{C}\\ \wedge E \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E_0 \wedge E(x[\widetilde{\sigma i}]) = b\end{bmatrix}\\
&= \frac{1}{|I_\eta(T')|} \Pr\begin{bmatrix}\exists(E', \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C}\\ \wedge E' \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E'_0\\ \wedge E'(x[\widetilde{\sigma i'}]) = b\end{bmatrix}\\
&\leq \frac{1}{|I_\eta(T')|}\frac{1}{|I_\eta(T)|} \Pr\begin{bmatrix}\exists E'_1, \mathbb{C}_0 {\to'}^* E'_1, (\sigma, P') \wedge\\ E'_{1|\mathrm{Dom}(E'_0)} \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E'_0\end{bmatrix}\\
&\leq \frac{1}{|I_\eta(T)|} \Pr\begin{bmatrix}\exists E_1, \mathbb{C}_0 {\to'}^* E_1, (\sigma, P)\\ \wedge E_{1|\mathrm{Dom}(E_0)} \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E_0\end{bmatrix}
\end{aligned}$$

The first step comes from the semantic rule (New), the second step from (L1) at P', the last step from the assignment of variables defined only by restrictions in the special $\to'$ semantics. (Note that $E'_1 = E_1$, but the condition $E'_{1|\mathrm{Dom}(E'_0)} \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E'_0$ does not use the value of $E'_1(y[\widetilde{\sigma i}])$.) This inequality proves (L1) at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$.

- Cases in which there is no assignment and no restriction between P and P'. Everything that is defined at P' is also defined at P, since the environment at P is an extension of the environment at P', so Item 2 of (L1) holds at P since it holds at P'. Let us now prove Item 3 of (L1). The final environment E' of the $\to'$ trace is the same for P and for P', so the right-hand side of the inequality is the same for P and for P'. The left-hand side decreases from P' to P, since all traces that reach P must first have reached P', so the inequality still holds.

From the previous results, we show that (L1) holds at all output processes P. The proof proceeds by decreasing induction on P. If $\mathrm{depend}_P(x) \neq \top$, we have the result using (S8) and (S9). Otherwise, let P' be the smallest output process such that P is a strict subprocess of P'. If $\mathrm{depend}_{P'}(x) \neq \top$, by (S8) and (S9'), (L1) holds at P' with $\sim_{\sigma,\top,\mathrm{indep}_{P'}(x)}$ instead of $\sim_{\sigma,P'}$. If $\mathrm{depend}_{P'}(x) = \top$, by induction hypothesis, (L1) holds at P', that is, (L1) holds at P' with $\sim_{\sigma,\top,\mathrm{indep}_{P'}(x)}$ instead of $\sim_{\sigma,P'}$. In both cases, by (S10), (L1) holds at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$. The only cases in which $\mathrm{indep}_{P'}(x) \neq \mathrm{indep}_P(x)$ are as follows:

- Case $P' = \mathsf{new}\ y[\widetilde{i}] : T'; P$, $y \neq x$, $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x) \cup \{y[\widetilde{i}]\}$. When y is defined only by restrictions, $y[\widetilde{i}]$ does not depend on x at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$, so, by (S7), (L1) holds at P. Otherwise, in traces of non-zero probability that reach P, the last reduction of the trace reduces P' by (New), so these traces are all of the form:

$$\mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \to E, (\sigma, P), \mathcal{Q}, \mathcal{C}$$

where $E = E'[y[\sigma\widetilde{i}] \mapsto a]$ for some $a \in I_\eta(T')$. So Item 2 of (L1) holds at P. Let $E'_0 = E_{0|\overline{y[\sigma\widetilde{i}]}}$. Let $\widetilde{i}'$ be the replication indices at the definition of x. ($\widetilde{i}'$ is a prefix of $\widetilde{i}$.) We prove Item 3 of (L1) as follows:

$$\begin{aligned}
p &= \Pr\begin{bmatrix}\exists(E, E', \mathcal{Q}, \mathcal{C}),\\ \mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C} \to E, (\sigma, P), \mathcal{Q}, \mathcal{C}\\ \wedge E \sim_{\sigma,\top,\mathrm{indep}_P(x)} E_0 \wedge E(x[\sigma\widetilde{i}']) = b\end{bmatrix}\\
&= \frac{1}{|I_\eta(T')|}\Pr\begin{bmatrix}\exists(E', \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E', (\sigma, P'), \mathcal{Q}, \mathcal{C}\\ \wedge E' \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E'_0\\ \wedge E'(x[\sigma\widetilde{i}']) = b\end{bmatrix}\\
&\leq \frac{1}{|I_\eta(T')|}\frac{1}{|I_\eta(T)|}\Pr\begin{bmatrix}\exists E'_1, \mathbb{C}_0 \to'^* E'_1, (\sigma, P') \wedge\\ E'_{1|\mathrm{Dom}(E'_0)} \sim_{\sigma,\top,\mathrm{indep}_{P'}(x)} E'_0\end{bmatrix}\\
&\leq \frac{1}{|I_\eta(T)|}\Pr\begin{bmatrix}\exists(E_1, E'_1),\\ \mathbb{C}_0 \to'^* E'_1, (\sigma, P') \to' E_1, (\sigma, P)\\ \wedge E_{1|\mathrm{Dom}(E_0)} \sim_{\sigma,\top,\mathrm{indep}_P(x)} E_0\end{bmatrix}
\end{aligned}$$

  The first step comes from the semantic rule (New), the second step from (L1) at P', the last step from the special $\to'$ semantics of new. This inequality proves (L1) at P.

- Case $P' = \mathsf{find}\ (\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ $\mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}$ P'', $\mathrm{depend}_P(x) = \mathrm{depend}_{P'}(x) = \top$, $P = P_j$, $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x) \cup \{M' \mid M' \in \mathrm{subterms}(M)$ for some $\mathsf{defined}(M) \in \mathcal{F}_{P_j}$, M' does not depend on x at $P'\}$. For all M' such that $M' \in \mathrm{subterms}(M)$ for some $\mathsf{defined}(M) \in \mathcal{F}_{P_j}$ and M' does not depend on x at P', M' does not depend on x at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$. Since $\mathcal{F}_P$ is correct for all P, for all E, σ such that $\Pr[\mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C}] > 0$, we have $E, \sigma \vdash \mathcal{F}_P$, so $E, \sigma, M' \Downarrow a$ for some a. So, by (S7), (L1) holds at P.

- Case $P' = \mathsf{let}\ y[\widetilde{i}] : T' = M\ \mathsf{in}\ P$, $y \neq x$, M does not depend on x at P'. The term M does not depend on x at P with $\mathrm{indep}_{P'}(x)$ instead of $\mathrm{indep}_P(x)$. By (S7), (L1) holds at P with $\mathrm{indep}_{P'}(x) \cup \{M\}$ instead of $\mathrm{indep}_P(x)$. In all traces (of non-zero probability) considered in (L1), we have $E, \sigma, y[\widetilde{i}] \Downarrow b$ if and only if $E, \sigma, M \Downarrow b$ and $E_1, \sigma, y[\widetilde{i}] \Downarrow b$ if and only if $E_1, \sigma, M \Downarrow b$, so (L1) holds at P with $\mathrm{indep}_P(x) = \mathrm{indep}_{P'}(x) \cup \{y[\widetilde{i}]\}$.

This result concludes the proof of soundness of the dependency analysis.

We now show the soundness of simplifyTerm. Essentially, when M simplifies to M', M and M' evaluate to the same value except in cases of negligible probability. More precisely, we show **S11**: for each P, M, M', if $M' = \mathrm{simplifyTerm}(M, P)$, then $\Pr[\exists(E, \sigma, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E, \sigma, (M' = M) \Downarrow \mathrm{false}] \leq q'(\eta) p_{\max}(\eta)$ for some polynomial q'. The proof proceeds by induction on the derivation that $M' = \mathrm{simplifyTerm}(M, P)$. We only consider the case $\mathrm{simplifyTerm}(M_1 = M_2, P) = \mathrm{false}$; the other cases are similar or easy. We show that if $\mathrm{simplifyTerm}(M_1 = M_2, P) = \mathrm{false}$ then $p = \Pr[\exists(E, \sigma, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E, \sigma, (M_1 = M_2) \Downarrow \mathrm{true}] \leq q'(\eta) p_{\max}(\eta)$ for some polynomial q'. When $\mathrm{depend}_P(x) = \top$, let $M_0 = M_1$; otherwise, let $M_0 = M_1 \mathrm{depend}_P(x)$. Let M'_0 and M'_2 be obtained respectively from M_0 and M_2 by replacing all array indices that depend on x at P with fresh replication indices. We assume that M'_0 characterizes a part of $x[\widetilde{i}]$ at P, and M'_2 does not depend on x at P.

Let σ and σ' be fixed, such that σ' is an extension of σ to the fresh replication indices of M'_0 and M'_2. We denote by $\overline{E}$ equivalence classes for $\sim_{\sigma,P} = \sim_{\sigma',P}$. We show that for all a, for all $\overline{E}$, there exists b such that for all $E \in \overline{E}$, if $E, \sigma', M'_0 \Downarrow a$, then $E, \sigma', f_1(\ldots f_k(x[\widetilde{i}])) \Downarrow b$.

- Assume that there exists $E' \in \overline{E}$ such that $E', \sigma', M'_0 \Downarrow a$. We define an environment E'' by $E''(y[\widetilde{a}]) = E(y[\widetilde{a}])$ for all $y[\widetilde{a}] \in \mathrm{Dom}(E)$ and $E''((\alpha y)[\widetilde{a}]) = E'(y[\widetilde{a}])$ for variables y renamed to fresh variables by α. We have $E''((\alpha y)[\widetilde{a}]) = E'(y[\widetilde{a}])$ for all $y[\widetilde{a}] \in \mathrm{Dom}(E')$, since when $\alpha y = y$, $E'(y[\widetilde{a}]) = E(y[\widetilde{a}])$ since $E \sim_{\sigma,P} E'$. Hence $E'', \sigma', M'_0 \Downarrow a$ and $E'', \sigma', \alpha M'_0 \Downarrow a$, so $E'', \sigma', (\alpha M'_0 = M'_0) \Downarrow \mathrm{true}$. So by rewriting, $E'', \sigma', (f_1(\ldots f_k((\alpha x)[\widetilde{i}])) = f_1(\ldots f_k(x[\widetilde{i}]))) \Downarrow \mathrm{true}$. Let b such that $E'', \sigma', f_1(\ldots f_k((\alpha x)[\widetilde{i}])) \Downarrow b$. Then $E'', \sigma', f_1(\ldots f_k(x[\widetilde{i}])) \Downarrow b$.

- Otherwise, there exists no $E \in \overline{E}$ such that $E, \sigma', M'_0 \Downarrow a$, so the result holds trivially.

So there exists a function f such that for all a, for all $\overline{E}$, for all $E \in \overline{E}$, if $E, \sigma', M'_0 \Downarrow a$, then $E, \sigma', f_1(\ldots f_k(x[\widetilde{i}])) \Downarrow f(a, \sigma', \overline{E})$.

If $E, \sigma, (M_1 = M_2) \Downarrow \mathrm{true}$ and $E \in \overline{E}$, $E, \sigma, M_1 \Downarrow a$ and $E, \sigma, M_2 \Downarrow a$ for some a. Then $E, \sigma, M_0 \Downarrow a$ by Item 1 of (L1). So there exists an extension σ' of σ to the fresh replication indices of M'_0 and M'_2 such that $E, \sigma', M'_0 \Downarrow a$ and $E, \sigma', M'_2 \Downarrow a$. Then $E, \sigma', f_1(\ldots f_k(x[\widetilde{i}])) \Downarrow f(a, \sigma', \overline{E})$. Since $E, \sigma', M'_2 \Downarrow a$ and M'_2 does not depend on x at P, by (S6), we have $a = f'(\sigma', \overline{E})$ for some function f', hence $E(x[\sigma\widetilde{i}]) \in S_x(\sigma', \overline{E}) = (I_\eta(f_1) \circ \ldots \circ I_\eta(f_k))^{-1}(f(f'(\sigma', \overline{E}), \sigma', \overline{E}))$. Let $T_1, \ldots, T_k$ be the types of the arguments of $f_1, \ldots, f_k$ respectively; let

$T_0 = T'$ be the type of the result of f_1; $T_k = T$. We have $|S_x(\sigma',\overline{E})| \leq \frac{|I_\eta(T_1)|}{|I_\eta(T_0)|} \times \ldots \times \frac{|I_\eta(T_k)|}{|I_\eta(T_{k-1})|} = \frac{|I_\eta(T_k)|}{|I_\eta(T_0)|} = \frac{|I_\eta(T)|}{|I_\eta(T')|}$, since $f_1, \ldots, f_k$ are uniform. Let $\widetilde{i}' = \mathrm{Dom}(\sigma)$ be the current replication indices at P.

$$p = \Pr\begin{bmatrix}\exists(E,\sigma,\mathcal{Q},\mathcal{C}), \mathbb{C}_0 \to^* E,(\sigma,P),\mathcal{Q},\mathcal{C}\\ \wedge\, E,\sigma,(M_1 = M_2) \Downarrow \text{true}\end{bmatrix}$$
$$\leq \sum_{\overline{E}}\sum_{\sigma'} \Pr\begin{bmatrix}\exists(E,\mathcal{Q},\mathcal{C}), \mathbb{C}_0 \to^* E,(\sigma'_{|\widetilde{i}'},P),\mathcal{Q},\mathcal{C}\\ \wedge\, E \in \overline{E} \wedge E(x[\sigma'\widetilde{i}]) \in S_x(\sigma',\overline{E})\end{bmatrix}$$
$$\leq \sum_{\overline{E}}\sum_{\sigma'}\sum_{b\in S_x(\sigma',\overline{E})} \Pr\begin{bmatrix}\exists(E,\mathcal{Q},\mathcal{C}),\\ \mathbb{C}_0 \to^* E,(\sigma'_{|\widetilde{i}'},P),\mathcal{Q},\mathcal{C}\\ \wedge\, E \in \overline{E} \wedge E(x[\sigma'\widetilde{i}]) = b\end{bmatrix}$$
$$\leq \sum_{\overline{E}}\sum_{\sigma'}\sum_{b\in S_x(\sigma',\overline{E})} \frac{1}{|I_\eta(T)|}\Pr\begin{bmatrix}\exists E', \mathbb{C}_0 \to'^* E',(\sigma'_{|\widetilde{i}'},P)\\ \wedge\, E'_{|\mathrm{Dom}(\overline{E})} \in \overline{E}\end{bmatrix}$$

by Item 3 of (L1). ($\mathrm{Dom}(\overline{E})$ denotes the domain of an element of $\overline{E}$, for instance the smallest one.)

$$p \leq \frac{1}{|I_\eta(T')|}\sum_{\sigma'}\sum_{\overline{E}} \Pr\begin{bmatrix}\exists E', \mathbb{C}_0 \to'^* E',(\sigma'_{|\widetilde{i}'},P)\\ \wedge\, E'_{|\mathrm{Dom}(\overline{E})} \in \overline{E}\end{bmatrix}$$
$$\leq \frac{q_1(\eta)}{|I_\eta(T')|}$$

where $q_1(\eta)$ is the number of possible σ', which is polynomial in η.

We now show the correctness of the game simplifications performed in depAnal. If Q_0 is the process before simplification and Q'_0 the process after simplification, we show that $Q_0 \approx^V Q'_0$. For simplicity, we consider one transformation at a time, and use transitivity of $\approx^V$ to conclude when several transformations are applied. For each trace $\mathrm{initConfig}(C[Q_0]) \to^* E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$, except in cases of negligible probability, we show that there exists a corresponding trace $\mathrm{initConfig}(C[Q'_0]) \to^* E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ with $E'_{m'} = E_m$, $P'_{m'}$ is obtained from P_m by the same transformation as Q'_0 from Q_0, $\mathcal{Q}'_{m'}$ is obtained from $\mathcal{Q}_m$ by the same transformation as Q'_0 from Q_0, $\mathcal{C}'_{m'} = \mathcal{C}_m$, with the same probability. The proof proceeds by induction on m. The case $m = 0$ is obvious, since the game simplifications do not change input processes. For the inductive step, we reason by cases on the last reduction step of the trace of $C[Q_0]$. We consider only the cases in which the transition may be altered by the game simplification.

- Case 1: When $\mathrm{simplifyTerm}(M,P) = M'$, we replace M with M' in P. We exclude traces such that $E,\sigma \not\vdash M = M'$. (They have negligible probability by (S11).) In the remaining traces, $E,\sigma \vdash M = M'$. So $E,\sigma,M \Downarrow a$ if and only if $E,\sigma,M' \Downarrow a$, and the transformed process reduces in the same way as the initial process.

- Case 2: When $M_j = \text{false}$, we remove the j-th branch of find ($\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then P_j) else P' In all traces $E,\sigma,(\mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j) \Downarrow \text{false}$, so in the reduction rule (Find1), the set S never contains $(j,\widetilde{v})$ for any $\widetilde{v}$, hence by (Find1) or (Find2), the process takes the same branch of the find with the same probability, whether or not the j-th branch is present.

- The other cases are similar.

We also show the converse property: for each trace of $C[Q'_0]$, except in cases of negligible probability, there exists a corresponding trace of $C[Q_0]$ with the same probability. Moreover, for all channels c and bitstrings a, $E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ executes $\overline{c}\langle a\rangle$ immediately if and only if $E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ executes $\overline{c}\langle a\rangle$ immediately, so $\Pr[C[Q_0] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C[Q'_0] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$, which yields the desired equivalence $Q_0 \approx^V Q'_0$.

Correctness of the equational prover. We say that $E,\sigma \vdash (\mathcal{F},\mathcal{R})$ when $E,\sigma \vdash \mathcal{F}$ and for all $(M_1 \to M_2) \in \mathcal{R}$, $E,\sigma \vdash M_1 = M_2$. For each P, the equational prover rewrites pairs $\mathcal{F},\mathcal{R}$ starting from $(\mathcal{F}_P,\emptyset)$ according to a certain sequence. We denote by $(\mathcal{F}_j,\mathcal{R}_j)(P)$ the j-th element of this sequence. So we have $(\mathcal{F}_0,\mathcal{R}_0)(P) = (\mathcal{F}_P,\emptyset)$, and for all j, we have $\frac{(\mathcal{F}_{j-1},\mathcal{R}_{j-1})(P)}{(\mathcal{F}_j,\mathcal{R}_j)(P)}$. Let $p_{m'}(P) = \Pr[\exists(E,\sigma,\mathcal{Q},\mathcal{C}), \mathbb{C}_0 \to^* E,(\sigma,P),\mathcal{Q},\mathcal{C} \wedge E,\sigma \not\vdash (\mathcal{F}_{m'},\mathcal{R}_{m'})(P)]$. We show **S12**: for each P, $p_{m'}(P) \leq q'(\eta)p_{\max}(\eta)$ for some polynomial q'. The proof proceeds by induction on m'. For $m' = 0$, this is an immediate consequence of the property that $E,\sigma \vdash (\mathcal{F}_0,\mathcal{R}_0)(P) = (\mathcal{F}_P,\emptyset)$ since $\mathcal{F}_P$ is correct for all P, with $q'(\eta) = 0$. For the inductive step,

$$p_{m'}(P) \leq p_{m'-1}(P)$$
$$+ \Pr\begin{bmatrix}\exists(E,\sigma,\mathcal{Q},\mathcal{C}), \mathbb{C}_0 \to^* E,(\sigma,P),\mathcal{Q},\mathcal{C}\\ \wedge\, E,\sigma \vdash (\mathcal{F}_{m'-1},\mathcal{R}_{m'-1})(P)\\ \wedge\, E,\sigma \not\vdash (\mathcal{F}_{m'},\mathcal{R}_{m'})(P)\end{bmatrix}$$

By induction hypothesis, $p_{m'-1}(P) \leq q'(\eta)p_{\max}(\eta)$ for some polynomial q'. So we just have to show that if $\frac{\mathcal{F},\mathcal{R}}{\mathcal{F}',\mathcal{R}'}$ then $\Pr[\exists(E,\sigma,\mathcal{Q},\mathcal{C}), \mathbb{C}_0 \to^* E,(\sigma,P),\mathcal{Q},\mathcal{C} \wedge E,\sigma \vdash (\mathcal{F},\mathcal{R}) \wedge E,\sigma \not\vdash (\mathcal{F}',\mathcal{R}')] \leq q'(\eta)p_{\max}(\eta)$ for some polynomial q'. We proceed by cases on the derivation of $\frac{\mathcal{F},\mathcal{R}}{\mathcal{F}',\mathcal{R}'}$.

- The cases (2), (5), (7), as well as the cases (1) and (6) when the reduction uses a rule of $\mathcal{R}$, are obvious and there is no loss of probability (that is, $q'(\eta) = 0$.)

- Cases (1) and (6) when the reduction uses a user-defined rewrite rule new $y_1 : T'_1, \ldots,$ new $y_l : T'_l, \forall x_1 : T_1, \ldots, \forall x_m : T_m, M_1 \to M_2$, with associated probability $p(\eta)$: Assuming this user-defined claim is correct, when $E,\sigma \vdash (\mathcal{F},\mathcal{R})$ but $E,\sigma \not\vdash (\mathcal{F}',\mathcal{R}')$, for at least one value of the indices of restrictions that correspond to $y_1, \ldots, y_l$, the process $C[Q_0]$ provides an adversary that satisfies the conditions of the definition of the corresponding user claim. (The proof of Proposition 2 below details a similar argument in a more complicated case.) So the probability that $E,\sigma \vdash (\mathcal{F},\mathcal{R})$ and $E,\sigma \not\vdash (\mathcal{F}',\mathcal{R}')$ is at most $p(\eta)$ times the number of possible values for the indices of restrictions that correspond to $y_1, \ldots, y_l$, which is polynomial in η, so the result holds with $q'(\eta)$ equal to the number of possible values for the indices of restrictions that correspond to $y_1, \ldots, y_l$.

- Case (3): Assume that $E,\sigma \vdash (\mathcal{F},\mathcal{R})$ and $E,\sigma \not\vdash (\mathcal{F}',\mathcal{R}')$. So for all $j \leq m$, $E,\sigma,M_j \Downarrow a_j$,

$E, \sigma, M'_j \Downarrow a'_j$, $(a_1, \ldots, a_m) \neq (a'_1, \ldots, a'_m)$, and $E(x[a_1, \ldots, a_m]) = E(x[a_1, \ldots, a_m])$. Since for each $a_1, \ldots, a_m$, $x[a_1, \ldots, a_m]$ is chosen randomly with uniform probability among $|I_\eta(T)|$ values, the probability that this happens is smaller than $\frac{q''(\eta)(q''(\eta)-1)}{2|I_\eta(T)|}$ where $q''(\eta)$ is the number of possible values of $a_1, \ldots, a_m$, which is a polynomial in η.

- Case (4): We first show that, if M characterizes a part of x with $S_{\text{def}}, S_{\text{dep}}$, then for all M_0 obtained from M by substituting variables of S_{def} with their definition, there exist a tuple of terms $\widetilde{M}$, a large type T, and uniform functions $f_1, \ldots, f_k$ such that T is the type of the result of f_1 (or of x when $k = 0$) and for each a, E_0, and σ, there exists b such that for all E such that E equals E_0 on variables not in S_{dep}, if $E, \sigma, M_0 \Downarrow a$ then $E, \sigma, f_1(\ldots f_k(x[\widetilde{M}])) \Downarrow b$. Indeed, $\mathcal{M}_0 = \{\alpha M_0 = M_0\}$ is rewritten into a set that contains $f_1(\ldots f_k((\alpha x)[\widetilde{M'}])) = f_1(\ldots f_k(x[\widetilde{M}]))$. Due to the form of rewrite rules, $(\alpha x)[\widetilde{M'}]$ is a subterm of αM_0 and $x[\widetilde{M}]$ is a subterm of M_0. Moreover, the variables in S_{dep} do not occur in $\widetilde{M}$ or $\widetilde{M'}$.
  - If a is such that there exists E' such that E' equals E_0 on variables not in S_{dep}, $E', \sigma, \alpha M_0 \Downarrow a$ and E' defines variables of αM_0, let b such that $E', \sigma, f_1(\ldots f_k((\alpha y)[\widetilde{M'}])) \Downarrow b$. Then for all E such that E equals E_0 on variables not in S_{dep} and $E, \sigma, M_0 \Downarrow a$, we can define the E'' that maps variables of M_0 as E and variables of αM_0 as E'. Then $E'', \sigma, (\alpha M_0 = M_0) \Downarrow \text{true}$, so by rewriting $E'', \sigma, f_1(\ldots f_k((\alpha x)[\widetilde{M'}])) = f_1(\ldots f_k(x[\widetilde{M}])) \Downarrow \text{true}$, so $E, \sigma, f_1(\ldots f_k(x[\widetilde{M}])) \Downarrow b$.
  - Otherwise, there is no E such that E equals E_0 on variables not in S_{dep} and $E, \sigma, M_0 \Downarrow a$, so the result holds trivially.

So there exists a function f such that for each a, σ, E, if $E, \sigma, M_0 \Downarrow a$ then $E, \sigma, f_1(\ldots f_k(x[\widetilde{M}])) \Downarrow f(a, \sigma, E_{|\overline{S_{\text{dep}}}})$. Since the variables in S_{dep} do not occur in $\widetilde{M}$, there exists a tuple of functions $\widetilde{f}$ such that $E, \sigma, \widetilde{M} \Downarrow \widetilde{f}(\sigma, E_{|\overline{S_{\text{dep}}}})$. So $E, \sigma, f_1(\ldots f_k(x[\widetilde{f}(\sigma, E_{|\overline{S_{\text{dep}}}})])) \Downarrow f(a, \sigma, E_{|\overline{S_{\text{dep}}}})$.

Let us now consider the three cases of Rule (4). In each case, we show that $p = \Pr[\exists E, \exists \sigma, \exists \mathcal{Q}, \exists \mathcal{C}, \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge E, \sigma \vdash M_1 = M_2] \leq q'(\eta) p_{\max}(\eta)$ for some polynomial q' and for M_1, M_2 that satisfy the hypothesis of Rule (4).

– First case: M'_1 is obtained from M_1 by replacing all array indices that are not replication indices with fresh replication indices, x occurs in M'_1, x is defined by restrictions new $x : T'$, T' is a large type, M'_1 characterizes a part of x, and M_2 is obtained by optionally applying function symbols to terms of the form $y[\widetilde{M'}]$ where y is defined by restrictions and $y \neq x$.

Let M'_2 be obtained from M_2 by replacing all array indices that are not replication indices with fresh replication indices. Let S_{indep} be the set of variables defined only by restrictions, excluding x. Since M'_1 characterizes a part of x, there exist a large type T, functions f and $\widetilde{f}$, and uniform functions $f_1, \ldots, f_k$ such that T is the type of the result of f_1 (or of x when $k = 0$) and for each a, E, and σ, if $E, \sigma, M_1 \Downarrow a$ then $E, \sigma, f_1(\ldots f_k(x[\widetilde{f}(\sigma, E_{|S_{\text{indep}}})])) \Downarrow f(a, \sigma, E_{|S_{\text{indep}}})$.

If $E, \sigma \vdash M_1 = M_2$ then we have $E, \sigma, M_1 \Downarrow a$ and $E, \sigma, M_2 \Downarrow a$ for some a. Then there exists an extension σ' of σ to the fresh replication indices of M'_1 and M'_2 such that $E, \sigma', M'_1 \Downarrow a$ and $E, \sigma', M'_2 \Downarrow a$. So $E, \sigma', f_1(\ldots f_k(x[\widetilde{f}(\sigma', E_{|S_{\text{indep}}})])) \Downarrow f(a, \sigma', E_{|S_{\text{indep}}})$ and since only the variables of S_{indep} occur in M'_2, there is a function f' such that $a = f'(\sigma', E_{|S_{\text{indep}}})$. So

$$E(x[\widetilde{f}(\sigma', E_{|S_{\text{indep}}})]) \in S_x(\sigma, E_{|S_{\text{indep}}}) = (I_\eta(f_1) \circ \ldots \circ I_\eta(f_k))^{-1}(f(f'(\sigma', E_{|S_{\text{indep}}}), \sigma', E_{|S_{\text{indep}}})).$$

Let $T_1, \ldots, T_k$ be the types of the arguments of $f_1, \ldots, f_k$ respectively; $T_0 = T$, $T_k = T'$. We have $|S_x(\sigma, E_{|S_{\text{indep}}})| \leq \frac{|I_\eta(T_1)|}{|I_\eta(T_0)|} \times \ldots \times \frac{|I_\eta(T_k)|}{|I_\eta(T_{k-1})|} = \frac{|I_\eta(T_k)|}{|I_\eta(T_0)|} = \frac{|I_\eta(T')|}{|I_\eta(T)|}$ since $f_1, \ldots, f_k$ are uniform. Let E_{indep} be an environment giving values to variables of S_{indep}. Let $\widetilde{i'} = \text{Dom}(\sigma)$ be the current replication indices at P.

$$\begin{aligned}
p &\leq \sum_{\sigma'} \sum_{E_{\text{indep}}} \Pr \begin{bmatrix} \exists (E, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma'_{\widetilde{i'}}, P), \mathcal{Q}, \mathcal{C} \\ \wedge E_{|S_{\text{indep}}} = E_{\text{indep}} \wedge \\ E(x[\widetilde{f}(\sigma', E_{\text{indep}})]) \in S_x(\sigma', E_{\text{indep}}) \end{bmatrix} \\
&\leq \sum_{\sigma'} \frac{1}{|I_\eta(T)|} \sum_{E_{\text{indep}}} \Pr \begin{bmatrix} \exists (E, \mathcal{Q}, \mathcal{C}), \\ \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \\ \wedge E_{|S_{\text{indep}}} = E_{\text{indep}} \end{bmatrix} \\
&\leq \frac{q_1(\eta)}{|I_\eta(T)|}
\end{aligned}$$

where $q_1(\eta)$ is the number of possible σ', which is polynomial in η. So the result follows with $q'(\eta) = q_1(\eta)$.

– Second case: x occurs in M_1, x is defined by restrictions new $x : T'$, T' is a large type, M_1 characterizes a part of x, only_dep$(x) = S$, and no variable of S occurs in M_2.

We consider traces of $C[Q_0]$ that differ by the choices of values of x. Since only_dep$(x) = S$, these traces differ only by the values of variables in S, after excluding exceptional traces in which we have $E, \sigma, (M_1 = M_2) \Downarrow \text{true}$ for M_1, M_2 considered in Rule (4) or for some test $M_1 = M_2$ or $M_1 \neq M_2$ in Q_0 such that M_1 characterizes a part of x with $S \setminus \{x\}, S$, and no variable in S occurs in M_2.

In the considered traces, the value of M_2 is the same a, which is therefore a function of σ and $E_{|\overline{S}}$, so $a = f'(\sigma, E_{|\overline{S}})$. Assume that $E, \sigma, (M_1 = M_2) \Downarrow \text{true}$. Then $E, \sigma, M_1 \Downarrow a$. Then there is some M_0 obtained from M_1 by substituting variables in $S \setminus \{x\}$ with their definition such that $E, \sigma, M_0 \Downarrow a$. (We choose the definition of these variables used to set them in environment E.) When M_1, M_2 come from Rule (4), we set $M_0 = M_1$. The number of choices of M_0 is independent of η: it can be bounded

knowing the number of different definitions of variables in S and the number of occurrences of these variables in the terms M_1.

Due to the properties of "characterize", there exist a large type T, functions f and $\widetilde{f}$, and uniform functions $f_1, \dots, f_k$ such that T is the type of the result of f_1 (or of x when $k = 0$) and for each a, σ, E, if $E, \sigma, M_0 \Downarrow a$ then $E, \sigma, f_1(\dots f_k(x[\widetilde{f}(\sigma, E_{|\overline{S}})])) \Downarrow f(a, \sigma, E_{|\overline{S}})$. So $E(x[\widetilde{f}(\sigma, E_{|\overline{S}})]) \in S_x(\sigma, E_{|\overline{S}}) = (I_\eta(f_1) \circ \dots \circ I_\eta(f_k))^{-1}(f(f'(\sigma, E_{|\overline{S}}), \sigma, E_{|\overline{S}})$. Let $T_1, \dots, T_k$ be the types of the arguments of $f_1, \dots, f_k$ respectively; $T_0 = T$, $T_k = T'$. We have $|S_x(\sigma, E_{|\overline{S}})| \leq \frac{|I_\eta(T_1)|}{|I_\eta(T_0)|} \times \dots \times \frac{|I_\eta(T_k)|}{|I_\eta(T_{k-1})|} = \frac{|I_\eta(T_k)|}{|I_\eta(T_0)|} = \frac{|I_\eta(T')|}{|I_\eta(T)|}$ since $f_1, \dots, f_k$ are uniform.

The probability that $E, \sigma, (M_1 = M_2) \Downarrow \mathrm{true}$ is at most the sum for all choices of M_0 of the probability that $E(x[\widetilde{f}(\sigma, E_{|\overline{S}})]) \in S_x(\sigma, E_{|\overline{S}})$, so it is at most $\sum_{M_0} \frac{1}{|I_\eta(T)|}$. (Note that T may depend on the choice of M_0.) Therefore, the probability of excluded traces is at most $\sum_{M_1, M_2} \sum_{M_0} \frac{q_1(\eta)}{|I_\eta(T)|}$ where the number of possible σ, that is, the number of executions of the test $M_1 = M_2$ or $M_1 \neq M_2$ is at most $q_1(\eta)$, polynomial in η.

For traces that have not been excluded, $E, \sigma, (M_1 = M_2) \Downarrow \mathrm{false}$, so the result follows with $q'(\eta) = \sum_{M_1, M_2} \sum_{M_0} q_1(\eta)$.

– Third case: $\mathrm{simplifyTerm}(M_1 = M_2, P) = \mathrm{false}$. The result follows immediately from the correctness of the local dependency analysis, Property (S11).

Similarly, we also have **S12'**: For each Q', $\Pr[\exists(E, \sigma, P, \mathcal{Q}, \mathcal{C}, c, M_1, \dots, M_l, \underline{N_1, \dots, N_k, Q''}, \sigma', \mathcal{Q}', \mathcal{C}'), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \wedge P = \overline{c[M_1, \dots, M_l]}\langle N_1, \dots, N_k \rangle.Q'' \wedge E, \{(\sigma, Q'')\}, \mathcal{C} \rightsquigarrow^* E, \mathcal{Q}', \mathcal{C}' \wedge (\sigma', Q') \in \mathcal{Q}' \wedge E, \sigma' \not\vdash (\mathcal{F}_{m'}, \mathcal{R}_{m'})(Q')] \leq q'(\eta) p_{\max}(\eta)$ for some polynomial q'.

We have $\mathcal{F}_{Q'} = \mathcal{F}_P$, hence $(\mathcal{F}_{m'}, \mathcal{R}_{m'})(Q') = (\mathcal{F}_{m'}, \mathcal{R}_{m'})(P)$, and σ' is an extension of σ, so $E, \sigma \vdash (\mathcal{F}_{m'}, \mathcal{R}_{m'})(P)$ implies $E, \sigma' \vdash (\mathcal{F}_{m'}, \mathcal{R}_{m'})(Q')$. So the result follows from (S12).

Correctness of game simplification. For simplicity, we consider one transformation at a time, and use transitivity of $\approx^V$ to conclude when several transformations are applied. For each trace $\mathrm{initConfig}(C[Q_0]) \to^* E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$, except in cases of negligible probability, we show that there exists a corresponding trace $\mathrm{initConfig}(C[Q_0']) \to^* E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ with $E'_{m'} = E_m$, $P'_{m'}$ is obtained from P_m by the same transformation as Q_0' from Q_0, $\mathcal{Q}'_{m'}$ is obtained from $\mathcal{Q}_m$ by the same transformation as Q_0' from Q_0, $\mathcal{C}'_{m'} = \mathcal{C}_m$, with the same probability. The proof proceeds by induction on m.

For the case $m = 0$, the only simplification that can be applied to input processes is the simplification of terms in input channels. Moreover, if Q' is the transformed process, $\mathcal{F}_{Q'} = \emptyset$ since $\mathcal{F}_{C[Q_0]} = \emptyset$ and Q' is obtained from $C[Q_0]$ by $\rightsquigarrow$, which reduces only input processes. So $(\mathcal{F}_0, \mathcal{R}_0)(Q') = (\emptyset, \emptyset)$. No rule of the equational prover applies on $(\emptyset, \emptyset)$, so $(\mathcal{F}_{m'}, \mathcal{R}_{m'})(Q') = (\emptyset, \emptyset)$, hence no rewrite rule of $\mathcal{R}_{m'}$ can be applied. So one can only simplify terms in the input channel of Q' by a user-defined rewrite rule. The proof then proceeds exactly as in Case 1 below.

For the inductive step, we reason by cases on the last reduction step of the trace of $C[Q_0]$. We consider only the cases in which the transition may be altered by the game simplification.

- Case 1: M reduces into M' by a user-defined rewrite rule, and we replace M with M' in the smallest (input or output) process $P_M = C_M[M]$ that contains M. If $E, \sigma, M \Downarrow a$ then $E, \sigma, M' \Downarrow a'$ (since the variable accesses in M' are included in those of M and M and M' are well-typed). When $a \neq a'$, the game provides an adversary that satisfies the conditions of the definition of the corresponding user claim (as in the item "Cases (1) and (6) when the reduction uses a user-defined rewrite rule" above) so this situation has negligible probability and can be excluded. Otherwise, $a = a'$, and $C_M[M']$ reduces in the same way as $P_M = C_M[M]$.

- Case 2: M reduces into M' by a rule of $\mathcal{R}$, and we replace M with M' in the smallest process $P_M = C_M[M]$ that contains M, where $\mathcal{R}$ is the set of rewrite rules obtained by the equational prover from $\mathcal{F}_{P_M}$. We first assume that P_M is an output process. We exclude traces such that $E, \sigma \not\vdash (\mathcal{F}_{m'}, \mathcal{R}_{m'})(P_M)$. (They have negligible probability by (S12).) In the remaining traces, for all $(M_1 \to M_2) \in \mathcal{R} = \mathcal{R}_{m'}$, $E, \sigma \vdash M_1 = M_2$, so $E, \sigma \vdash M = M'$. So $E, \sigma, M \Downarrow a$ if and only if $E, \sigma, M' \Downarrow a$, and $C_M[M']$ reduces in the same way as $P_M = C_M[M]$. When we reduce a term in the channel of an input, we have a similar proof with an input process $Q_M = C_M[M]$ instead of P_M and using (S12') instead of (S12).

- Case 3: $P = \mathsf{find}\ (\bigoplus_{j=1}^m u_{j1}[\widetilde{i}] \leq n_{j1}, \dots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P'$, $\mathcal{F}_{P_j}$ yields a contradiction, and we remove the j-th branch of the find. We exclude traces in which $E, \sigma, (\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j) \Downarrow \mathrm{true}$. Let S we the set defined in the reduction rule (Find1). We have $|S| \leq \sum_{j=1}^m \prod_{l=1}^{m_j} n_{jl} = q(\eta)$ for some polynomial q, and $\mathrm{among}(S) = \frac{2^{k+f(\eta)} \operatorname{div} |S|}{2^{k+f(\eta)}}$ where k is the smallest integer such that $2^k \geq |S|$, so $\mathrm{among}(S) \geq \frac{2^{f(\eta)}}{2^{k+f(\eta)}} \geq \frac{1}{2^k} \geq \frac{1}{2|S|} \geq \frac{1}{2q(\eta)}$. By (Find1), P reduces into P_j with probability at least $\mathrm{among}(S)$, so at least $\frac{1}{2q(\eta)}$, when $E, \sigma, (\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j) \Downarrow \mathrm{true}$. Therefore,

$$\Pr\left[\begin{array}{l}\exists(E, \sigma, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P), \mathcal{Q}, \mathcal{C} \\ \wedge E, \sigma, (\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j) \Downarrow \mathrm{true}\end{array}\right]$$
$$\leq 2q(\eta) \Pr\left[\exists(E, \sigma, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P_j), \mathcal{Q}, \mathcal{C}\right]$$
$$\leq 2q(\eta) \Pr\left[\begin{array}{l}\exists(E, \sigma, \mathcal{Q}, \mathcal{C}), \mathbb{C}_0 \to^* E, (\sigma, P_j), \mathcal{Q}, \mathcal{C} \\ \wedge E, \sigma \not\vdash (\mathcal{F}_{m'}, \mathcal{R}_{m'})(P_j)\end{array}\right]$$

since $E, \sigma \not\vdash (\mathcal{F}_{m'}, \mathcal{R}_{m'})(P_j)$ is always true since $\mathcal{F}_{P_j}$ yields a contradiction. So the excluded traces have negligible probability by (S12). In the remaining traces, $E, \sigma, (\mathsf{defined}(M_{j1}, \dots, M_{jl_j}) \wedge M_j) \Downarrow \mathrm{false}$, so the set S

never contains $(j, \widetilde{v})$ for any $\widetilde{v}$, hence by (Find1) or (Find2), the process takes the same branch of the find with the same probability, whether or not the j-th branch is present.

- Case 4: $P_0 = \mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq n_{jm_j}\ \mathsf{suchthat\ defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)\ \mathsf{else}\ P'$, $x[N_1, \ldots, N_l]$ is a subterm of M_{jk}, and none of the following conditions holds: a) P_0 is under a definition of x in Q_0; b) Q_0 contains $Q_1 \mid Q_2$ such that a definition of x occurs in Q_1 and P_0 is under Q_2 or a definition of x occurs in Q_2 and P_0 is under Q_1; c) Q_0 contains $lp + 1$ replications above a process Q that contains a definition of x and P_0, where lp is the length of the longest common prefix between $N_1, \ldots, N_l$ and the current replication indices at the definitions of x. The j-th branch of the find is removed.

  We show that $x[N_1, \ldots, N_l]$ cannot be defined at P_0 as follows. We say that the formula $\phi(E, (\sigma, P), \mathcal{Q}, \mathcal{C})$ is true when one of the following condition holds:

  A. $x[a_1, \ldots, a_m] \in \mathrm{Dom}(E)$, $(\sigma'', P'') \in \mathcal{Q} \uplus \{(\sigma, P)\}$, P_0 is under P'', and $\sigma'' i''_k = a_k$ for all $k \leq \min(lp, |\mathrm{Dom}(\sigma'')|)$, where i''_k is the k-th replication index at P'';

  B. $\{(\sigma', P'), (\sigma'', P'')\} \subseteq \mathcal{Q} \uplus \{(\sigma, P)\}$ (multiset inclusion), P' contains a definition of x, P_0 is under P'', $\sigma' i'_k = \sigma'' i''_k$ for all $k \leq \min(lp, |\mathrm{Dom}(\sigma')|, |\mathrm{Dom}(\sigma'')|)$ where i'_k is the k-th replication index at P' and i''_k is the k-th replication index at P'';

  C. $(\sigma', P') \in \mathcal{Q} \uplus \{(\sigma, P)\}$ where

     C.a. P_0 is under a definition of x in P';

     C.b. or P' contains $Q_1 \mid Q_2$ such that a definition of x occurs in Q_1 and P_0 is under Q_2 or a definition of x occurs in Q_2 and P_0 is under Q_1;

     C.c. or P' contains $lp + 1 - |\mathrm{Dom}(\sigma')|$ replications above a process Q that contains a definition of x and P_0.

  Next, we show that if a configuration in the trace satisfies ϕ, then the previous configuration also satisfies ϕ.

  More precisely, we first show that if $\phi(E, (\sigma, P), \mathcal{Q}'' \uplus \mathcal{Q}', \mathcal{C}')$ and $E, \mathcal{Q}, \mathcal{C} \rightsquigarrow E, \mathcal{Q}', \mathcal{C}'$, then $\phi(E, (\sigma, P), \mathcal{Q}'' \uplus \mathcal{Q}, \mathcal{C})$. The proof is by cases on the reduction rule of $\rightsquigarrow$. Case (Nil) is obvious. For rule (Par), if we are in case B and both processes P' and P'' are generated by (Par), then before applying (Par), we are in case C.b. In all other cases, we remain in the same case of the definition of ϕ before applying (Par). For rule (Repl), if we are in case B and both processes P' and P'' are generated by (Repl), then before applying (Repl), we are in case C.c. In all other cases, we remain in the same case before applying (Repl). For rules (NewChannel) and (Input), we remain in the same case.

  Therefore, if $\phi(E, (\sigma, P), \mathcal{Q}'' \uplus \mathcal{Q}', \mathcal{C}')$ and $E, \mathcal{Q}', \mathcal{C}' = \mathrm{reduce}(E, \mathcal{Q}, \mathcal{C})$, then $\phi(E, (\sigma, P), \mathcal{Q}'' \uplus \mathcal{Q}, \mathcal{C})$.

  We also show that, if $\phi(E', (\sigma', P'), \mathcal{Q}', \mathcal{C}')$ and $E, (\sigma, P), \mathcal{Q}, \mathcal{C} \xrightarrow{p}_t E', (\sigma', P'), \mathcal{Q}', \mathcal{C}'$, then $\phi(E, (\sigma, P), \mathcal{Q}, \mathcal{C})$. The proof is by cases on the reduction rule of $\xrightarrow{p}_t$. For rule (Find2), we remain in the same case of the definition of ϕ. For rules (New), (Let), (Find1), if we are in case A after applying the reduction and the reduction defines $x[a_1, \ldots, a_m]$, then we are in case C.a before the reduction if (σ'', P'') is (σ, P) and in case B otherwise. Otherwise, we remain in the same case. For rule (Output), $E, (\sigma, \overline{c[\widetilde{M}]}\langle N_1, \ldots, N_k\rangle.Q'')$, $\{(\sigma', c[\widetilde{a}](x_1[\widetilde{a'}] : T_1, \ldots, x_k[\widetilde{a'}] : T_k).P)\} \uplus \mathcal{Q}, \mathcal{C}$ is transformed into $E', (\sigma', P), \mathcal{Q} \uplus \{(\sigma, Q'')\}, \mathcal{C}$, where $E' = E[x_1[\widetilde{a'}] \mapsto \ldots, \ldots, x_k[\widetilde{a'}] \mapsto \ldots]$, then we reduce $E', \{(\sigma, Q'')\}, \mathcal{C}$ by the function reduce. By the property shown for reduce, we have $\phi(E', (\sigma', P), \mathcal{Q} \uplus \{(\sigma, Q'')\}, \mathcal{C})$. If we are in case A and the input defines $x[a_1, \ldots, a_m]$, then before (Output), we are in case C.a if (σ'', P'') is (σ, P) and in case B otherwise. Otherwise, we remain in the same case.

  Next, we show that if the j-th branch of the find is taken by (Find1) when evaluating P_0, then the last configuration of the trace satisfies ϕ. In this case, $x[a_1, \ldots a_m] \in \mathrm{Dom}(E)$ in a configuration $E, (\sigma, P_0), \mathcal{Q}, \mathcal{C}$ such that $\sigma i_k = a_k$ for all $k \leq lp$, where i_k is the k-th replication index at P_0. So $\phi(E, (\sigma, P_0), \mathcal{Q}, \mathcal{C})$ (case A).

  Therefore, by the previous proof, ϕ holds for the initial configuration, so we have $\phi(\emptyset, (\emptyset, \overline{start}\langle\rangle), \{(\emptyset, C[Q_0])\}, \emptyset)$. Case A cannot happen because E is empty; case B cannot happen because $\overline{start}\langle\rangle$ contains neither P_0 nor a definition of x and (σ', P') and (σ'', P'') cannot be the same process $(\emptyset, C[Q_0])$. So we are in case C with $P' = C[Q_0]$ and $\sigma' = \emptyset$. Since C contains neither P_0 nor a definition of x, we obtain that one of the conditions a), b), c) holds, which contradicts the hypothesis. So the j-th branch of the find cannot be taken, and can be removed.

- The other cases can be handled in a way similar to cases 1–3.

We also show the converse property: for each trace of $C[Q'_0]$, except in cases of negligible probability, there exists a corresponding trace of $C[Q_0]$ with the same probability. Moreover, for all channels c and bitstrings a, $E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ executes $\overline{c}\langle a\rangle$ immediately if and only if $E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ executes $\overline{c}\langle a\rangle$ immediately, so $\Pr[C[Q_0] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C[Q'_0] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$, which yields the desired equivalence.

We leave the proof of the additional transformations **MoveNew**, **RemoveAssign**(**useless**), and **SArename**(**auto**) to the reader. The proof technique is similar to that for **SArename**(x). □

E.2 Proving the Last Hypothesis of Proposition 5

In this section, we show how to prove the last hypothesis of Proposition 5. We use the notations of Proposition 5 and of the proof of **Simplify** in the previous section.

For each definition P of x in Q, we define $\mathrm{defRestr}_P(x[\widetilde{i}])$

as follows:

$$\mathrm{defRestr}_P(x[\widetilde{i}]) = \begin{cases} x[\widetilde{i}] & \text{if } P = \mathsf{new}\ x[\widetilde{i'}] : T; P' \\ z[M_1, \ldots, M_l]\{\widetilde{i}/\widetilde{i'}\} & \\ \quad \text{if } P = \mathsf{let}\ x[\widetilde{i'}] : T = z[M_1, \ldots, M_l] \ \mathsf{in}\ P' & \end{cases}$$

Let $\mathcal{F}_P[\widetilde{i}]$ denote the facts that hold at P with current replication indices renamed to $\widetilde{i}$, that is, $\mathcal{F}_P[\widetilde{i}] = \mathcal{F}_P\{\widetilde{i}/\widetilde{i'}\}$ where the replication indices at P are $\widetilde{i'}$.

For each pair of definitions of x, P, P', we check that, if $\mathrm{defRestr}_P(x[\widetilde{i}]) = z[M_1, \ldots, M_l]$ and $\mathrm{defRestr}_{P'}(x[\widetilde{i'}]) = z[M'_1, \ldots, M'_l]$, then $\mathcal{F}_P[\widetilde{i}] \cup \mathcal{F}_{P'}[\widetilde{i'}] \cup \{\widetilde{i} \neq \widetilde{i'}, M_1 = M'_1, \ldots, M_l = M'_l\}$ yields a contradiction. That is, $\widetilde{i} \neq \widetilde{i'} \wedge M_1 = M'_1 \wedge \ldots \wedge M_l = M'_l$ is false except in cases of negligible probability, taking into account the facts that are known to hold at P and P'. When this check succeeds, the last hypothesis of Proposition 5 holds, as shown by the next proposition.

Proposition 7 *Assume that, for all pairs P, P' of definitions of x in Q, if $\mathrm{defRestr}_P(x[\widetilde{i}]) = z[M_1, \ldots, M_l]$ and $\mathrm{defRestr}_{P'}(x[\widetilde{i'}]) = z[M'_1, \ldots, M'_l]$, then $\mathcal{F}_P[\widetilde{i}] \cup \mathcal{F}_{P'}[\widetilde{i'}] \cup \{\widetilde{i} \neq \widetilde{i'}, M_1 = M'_1, \ldots, M_l = M'_l\}$ yields a contradiction (with local dependency analysis disabled).*

Then $\Pr[\exists(\mathcal{T}, \widetilde{a}, \widetilde{a'}), C[Q]$ reduces according to $\mathcal{T} \wedge \widetilde{a} \neq \widetilde{a'} \wedge \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])]$ is negligible.

The local dependency analysis is disabled because it gives information valid only at a certain process occurrence, and here we combine facts obtained at two occurrences P and P'.

Proof Consider a trace $\mathcal{T}$ of $C[Q]$ and $\widetilde{a} \neq \widetilde{a'}$ such that $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])]$. Let P and P' be the processes that define $x[\widetilde{a}]$ and $x[\widetilde{a'}]$, respectively, in this trace. Let σ be mapping the replication indices at P to $\widetilde{a}$, σ' be mapping the replication indices at P' to $\widetilde{a'}$, and σ'' be mapping $\widetilde{i}$ to $\widetilde{a}$ and $\widetilde{i'}$ to $\widetilde{a'}$. Let E'' be the environment at the end of $\mathcal{T}$.

Just before the definition of $x[\widetilde{a}]$ is executed, the configuration of $\mathcal{T}$ is of the form $E, (\sigma, P), \ldots$, so, since $\mathcal{F}_P$ is correct for all P, $E, \sigma \vdash \mathcal{F}_P$, so $E'', \sigma'' \vdash \mathcal{F}_P[\widetilde{i}]$. Similarly, $E'', \sigma'' \vdash \mathcal{F}_{P'}[\widetilde{i'}]$. Since $\widetilde{a} \neq \widetilde{a'}$, $E'', \sigma'' \vdash \widetilde{i} \neq \widetilde{i'}$. Since $\mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])]$, $\mathrm{defRestr}_P(x[\widetilde{i}]) = z[M_1, \ldots, M_l]$, $\mathrm{defRestr}_{P'}(x[\widetilde{i'}]) = z[M'_1, \ldots, M'_l]$, for some $z, M_1, \ldots, M_l, M'_1, \ldots, M'_l$, and $E'', \sigma'' \vdash M_1 = M'_1$, ..., $E'', \sigma'' \vdash M_l = M'_l$. So $E'', \sigma'' \vdash \mathcal{F}_{P,P'}$, where $\mathcal{F}_{P,P'} = \mathcal{F}_P[\widetilde{i}] \cup \mathcal{F}_{P'}[\widetilde{i'}] \cup \{\widetilde{i} \neq \widetilde{i'}, M_1 = M'_1, \ldots, M_l = M'_l\}$.

Hence $\Pr[\exists(\mathcal{T}, \widetilde{a}, \widetilde{a'}), C[Q]$ reduces according to $\mathcal{T} \wedge \widetilde{a} \neq \widetilde{a'} \wedge \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])] \leq \sum_{P,P'} \Pr[\exists(E'', \sigma''), \mathbb{C}_0 \to^* E'', \ldots \wedge E'', \sigma'' \vdash \mathcal{F}_{P,P'}]$.

When the local dependency analysis is disabled, the proof of correctness of the equational prover (S12) shown in the previous section also shows that, if $\frac{\mathcal{F}, \mathcal{R}}{\mathcal{F}', \mathcal{R}'}$, then

$$\Pr\begin{bmatrix}\exists(E'', \sigma''), \mathbb{C}_0 \to^* E'', \ldots \\ \wedge E'', \sigma'' \vdash \mathcal{F}, \mathcal{R} \wedge E'', \sigma'' \not\vdash \mathcal{F}', \mathcal{R}'\end{bmatrix}$$

is negligible. Moreover, for all P and P' definitions of x in Q, since $\mathcal{F}_{P,P'}$ yields a contradiction, $\mathcal{F}_{P,P'}, \emptyset$ is transformed into $\mathrm{false}, \mathcal{R}'$ by the equational prover, so $\Pr[\exists(E'', \sigma''), \mathbb{C}_0 \to^* E'', \ldots \wedge E'', \sigma'' \vdash \mathcal{F}_{P,P'}]$ is negligible, which shows the desired result. □

E.3 Proof of Proposition 2

Proof of Proposition 2 The idea of the proof is to show that if an adversary (represented by a context C) distinguishes $[\![L]\!]$ from $[\![R]\!]$, then we can build an adversary $\mathcal{A}_a$ against the security of the mac for the key $\mathrm{mkgen}(r[a])$, for some $a \in I_\eta(n'')$.

Let C be an evaluation context acceptable for $[\![L]\!], [\![R]\!], \emptyset$.

We define a probabilistic polynomial Turing machine $\mathcal{A}_a$, for $a \in [1, I_\eta(n'')]$, as follows. $\mathcal{A}_a$ uses oracles $\mathrm{mac}(., k)$ and $\mathrm{check}(., k, .)$. $\mathcal{A}_a$ simulates $C[[\![L]\!]]$ except that:

- for $a' < a$, in copies corresponding to $i'' = a'$ of L, $\mathcal{A}_a$ computes $\mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[u]) \wedge (m = x[u]) \wedge \mathrm{check}(m, \mathrm{mkgen}(r), ma)\ \mathsf{then}\ \mathrm{true}\ \mathsf{else}\ \mathrm{false}$ instead of $\mathrm{check}(m, \mathrm{mkgen}(r), ma)$, and
- in the copy corresponding to $i'' = a$, $\mathcal{A}_a$ does not choose a random number $r[a]$, it calls the oracle $\mathrm{mac}(., k)$ on x instead of computing $\mathrm{mac}(x, \mathrm{mkgen}(r))$, and instead of computing $\mathrm{check}(m, \mathrm{mkgen}(r), ma)$, it computes $b_1 = \mathrm{check}(m, k, ma)$ using the oracle $\mathrm{check}(., k, .)$ and $b_2 = \mathsf{find}\ u \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[u]) \wedge (m = x[u]) \wedge b_1\ \mathsf{then}\ \mathrm{true}\ \mathsf{else}\ \mathrm{false}$; if $b_1 \neq b_2$, the execution of the Turing machine stops, with result (m, ma); otherwise, the execution continues using value $b_1 = b_2$.

When $\mathcal{A}_a$ has not stopped due to the last item above, it returns $\perp$ when the simulation of $C[[\![L]\!]]$ terminates.

When $\mathcal{A}_a$ returns (m, t), $b_1 \neq b_2$. Moreover, if $b_1 = 0$, then $b_2 = 0$ by definition of b_2. So $b_1 = 1$ and $b_2 = 0$. Therefore, there is no u such that $m = x[u]$, hence $\mathcal{A}_a$ has not called the oracle $\mathrm{mac}(., k)$ on m. Moreover, there exists a polynomial q such that for all a, $\mathcal{A}_a$ runs in time $q(\eta)$. So by Definition 1, $\max_a p_a(\eta)$ is negligible, where

$$p_a(\eta) = \Pr\begin{bmatrix} r \xleftarrow{R} I_\eta(T_{mr}); k \leftarrow \mathrm{mkgen}_\eta(r); (m, t) \leftarrow \mathcal{A}_a : \\ \mathrm{check}_\eta(m, k, t) \end{bmatrix}$$

Since $I_\eta(n'')$ is polynomial in η, $\sum_{a \in [1, I_\eta(n'')]} p_a(\eta) \leq \max_a p_a(\eta) \times I_\eta(n'')$ is also negligible.

On the other hand, let c be a channel and a' be a bitstring. We need to evaluate $|\Pr[C[[\![L]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle] - \Pr[C[[\![R]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle]|$. We consider three categories of pairs of traces $(\mathcal{T}, \mathcal{T}')$ where $\mathcal{T}$ and $\mathcal{T}'$ are traces of $C[[\![L]\!]$ and $C[[\![R]\!]]$ respectively:

1. Traces $\mathcal{T}$ and $\mathcal{T}'$ have the same configurations except for the replacement of L with R in processes, they terminate, and none of their configurations executes $\overline{c}\langle a'\rangle$ immediately.
2. Traces $\mathcal{T}$ and $\mathcal{T}'$ have the same configurations except for the replacement of L with R in processes up to a point at which their corresponding configurations both execute $\overline{c}\langle a'\rangle$ immediately.

3. Traces $\mathcal{T}$ and $\mathcal{T}'$ have the same configurations except for the replacement of L with R in processes up to a point at which their configurations differ because for some $a \in [1, I_\eta(n'')]$, for some messages m, ma received on channel $c_2[a]$ (where c_2 is the channel used in $[\![L]\!]$ and $[\![R]\!]$ for the second parallel process of L and R), the result returned by $[\![L]\!]$ differs from the one returned by $[\![R]\!]$. In this case, the simulating Turing machine that runs $r \stackrel{R}{\leftarrow} I_\eta(T_{mr}); k \leftarrow \mathrm{mkgen}_\eta(r)$ and executes $\mathcal{A}_a$ will return (m, ma), by construction.

All traces of $C[[\![L]\!]]$ fall in one of the above categories, and similarly for traces of $C[[\![R]\!]]$. Traces of the first category have no contribution to $\Pr[C[[\![L]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle]$ and to $\Pr[C[[\![R]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle]$; traces of the second category cancel out when computing $\Pr[C[[\![L]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle] - \Pr[C[[\![R]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle]$. So

$$\begin{aligned}
&|\Pr[C[[\![L]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle] - \Pr[C[[\![R]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle]| \\
&\leq \Pr[(\mathcal{T}, \mathcal{T}') \text{ is in the third category}] \\
&\leq \sum_{a \in [1, I_\eta(n'')]} \Pr[r \stackrel{R}{\leftarrow} I_\eta(T_{mr}); k \leftarrow \mathrm{mkgen}_\eta(r); (m,t) \leftarrow \mathcal{A}_a] \\
&\leq \sum_{a \in [1, I_\eta(n'')]} p_a(\eta)
\end{aligned}$$

Hence $|\Pr[C[[\![L]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle] - \Pr[C[[\![R]\!]] \rightsquigarrow_\eta \overline{c}\langle a'\rangle]|$ is negligible, so $[\![L]\!] \approx [\![R]\!]$. □

E.4 Proof of Proposition 3

Let us first introduce some notations. We denote by $L_{j_0,\dots,j_k}$ the subtrees of L defined as follows by induction on k. We define $L_1, \dots, L_{m'}$ such that $L = (L_1, \dots, L_{m'})$. The functional process $L_{j_0,\dots,j_k}$ being defined, we define $L_{j_0,\dots,j_k,1}, \dots, L_{j_0,\dots,j_k,m'}$ to be the immediate sub-functional-processes of $L_{j_0,\dots,j_k}$, so that $L_{j_0,\dots,j_k}$ is of the form $!^{i \leq n}\mathsf{new}\ y_1 : T_1; \dots; \mathsf{new}\ y_m : T_m; (L_{j_0,\dots,j_k,1}, \dots, L_{j_0,\dots,j_k,m'})$.

When $L_{j_0,\dots,j_k} = !^{i \leq n}\mathsf{new}\ y_1 : T_1; \dots; \mathsf{new}\ y_m : T_m; (L_{j_0,\dots,j_k,1}, \dots, L_{j_0,\dots,j_k,m'})$, we define $i_{j_0,\dots,j_k} = i$, $n_{j_0,\dots,j_k} = n$, $y_{(j_0,\dots,j_k),k'} = y_{k'}$, and $\mathrm{nNew}_{j_0,\dots,j_k} = m$.

When $L_{j_0,\dots,j_l} = (x_1 : T_1, \dots, x_m : T_m) \to FP$, we say that $L_{j_0,\dots,j_l}$ is a leaf of L, and we define $x_{(j_0,\dots,j_l),k'} = x_{k'}$, $T_{(j_0,\dots,j_l),k'} = T_{k'}$, and $\mathrm{nInput}_{j_0,\dots,j_l} = m$.

In order to prove Proposition 3, we define a context C such that $Q_0 \approx_0^V C[[\![L]\!]]$ and $C[[\![R]\!]] \approx_0^V Q_0'$. While Q_0 evaluates the terms in $\mathcal{M}$ directly, the context C will send messages to $[\![L]\!]$ in order to evaluate these terms in $C[[\![L]\!]]$. Similarly, the process Q_0' contains inlined versions of the functional processes in R, while $C[[\![R]\!]]$ computes the same result by sending messages to $[\![R]\!]$.

In order to define C, we first define a process $\mathrm{relay}(L)$ as follows:

$$\mathrm{relay}((G_1, \dots, G_m)) = \mathrm{relay}(G_1)^1 \mid \dots \mid \mathrm{relay}(G_m)^m$$

$$\begin{aligned}
&\mathrm{relay}(!^{i \leq n}\mathsf{new}\ y_1 : T_1; \dots; \mathsf{new}\ y_l : T_l; (G_1, \dots, G_m))_{\widetilde{i}}^{\widetilde{j}} = \\
&\quad !^{i \leq n} d_{\widetilde{j}}[\widetilde{i}, i](); \overline{c_{\widetilde{j}}[\widetilde{i}, i]}\langle\rangle; c_{\widetilde{j}}[\widetilde{i}, i](); \overline{d_{\widetilde{j}}[\widetilde{i}, i]}\langle\rangle; \\
&\qquad (\mathrm{relay}(G_1)_{\widetilde{i},i}^{\widetilde{j},1} \mid \dots \mid \mathrm{relay}(G_m)_{\widetilde{i},i}^{\widetilde{j},m} \mid \\
&\qquad !^{i' \leq n'} d_{\widetilde{j}}[\widetilde{i}, i](); \overline{d_{\widetilde{j}}[\widetilde{i}, i]}\langle\rangle) \\
&\mathrm{relay}((x_1 : T_1, \dots, x_l : T_l) \to FP)_{\widetilde{i}}^{\widetilde{j}} = \\
&\quad d_{\widetilde{j}}[\widetilde{i}](x_1 : T_1, \dots, x_l : T_l); \overline{c_{\widetilde{j}}[\widetilde{i}]}\langle x_1, \dots, x_l\rangle; \\
&\quad c_{\widetilde{j}}[\widetilde{i}](r : \mathit{bitstring}); \overline{d_{\widetilde{j}}[\widetilde{i}]}\langle r\rangle; \\
&\quad !^{i' \leq n'} d_{\widetilde{j}}[\widetilde{i}](x_1 : T_1, \dots, x_l : T_l); \overline{d_{\widetilde{j}}[\widetilde{i}]}\langle r\rangle
\end{aligned}$$

where $\widetilde{i} = i_1, \dots, i_{l'}$ and $\widetilde{j} = j_0, \dots, j_{l'}$. The relay process corresponding to replicated restrictions relays messages sent on channel $d_{\widetilde{j}}$ to channel $c_{\widetilde{j}}$ (used in $[\![L]\!]$ and $[\![R]\!]$) so that the corresponding random numbers $y_1, \dots, y_l$ are chosen by $[\![L]\!]$. When those random numbers have already been chosen, the process accepts messages on $d_{\widetilde{j}}$ but yields control back to the sending process without executing anything by outputting on $d_{\widetilde{j}}$. Thus, the caller of the relay process can harmlessly ask several times for choosing the same random numbers. Similarly, the relay process corresponding to a function relays the arguments of the function received on channel $d_{\widetilde{j}}$ to channel $c_{\widetilde{j}}$, so that $[\![L]\!]$ replies on channel $c_{\widetilde{j}}$ with the result r of the function, which is forwarded to channel $d_{\widetilde{j}}$. The relay process also allows calling several times the same function with the same values of $\widetilde{j}$ and $\widetilde{i}$, in which case it always returns the same result r. (We make sure in the following that when a function is called several times, the calls all use the same arguments.) Since L and R are required to have the same structure by Hypothesis H2, $\mathrm{relay}(L) = \mathrm{relay}(R)$.

We introduce the following auxiliary definitions, which allow us to define the correspondence mapIdx_M from replication indices at M in Q_0 to replication indices at N_M in L:

- For each $M \in \mathcal{M}$ and $k \leq \mathrm{nNewSeq}_M$, we define $\mathrm{count}_\eta(k, M)$ as follows. Let $n_1, \dots, n_l$ be the sequence of bounds of replications above the definition of $z_{kk',M}$ for any k'. Let l' be the length of the longest common prefix of $\mathrm{im}\ \mathrm{index}_k(M)$ and $\mathrm{im}\ \mathrm{index}_{k_0}(M)$ for $k_0 < k$. We define $\mathrm{count}_\eta(k, M) = I_\eta(n_{l'+1}) \times \dots \times I_\eta(n_l)$. We define parameters $\mathrm{count}_{k,M}$ such that $I_\eta(\mathrm{count}_{k,M}) = \mathrm{count}_\eta(k, M)$.

  We define function symbols $\mathrm{num}_{k,M} : [1, n_1] \times \dots \times [1, n_l] \to [1, \mathrm{count}_{k,M}]$ such that $I_\eta(\mathrm{num}_{k,M})(a_1, \dots, a_l) = 1 + (a_{l'+1} - 1) + I_\eta(n_{l'+1}) \times ((a_{l'+2} - 1) + I_\eta(n_{l'+2}) \times \dots + I_\eta(n_{l-1}) \times (a_l - 1))$. Then $\mathrm{num}_{k,M}$ establishes a bijection between the last $l - l'$ components of its argument and its result.

- We define $\mathrm{tot_count}_\eta(j_0, \dots, j_k)$ as the sum of $\mathrm{count}_\eta(k+1, M'')$ for all M'' such that the first $k+1$ elements of $BL(M'')$ are equal to $j_0, \dots, j_k$, counting only once terms M'' that share the first $k+1$ sequences of random variables.

  We set $I_\eta(n_{j_0,\dots,j_k}) = \mathrm{tot_count}_\eta(j_0, \dots, j_k)$, where $n_{j_0,\dots,j_k}$ is the bound of the replication at the root of

$L_{j_0,\dots,j_k}$ in L. The value of $I_\eta(n_{j_0,\dots,j_k})$ is then large enough so that there is always an available copy of the desired replicated process when we need to execute one.

The replication at the root of $\text{relay}(L_{j_0,\dots,j_k})^{j_0,\dots,j_k}_{i_1,\dots,i_k}$ is also bounded by $n_{j_0,\dots,j_k}$. The other replication of $\text{relay}(L_{j_0,\dots,j_k})^{j_0,\dots,j_k}_{i_1,\dots,i_k}$ is bounded by n', where $I_\eta(n')$ is the sum for all $M \in \mathcal{M}$ of $I_\eta(n_1) \times \ldots \times I_\eta(n_l)$ where $n_1, \dots, n_l$ is the sequence of bounds of replications above M in Q_0.

- We order the term occurrences in $\mathcal{M}$ arbitrarily, with a total ordering. Let $\text{start}_\eta(k, M)$ be defined as follows. Let M' the smallest (in the chosen ordering of $\mathcal{M}$) term occurrence of $\mathcal{M}$ that shares the first k sequences of random variables with M. Then $\text{start}_\eta(k, M)$ is the sum of $\text{count}_\eta(k, M'')$ for all M'' smaller than M' such that the first k elements of $BL(M'')$ are equal to the first k elements of $BL(M')$, counting only once terms M'' that share the first k sequences of random variables.

  We define function symbols $\text{addstart}_{k,M} : [1, \text{count}_{k,M}] \to [1, n_{j_0,\dots,j_k}]$ where $BL(M) = (j_0, \dots, j_k, \dots)$, such that $I_\eta(\text{addstart}_{k,M})(a) = \text{start}_\eta(k, M) + a$.

- Let us define $\text{convindex}(k, M)$ as the sequence of terms

$$\begin{aligned}&\text{convindex}(k, M) = \\ &\quad (\text{addstart}_{1,M}(\text{num}_{1,M}(\text{im index}_1(M))), \\ &\qquad \dots, \text{addstart}_{k,M}(\text{num}_{k,M}(\text{im index}_k(M))))\end{aligned}$$

  This sequence of terms implements the function mapIdx_M mentioned in the explanation of the transformation, in Section 3.2. More precisely, $\text{mapIdx}_M(\widetilde{a}) = \text{convindex}(l, M)\{\widetilde{a}/\widetilde{i}\}$, where $\widetilde{i}$ is the sequence of current replication indices at M and $l = \text{nNewSeq}_M$.

Then we define $C = (\mathsf{newChannel}\ c_{\widetilde{j}}; \mathsf{newChannel}\ d_{\widetilde{j}};)_{\widetilde{j}}([\,] \mid \text{relay}(L) \mid Q_0'')$ where the process Q_0'' is defined from Q_0 as follows:

- When $x \in S$, we replace its definition $\mathsf{new}\ x : T; Q$ with $\mathsf{let}\ x : T = cst$ in Q for some constant cst.

- For each $M \in \mathcal{M}$, let $P_M = C_M[M]$ be the smallest subprocess of Q_0 containing M. Let $l = \text{nNewSeq}_M$ and $m = \text{nInput}_M$. Let $BL(M) = (j_0, \dots, j_l)$. Let $d_M = d_{j_0,\dots,j_l}[\text{convindex}(l, M)]$ and for all $k \le l$, $d_{M,k} = d_{j_0,\dots,j_{k-1}}[\text{convindex}(k, M)]$. We replace P_M with $\overline{d_{M,1}}\langle\rangle; d_{M,1}(); \dots \overline{d_{M,l}}\langle\rangle; d_{M,l}(); \overline{d_M}\langle \sigma_M x_{1,M}, \dots, \sigma_M x_{m,M}\rangle; d_M(y : \mathit{bitstring}); C_M[y]$ where y is a fresh variable.

Instead of evaluating the terms $M \in \mathcal{M}$ directly as in Q_0, Q_0'' sends messages to the relay process $\text{relay}(L)$, which will then forward them to $[\![L]\!]$ in $C[[\![L]\!]]$ and to $[\![R]\!]$ in $C[[\![R]\!]]$.

Lemma 11 $Q_0 \approx_0^V C[[\![L]\!]]$

Proof The bounds of replications of $[\![L]\!]$ and $\text{relay}(L)$ have been defined above. As outlined in the proof of Proposition 6, the length of all bitstrings manipulated by Q_0 is polynomial in η. We can therefore define $\text{maxlen}_\eta(c_{\widetilde{j}})$ to be a polynomial large enough so that messages sent on $c_{\widetilde{j}}$ by $C[[\![L]\!]]$ are never truncated. We define $\text{maxlen}_\eta(d_{\widetilde{j}}) = \text{maxlen}_\eta(c_{\widetilde{j}})$; then messages on $d_{\widetilde{j}}$ are never truncated.

Let C' be any evaluation context acceptable for Q_0, $C[[\![L]\!]]$, V. We relate traces of $C'[Q_0]$ and of $C'[C[[\![L]\!]]]$ as follows.

We assume that the channels $c_{\widetilde{j}}$ and $d_{\widetilde{j}}$ do not occur in C' and Q_0, and that during reductions (NewChannel), these channels are substituted by themselves. (This is easy to guarantee by renaming; this assumption simplifies notations in the proof.)

We write $M =_E M'$ when $E, M \Downarrow a$ and $E, M' \Downarrow a$ for some bitstring a. We denote by $k\text{-th}(\widetilde{i})$ the k-th component of the tuple $\widetilde{i}$, and by $|\widetilde{i}|$ the number of elements of the tuple $\widetilde{i}$.

We define a relation between variables of S in Q_0 and variables y defined by new in $[\![L]\!]$: we say that $y[a_1, \dots, a_j] \xrightarrow{\text{var}}_E \text{varImL}(y, M)[\widetilde{a'}]$ when for all $k' \le j$, $E, \text{addstart}_{k',M}(\text{num}_{k',M}(\text{im}\ (\rho_{j-1}(M) \circ \ldots \circ \rho_{k'}(M))\{\widetilde{a'}/\widetilde{i}\})) \Downarrow a_{k'}$, where $\widetilde{i} \le \widetilde{n}$ are the current replication indices at the definition of $\text{varImL}(y, M)$ with their associated bounds, and for all $l \le |\widetilde{i}|$, $l\text{-th}(\widetilde{a'}) \in [1, I_\eta(l\text{-th}(\widetilde{n}))]$. (Note that $\xrightarrow{\text{var}}$ depends on η.)

We show that the relation $\xrightarrow{\text{var}}_E$ is a (partial) function, that is, if $y[a_1, \dots, a_j] \xrightarrow{\text{var}}_E M_V$ and $y[a_1, \dots, a_j] \xrightarrow{\text{var}}_E M_V'$ then $M_V = M_V'$. Assume that $y[a_1, \dots, a_j] \xrightarrow{\text{var}}_E z'[\widetilde{a'}]$ and $y[a_1, \dots, a_j] \xrightarrow{\text{var}}_E z''[\widetilde{a''}]$. Then

- we have $z' = \text{varImL}(y, M')$ and

$$\begin{aligned}&E, \text{addstart}_{k',M'}(\text{num}_{k',M'}(\text{im}\ (\rho_{j-1}(M') \circ \\ &\quad \ldots \circ \rho_{k'}(M'))\{\widetilde{a'}/\widetilde{i'}\})) \Downarrow a_{k'} \text{ for all } k' \le j\end{aligned}$$

  where $\widetilde{i'} \le \widetilde{n'}$ are the current replication indices at the definition of z' with their associated bounds, and for all $l \le |\widetilde{i'}|$, $l\text{-th}(\widetilde{a'}) \in [1, I_\eta(l\text{-th}(\widetilde{n'}))]$,

- we have $z'' = \text{varImL}(y, M'')$ and

$$\begin{aligned}&E, \text{addstart}_{k',M''}(\text{num}_{k',M''}(\text{im}\ (\rho_{j-1}(M'') \circ \\ &\quad \ldots \circ \rho_{k'}(M''))\{\widetilde{a''}/\widetilde{i''}\})) \Downarrow a_{k'} \text{ for all } k' \le j\end{aligned}$$

  where $\widetilde{i''} \le \widetilde{n''}$ are the current replication indices at the definition of z'' with their associated bounds, and for all $l \le |\widetilde{i''}|$, $l\text{-th}(\widetilde{a''}) \in [1, I_\eta(l\text{-th}(\widetilde{n''}))]$.

For all terms M'', we have either $\text{start}_\eta(k', M'') \le \text{start}_\eta(k', M')$ or $\text{start}_\eta(k', M'') \ge \text{start}_\eta(k', M') + \text{count}_\eta(k', M')$ since $\text{start}_\eta(k', M'')$ is computed by adding $\text{count}_\eta(k', M_3)$ for some terms M_3 in a fixed order. Moreover, $\text{num}_{k',M'}(\ldots)$ evaluates to a bitstring in $[1, \text{count}_\eta(k', M')]$. Therefore, $\text{start}_\eta(k', M'') \le \text{start}_\eta(k', M')$. By symmetry, $\text{start}_\eta(k', M'') \ge \text{start}_\eta(k', M')$. So we have for all $k' \le j$, $\text{start}_\eta(k', M') = \text{start}_\eta(k', M'')$ and $\text{num}_{k',M'}(\text{im}\ (\rho_{j-1}(M') \circ \ldots \circ \rho_{k'}(M'))\{\widetilde{a'}/\widetilde{i'}\}) =_E \text{num}_{k',M''}(\text{im}\ (\rho_{j-1}(M'') \circ \ldots \circ \rho_{k'}(M''))\{\widetilde{a''}/\widetilde{i''}\})$. Since $\text{start}_\eta(j, M') = \text{start}_\eta(j, M'')$, by definition of start_η, M' shares the first j sequences of random variables with M''. Since y has j indices, y is defined under j replications in L, so $\text{varImL}(y, M') = \text{varImL}(y, M'')$, that is, $z' = z''$. So

$|\widetilde{a'}| = |\widetilde{a''}|$. By Hypothesis H'4.2, $\rho_{k'}(M') = \rho_{k'}(M'')$ for all $k' < j$. By definition of num, $I_\eta(\mathrm{num}_{k',M'}) = I_\eta(\mathrm{num}_{k',M''})$ for all $k' \leq j$.

We show by induction on k' that if for all $k'' \leq k'$, $\mathrm{num}_{k'',M'}(\mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a'}/\widetilde{i'}\}) =_E \mathrm{num}_{k'',M'}(\mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a''}/\widetilde{i'}\})$, where $\widetilde{i'} \leq \widetilde{n'}$ are the current replication indices at the definition of $z_{k'_,M'}$ with their associated bounds, and $l\text{-th}(\widetilde{a'}), l\text{-th}(\widetilde{a''}) \in [1, I_\eta(l\text{-th}(\widetilde{n'}))]$, then $\widetilde{a'} = \widetilde{a''}$.

- For $k' = 1$, we assume $\mathrm{num}_{1,M'}(\widetilde{a'}) =_E \mathrm{num}_{1,M'}(\widetilde{a''})$. The longest common prefix of $\mathrm{index}_1(M')$ and $\mathrm{index}_{j''}(M')$ for $j'' < 1$ is empty, since $\mathrm{index}_{j''}(M')$ is defined only for $j'' \geq 1$. So $\mathrm{num}_{1,M'}$ establishes a bijection between the tuples $\widetilde{a'}$ smaller than the current replication bounds at definition of $z_{1_,M'}$ and the interval $[1, \mathrm{count}_\eta(1, M')]$. So $\widetilde{a'} = \widetilde{a''}$.

- For $k' > 1$, we assume that $\mathrm{num}_{k'',M'}(\mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a'}/\widetilde{i'}\}) =_E \mathrm{num}_{k'',M'}(\mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a''}/\widetilde{i'}\})$ for all $k'' \leq k'$. Let $k'_{\mathrm{ind}} < k'$. Let $E, \mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k'_{\mathrm{ind}}}(M'))\{\widetilde{a'}/\widetilde{i'}\} \Downarrow \widetilde{a'}_{\mathrm{ind}}$ and $E, \mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k'_{\mathrm{ind}}}(M'))\{\widetilde{a''}/\widetilde{i'}\} \Downarrow \widetilde{a''}_{\mathrm{ind}}$. By hypothesis, we have for all $k'' \leq k'_{\mathrm{ind}}$, $\mathrm{num}_{k'',M'}(\mathrm{im}\ (\rho_{k'_{\mathrm{ind}}-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a'}_{\mathrm{ind}}/\widetilde{i'}_{\mathrm{ind}}\}) =_E \mathrm{num}_{k'',M'}(\mathrm{im}\ (\rho_{k'_{\mathrm{ind}}-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a''}_{\mathrm{ind}}/\widetilde{i'}_{\mathrm{ind}}\})$ where $\widetilde{i'}_{\mathrm{ind}} \leq \widetilde{n'}_{\mathrm{ind}}$ are the current replication indices at the definition of $z_{k'_{\mathrm{ind}}_,M'}$ with their associated bounds. By induction hypothesis, $\widetilde{a'}_{\mathrm{ind}} = \widetilde{a''}_{\mathrm{ind}}$, so for all $k'' < k'$, $\mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a'}/\widetilde{i'}\} =_E \mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k''}(M'))\{\widetilde{a''}/\widetilde{i'}\}$. For $k'' = k'$, we have $\mathrm{num}_{k',M'}(\widetilde{a'}) =_E \mathrm{num}_{k',M'}(\widetilde{a''})$.

  Let l be the length of the longest common prefix of $\mathrm{im}\ \mathrm{index}_{k'}(M')$ and $\mathrm{im}\ \mathrm{index}_{k''_0}(M')$ for $k''_0 < k'$. Since $\mathrm{index}_{k''_0}(M') = \mathrm{index}_{k'}(M') \circ \rho_{k'-1}(M') \circ \ldots \circ \rho_{k''_0}(M')$, the first l components of $\mathrm{im}\ (\rho_{k'-1}(M') \circ \ldots \circ \rho_{k''_0}(M'))$ are then the first l components of $\widetilde{i'}$, so the first l components of $\widetilde{a'}$ and $\widetilde{a''}$ are equal. Moreover $\mathrm{num}_{k',M'}$ establishes a bijection between the last $|\widetilde{a'}| - l$ components of its argument and the interval $[1, \mathrm{count}_\eta(k', M')]$. So the last $|\widetilde{a'}| - l$ components of $\widetilde{a'}$ and $\widetilde{a''}$ are equal. Hence $\widetilde{a'} = \widetilde{a''}$.

Therefore, we conclude that $\widetilde{a'} = \widetilde{a''}$, so $z'[\widetilde{a'}] = z''[\widetilde{a''}]$.

Next, we show that the function $\xrightarrow{\mathrm{var}}_E$ is injective. If $y'[a'_1, \ldots, a'_{j'}] \xrightarrow{\mathrm{var}}_E z[a_1, \ldots, a_j]$ and $y''[a''_1, \ldots, a''_{j''}] \xrightarrow{\mathrm{var}}_E z[a_1, \ldots, a_j]$, then $z = \mathrm{varImL}(y', M')$ and $z = \mathrm{varImL}(y'', M'')$. By Hypothesis H'4.1, M' and M'' share at least the first $j' = j''$ sequences of random variables and $y' = y''$. By Hypothesis H'4.2, $\rho_{k'}(M') = \rho_{k'}(M'')$ for all $k' < j' = j''$. By definition of addstart and num, $\mathrm{start}_\eta(k', M') = \mathrm{start}_\eta(k', M'')$ and $I_\eta(\mathrm{num}_{k',M'}) = I_\eta(\mathrm{num}_{k',M''})$ for all $k' \leq j' = j''$. Hence $a'_{k'} = a''_{k'}$ for all $k' \leq j' = j''$. So $y'[a'_1, \ldots, a'_{j'}] = y''[a''_1, \ldots, a''_{j''}]$.

For each trace $\mathrm{initConfig}(C'[Q_0]) \to \ldots \to E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ of $C'[Q_0]$ of probability p_m, we show that there exists a trace $\mathrm{initConfig}(C'[C[\llbracket L \rrbracket]]) \to \ldots \to E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ of $C'[C[\llbracket L \rrbracket]]$ of probability $p'_{m'}$ such that

- For all $z \notin S$, $E'_{m'}(z[a'_1, \ldots, a'_{j'}]) = E_m(z[a'_1, \ldots, a'_{j'}])$; for all $z \in S$, $z[a'_1, \ldots, a'_{j'}]$ is in $\mathrm{Dom}(E_m)$ if and only if it is in $\mathrm{Dom}(E'_{m'})$; if y is defined by new in L and $y[a_1, \ldots, a_j] \in \mathrm{Dom}(E'_{m'})$ then there exists M_V such that $y[a_1, \ldots, a_k] \xrightarrow{\mathrm{var}}_{E_m} M_V$ and $M_V \in \mathrm{Dom}(E_m)$ and for all such M_V, $E'_{m'}(y[a_1, \ldots, a_j]) = E_m(M_V)$.

- $P'_{m'}$ is obtained from P_m as Q''_0 from Q_0 (transforming only the occurrences that appear in P_m), $\mathcal{Q}'_{m'} = \mathcal{Q}^1_{m'} \uplus \mathcal{Q}^2_{m'} \uplus \mathcal{Q}^3_{m'}$, where $\mathcal{Q}^1_{m'}$ is obtained from $\mathcal{Q}_m$ as Q''_0 from Q_0 (transforming only the occurrences that appear in $\mathcal{Q}_m$), $\mathcal{Q}^2_{m'}$ is what remains of $\mathrm{relay}(L)$ after partial execution, and $\mathcal{Q}^3_{m'}$ is what remains of $\llbracket L \rrbracket$ after partial execution. More precisely, let

$$\mathrm{relay}(L^{a_1,\ldots,a_k}_{j_0,\ldots,j_k}) = \mathrm{relay}(L_{j_0,\ldots,j_k})^{j_0,\ldots,j_k}_{i_1,\ldots,i_k}\{a_1/i_1, \ldots, a_k/i_k\}$$
$$\llbracket L^{a_1,\ldots,a_k}_{j_0,\ldots,j_k} \rrbracket = \llbracket L_{j_0,\ldots,j_k} \rrbracket^{j_0,\ldots,j_k}_{i_1,\ldots,i_k}\{a_1/i_1, \ldots, a_k/i_k\}$$

  where $i_1, \ldots, i_k$ are the replications indices of L above $L_{j_0,\ldots,j_k}$. These processes correspond respectively to the relay process and to the translation of the subtree $L_{j_0,\ldots,j_k}$ of L, for the value of the replication indices $a_1, \ldots, a_k$. Let $\mathrm{redRepl}(a, !^{i \leq n} P) = P\{a/i\}$. Then $\mathcal{Q}^2_{m'}$ and $\mathcal{Q}^3_{m'}$ are formed as follows:

  - for each $j_0, \ldots, j_{k-1}, a_1, \ldots, a_k$ such that
  $$y_{(j_0,\ldots,j_{k-1}),k'}[a_1, \ldots, a_k] \in \mathrm{Dom}(E'_{m'}),$$
  $\mathcal{Q}^2_{m'}$ contains
  $$d_{j_0,\ldots,j_{k-1}}[a_1, \ldots, a_k]();\overline{d_{j_0,\ldots,j_{k-1}}[a_1, \ldots, a_k]}\langle\rangle$$
  possibly several times.
  - for each $j_0, \ldots, j_{k-1}, a_1, \ldots, a_k$ such that
  $$y_{(j_0,\ldots,j_{k-2}),k''}[a_1, \ldots, a_{k-1}] \in \mathrm{Dom}(E'_{m'}) \text{ and}$$
  $$y_{(j_0,\ldots,j_{k-1}),k'}[a_1, \ldots, a_k] \notin \mathrm{Dom}(E'_{m'}),$$
  $\mathcal{Q}^2_{m'}$ contains $\mathrm{redRepl}(a_k, \mathrm{relay}(L^{a_1,\ldots,a_{k-1}}_{j_0,\ldots,j_{k-1}}))$ and $\mathcal{Q}^3_{m'}$ contains $\mathrm{redRepl}(a_k, \llbracket L^{a_1,\ldots,a_{k-1}}_{j_0,\ldots,j_{k-1}} \rrbracket)$.
  - for each $j_0, \ldots, j_l, a_1, \ldots, a_l$ such that
  $$y_{(j_0,\ldots,j_{l-1}),k'}[a_1, \ldots, a_l] \in \mathrm{Dom}(E'_{m'})$$
  and $L_{j_0,\ldots,j_l}$ is a leaf of L, either $\mathcal{Q}^2_{m'}$ contains $\mathrm{relay}(L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l})$ and $\mathcal{Q}^3_{m'}$ contains $\llbracket L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l} \rrbracket$, or $\mathcal{Q}^2_{m'}$ contains
  $$d_{j_0,\ldots,j_l}[a_1, \ldots, a_l](x_{(j_0,\ldots,j_l),1} : T_{(j_0,\ldots,j_l),1}, \ldots, x_{(j_0,\ldots,j_l),l'} : T_{(j_0,\ldots,j_l),l'});\overline{d_{j_0,\ldots,j_l}[a_1, \ldots, a_l]}\langle r\rangle$$
  with $l' = \mathrm{nInput}_{j_0,\ldots,j_l}$, possibly several times, and there exist $M' \in \mathcal{M}$ and $\widetilde{a'}$ such that $E_m, \mathrm{convindex}(l, M')\{\widetilde{a'}/\widetilde{i'}\} \Downarrow a_1, \ldots, a_l$, $E_m, M'\{\widetilde{a'}/\widetilde{i'}\} \Downarrow r$, and $BL(M') = (j_0, \ldots, j_l)$, where $\widetilde{i'}$ is the sequence of replication indices at M'.

where for each k, a_k is a bitstring in $[1, \mathrm{tot_count}_\eta(j_0, \dots, j_{k-1})]$.

- $\mathcal{C}'_{m'} = \mathcal{C}_m \cup \{c_{\widetilde{j}}, d_{\widetilde{j}} \mid \widetilde{j}\}$.
- $p'_{m'} = p_m \times \prod_{z, a'_1, \dots, a'_{j'}} |I_\eta(T)|$ where T is the type of z and $z \in S, a'_1, \dots, a'_{j'}$ are such that $z[a'_1, \dots, a'_{j'}] \in \mathrm{Dom}(E_m)$ and there exists no $y[a_1, \dots, a_j] \in \mathrm{Dom}(E'_{m'})$ such that $y[a_1, \dots, a_j] \xrightarrow{\mathrm{var}}_{E_m} z[a'_1, \dots, a'_{j'}]$.

Note that the same trace of $C'[C[\llbracket L \rrbracket]]$ corresponds to $\prod_{z, a'_1, \dots, a'_{j'}} |I_\eta(T)|$ traces of $C'[Q_0]$ that differ only by the values of $E_m(z[a'_1, \dots, a'_{j'}])$ for $z \in S, a'_1, \dots, a'_{j'}$ as defined in the last item above.

The proof proceeds by induction on the length m of the trace of $C'[Q_0]$. For the induction step, we distinguish cases depending on the last reduction step of the trace.

- For the initial case, we show by induction on C'' that for all $C'', \mathcal{Q}, \mathcal{C}, \sigma$ such that σ substitutes channel names for channel names without touching $c_{\widetilde{j}}$ and $d_{\widetilde{j}}$, there exist $\mathcal{Q}', \mathcal{C}', \sigma'$ such that σ' substitutes channel names for channel names without touching $c_{\widetilde{j}}$ and $d_{\widetilde{j}}$, $\emptyset, \{C''[\sigma Q_0]\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow^* \emptyset, \{\sigma' Q_0\} \uplus \mathcal{Q}', \mathcal{C}'$, and $\emptyset, \{C''[\sigma C[\llbracket L \rrbracket]]\} \uplus \mathcal{Q}, \mathcal{C} \rightsquigarrow^* \emptyset, \{\sigma' C[\llbracket L \rrbracket]\} \uplus \mathcal{Q}', \mathcal{C}'$. This is obvious when $C'' = [\,]$, with $\sigma' = \sigma$, $\mathcal{Q}' = \mathcal{Q}$, and $\mathcal{C}' = \mathcal{C}$. We show this result by applying (Par) when $C'' = C_1 \mid Q_1$ or $C'' = Q_1 \mid C_1$, and (NewChannel) when $C'' = \mathsf{newChannel}\ c; C_1$.

  So we can apply this result to $C'' = C'$, $\sigma = \mathrm{Id}$, $\mathcal{Q} = \emptyset$, and $\mathcal{C} = \mathrm{fc}(C'[Q_0])$. We have $\mathrm{fc}(C'[Q_0]) = \mathrm{fc}(C'[C[\llbracket L \rrbracket]])$, since $\mathrm{fc}(Q_0) = \mathrm{fc}(Q''_0) = \mathrm{fc}(C[\llbracket L \rrbracket])$. Therefore, there exist $\mathcal{Q}, \mathcal{C}, \sigma$ such that σ substitutes channel names for channel names without touching $c_{\widetilde{j}}$ and $d_{\widetilde{j}}$, $\emptyset, \{C'[Q_0]\}, \mathrm{fc}(C'[Q_0]) \rightsquigarrow^* \emptyset, \{\sigma Q_0\} \uplus \mathcal{Q}, \mathcal{C}$, and

$$\begin{aligned}
&\emptyset, \{C'[C[\llbracket L \rrbracket]]\}, \mathrm{fc}(C'[C[\llbracket L \rrbracket]]) \rightsquigarrow^* \emptyset, \{\sigma C[\llbracket L \rrbracket]\} \uplus \mathcal{Q}, \mathcal{C}\\
&\quad \rightsquigarrow^* \emptyset, \{\sigma Q''_0, \mathrm{relay}(L), \llbracket L \rrbracket\} \uplus \mathcal{Q}, \mathcal{C} \cup \{c_{\widetilde{j}}, d_{\widetilde{j}} \mid \widetilde{j}\}\\
&\qquad\qquad \text{by (NewChannel) and (Par)}\\
&\quad \rightsquigarrow^* \emptyset, \{\sigma Q''_0\} \uplus \mathcal{Q}^2_0 \uplus \mathcal{Q}^3_0 \uplus \mathcal{Q}, \mathcal{C} \cup \{c_{\widetilde{j}}, d_{\widetilde{j}} \mid \widetilde{j}\}\\
&\qquad\qquad \text{by (Par) and (Repl)}
\end{aligned}$$

  where $\mathcal{Q}^2_0 = \{\mathrm{redRepl}(a, \mathrm{relay}(L_{j_0})^{j_0}) \mid j_0, a \in [1, \mathrm{tot_count}_\eta(j_0)]\}$ is what remains from $\mathrm{relay}(L)$ after expansion of parallel compositions and replications and $\mathcal{Q}^3_0 = \{\mathrm{redRepl}(a, \llbracket L_{j_0} \rrbracket^{j_0}) \mid j_0, a \in [1, \mathrm{tot_count}_\eta(j_0)]\}$ is what remains of $\llbracket L \rrbracket$ after expansion of parallel compositions and replications.

  Moreover, $\sigma Q''_0$ is obtained from σQ_0 as Q''_0 from Q_0, and $\mathcal{Q}$ does not contain any occurrence modified when transforming Q_0 into Q''_0, so $\{\sigma Q''_0\} \uplus \mathcal{Q}$ is obtained from $\{\sigma Q_0\} \uplus \mathcal{Q}$ as Q''_0 from Q_0.

  Reducing $\{\sigma Q''_0\} \uplus \mathcal{Q}$ and $\{\sigma Q_0\} \uplus \mathcal{Q}$ by $\rightsquigarrow$ until they are in normal form, we obtain that $\mathrm{reduce}(\emptyset, \{C'[Q_0]\}, \mathrm{fc}(C'[Q_0])) = (\emptyset, \mathcal{Q}_0, \mathcal{C}')$ and $\mathrm{reduce}(\emptyset, \{C'[C[\llbracket L \rrbracket]]\}, \mathrm{fc}(C'[C[\llbracket L \rrbracket]])) = (\emptyset, \mathcal{Q}^1_0 \uplus \mathcal{Q}^2_0 \uplus \mathcal{Q}^3_0, \mathcal{C}' \cup \{c_{\widetilde{j}}, d_{\widetilde{j}} \mid \widetilde{j}\})$, where $\mathcal{Q}^1_0$ is obtained from $\mathcal{Q}_0$ as Q''_0 from Q_0. Therefore, $\mathrm{initConfig}(C'[Q_0])$ and $\mathrm{initConfig}(C'[C[\llbracket L \rrbracket]])$ satisfy the desired invariant.

- When the trace of $C'[Q_0]$ executes $\mathsf{new}\ x[a_1, \dots, a_l] : T$ by (New) for $x \in S$ at step m, the corresponding trace of $C'[C[\llbracket L \rrbracket]]$ executes $\mathsf{let}\ x[a_1, \dots, a_l] : T = \mathit{cst}\ \mathsf{in}$ by (Let) at step m'. This yields $|I_\eta(T)|$ traces of $C'[Q_0]$, one for each value of $E_m(x[a_1, \dots, a_l])$, each with probability $p_m = p_{m-1}/|I_\eta(T)|$. In contrast, this yields a single trace of $C'[C[\llbracket L \rrbracket]]$, with probability $p'_{m'} = p'_{m'-1}$.

  Moreover, there exists no $y[a'_1, \dots, a'_{l'}] \in \mathrm{Dom}(E'_{m'})$ such that $y[a'_1, \dots, a'_{l'}] \xrightarrow{\mathrm{var}}_{E_m} x[a_1, \dots, a_l]$. Otherwise, by the first point of the invariant, before the definition of $x[a_1, \dots, a_l]$, there would exist M_V such that $y[a'_1, \dots, a'_{l'}] \xrightarrow{\mathrm{var}}_{E_{m-1}} M_V$ and $M_V \in \mathrm{Dom}(E_{m-1})$. Since E_m is an extension of E_{m-1}, $y[a'_1, \dots, a'_{l'}] \xrightarrow{\mathrm{var}}_{E_m} M_V$. Since $\xrightarrow{\mathrm{var}}_{E_m}$ is injective, $M_V = x[a_1, \dots, a_l]$. This yields a contradiction, since $M_V \in \mathrm{Dom}(E_{m-1})$ but $x[a_1, \dots, a_l] \notin \mathrm{Dom}(E_{m-1})$ by Invariant 4. (The array cell $x[a_1, \dots, a_l]$ cannot be defined several times in a trace.)

  It is then easy to see that the invariant is satisfied.

- When the trace of $C'[Q_0]$ executes $\sigma_{\mathrm{i}} P_M$ for $M \in \mathcal{M}$, the corresponding trace of $C'[C[\llbracket L \rrbracket]]$ executes

$$\begin{aligned}
&\sigma_{\mathrm{i}}(\overline{d_{M,1}}\langle\rangle; d_{M,1}(); \dots \overline{d_{M,l}}\langle\rangle; d_{M,l}();\\
&\overline{d_M}\langle \sigma_M x_{1,M}, \dots, \sigma_M x_{m,M}\rangle; d_M(y : \mathit{bitstring}); C_M[y])
\end{aligned}$$

  where $\sigma_{\mathrm{i}} = \{\widetilde{a}/\widetilde{i}\}$, $\widetilde{i}$ is the sequence of current replication indices at P_M, and $BL(M) = (j_0, \dots, j_l)$.

  For $k \leq l$, let a_k be such that

$$E_m, \mathrm{addstart}_{k,M}(\mathrm{num}_{k,M}(\sigma_{\mathrm{i}}(\mathrm{im\ index}_k(M))))) \Downarrow a_k$$

  and let $\widetilde{b_k}$ be such that $E_m, \sigma_{\mathrm{i}}(\mathrm{im\ index}_k(M)) \Downarrow \widetilde{b_k}$.

  Let m'_k be the step of the trace of $C'[C[\llbracket L \rrbracket]]$ after executing $\overline{\sigma_{\mathrm{i}} d_{M,k}}\langle\rangle; \sigma_{\mathrm{i}} d_{M,k}()$, where $d_{M,k} = d_{j_0, \dots, j_{k-1}}[\mathrm{convindex}(k, M)]$. We show by induction on k that for all k', $y_{(j_0, \dots, j_{k-1}), k'}[a_1, \dots, a_k] \in \mathrm{Dom}(E'_{m'_k})$ and that the invariant is satisfied at step m'_k except that $\sigma_{\mathrm{i}}(\overline{d_{M,1}}\langle\rangle; d_{M,1}(); \dots; \overline{d_{M,k}}\langle\rangle; d_{M,k}())$ has been removed from $P'_{m'_k}$. Let $z_{kk'} = \mathrm{varImL}(y_{((j_0, \dots, j_{k-1}), k'}, M)$. We have $y_{(j_0, \dots, j_{k-1}), k'}[a_1, \dots, a_k] \xrightarrow{\mathrm{var}}_{E_m} z_{kk'}[\widetilde{b_k}]$. Moreover, $z_{kk'}[\widetilde{b_k}] \in \mathrm{Dom}(E_m)$ since $z_{kk'}[\sigma_{\mathrm{i}}(\mathrm{im\ index}_k(M))]$ occurs in $\sigma_{\mathrm{i}} M$, and $\sigma_{\mathrm{i}} M$ is successfully evaluated in the trace of $C'[Q_0]$. We distinguish two cases:

  - 1) $y_{((j_0, \dots, j_{k-1}), k'}[a_1, \dots, a_k] \in \mathrm{Dom}(E'_{m'_{k-1}})$.

    By the invariant at step m'_{k-1}, $\mathcal{Q}^2_{m'_{k-1}}$ contains $d_{j_0, \dots, j_{k-1}}[a_1, \dots, a_k](); \overline{d_{j_0, \dots, j_{k-1}}[a_1, \dots, a_k]}\langle\rangle$.

    So we can execute $\overline{\sigma_{\mathrm{i}} d_{M,k}}\langle\rangle; \sigma_{\mathrm{i}} d_{M,k}()$ by two (Output) steps, without changing the environment, so $y_{(j_0, \dots, j_{k-1}), k'}[a_1, \dots, a_k] \in \mathrm{Dom}(E'_{m'_k})$ and the invariant is satisfied at step m'_k except that $\sigma_{\mathrm{i}}(\overline{d_{M,1}}\langle\rangle; d_{M,1}(); \dots \overline{d_{M,k}}\langle\rangle; d_{M,k}())$ is removed from $P'_{m'_k}$.

– 2) $y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k] \notin \mathrm{Dom}(E'_{m'_{k-1}})$. By induction hypothesis, $y_{(j_0,\ldots,j_{k-2}),k'}[a_1,\ldots,a_{k-1}] \in \mathrm{Dom}(E'_{m'_{k-1}})$. By the invariant at step m'_{k-1},

$$\mathrm{redRepl}(a_k, \mathrm{relay}(L^{a_1,\ldots,a_{k-1}}_{j_0,\ldots,j_{k-1}})) \in \mathcal{Q}^2_{m'_{k-1}} \text{ and}$$
$$\mathrm{redRepl}(a_k, [\![L^{a_1,\ldots,a_{k-1}}_{j_0,\ldots,j_{k-1}}]\!]) \in \mathcal{Q}^3_{m'_{k-1}}.$$

By (Output) twice, we send an empty message on $d_{j_0,\ldots,j_{k-1}}[a_1,\ldots,a_k]$ and on $c_{j_0,\ldots,j_{k-1}}[a_1,\ldots,a_k]$. By (New), we define $y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k]$ for each k'. We choose $E_m(z_{kk'}[\widetilde{b_k}])$ as value of $y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k]$ (with probability $\frac{1}{|I_\eta(T)|}$ where T is the type of $y_{(j_0,\ldots,j_{k-1}),k'}$). Finally, by (Output) twice, we send an empty message on $c_{j_0,\ldots,j_{k-1}}[a_1,\ldots,a_k]$ and on $d_{j_0,\ldots,j_{k-1}}[a_1,\ldots,a_k]$. Then the invariant is satisfied at step m'_k except that $\sigma_{\mathrm{i}}(\overline{d_{M,1}}\langle\rangle; d_{M,1}(); \ldots \overline{d_{M,k}}\langle\rangle; d_{M,k}())$ is removed from $P'_{m'_k}$. (Note that the probability of the trace of $C'[C[[\![L]\!]]]$ is divided by $\prod_{k'} |I_\eta(T_{(j_0,\ldots,j_{k-1}),k'})|$ where $T_{(j_0,\ldots,j_{k-1}),k'}$ is the type of $y_{(j_0,\ldots,j_{k-1}),k'}[a_1, \ldots, a_k]$. This is what is required by the invariant since $y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k]$ is defined at step m'_k but was not at step m'_{k-1}.)

So $y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k] \in \mathrm{Dom}(E'_{m'_l})$ for all $k \leq l$ and k', and the invariant is satisfied at step m'_l except that $\sigma_{\mathrm{i}}(\overline{d_{M,1}}\langle\rangle; d_{M,1}(); \ldots \overline{d_{M,l}}\langle\rangle; d_{M,l}())$ is removed from $P'_{m'_l}$. Let a be such that $E_m, \sigma_{\mathrm{i}} M \Downarrow a$. Let m'' be the step of the trace of $C'[C[[\![L]\!]]]$ after executing $\sigma_{\mathrm{i}}(\overline{d_M}\langle\sigma_M x_{1,M},\ldots,\sigma_M x_{l',M}\rangle; d_M(y : \mathit{bitstring}))$ with $l' = \mathrm{nInput}_M$. By the invariant, we have two cases:

– 1) $\mathrm{relay}(L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l}) \in \mathcal{Q}^2_{m'_l}$ and $[\![L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l}]\!] \in \mathcal{Q}^3_{m'_l}$.

The process $\sigma_{\mathrm{i}}\overline{d_M}\langle\sigma_M x_{1,M},\ldots,\sigma_M x_{l',M}\rangle$ sends the value of $\sigma_{\mathrm{i}}\sigma_M x_{k',M}$ for $k' \leq l'$ on channel $d_{j_0,\ldots,j_l}[a_1,\ldots,a_l]$. By (Output), this message is received by $\mathrm{relay}(L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l})$, which forwards it by (Output) again to $[\![L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l}]\!]$ on channel $c_{j_0,\ldots,j_l}[a_1,\ldots,a_l]$. On reception of this message by $[\![L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l}]\!]$, $E'_{m''}(x_{(j_0,\ldots,j_l),k'}[a_1,\ldots,a_l])$ is set to the received value, so $E_m, \sigma_{\mathrm{i}}\sigma_M x_{k',M} \Downarrow E'_{m''}(x_{(j_0,\ldots,j_l),k'}[a_1,\ldots,a_l])$ for each $k' \leq l'$. For all $k \leq l$ and k', since $y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots, a_k] \xrightarrow{\mathrm{var}}_{E_m} z_{kk'}[\widetilde{b_k}]$, by the invariant we have $E'_{m'_l}(y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k]) = E_m(z_{kk'}[\widetilde{b_k}])$, so $E'_{m''}(y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k]) = E_m(z_{kk'}[\widetilde{b_k}])$. Moreover, $\sigma_M y_{kk',M} = z_{kk'}[\mathrm{im}\ \mathrm{index}_k(M)]$, so

$$E_m, \sigma_{\mathrm{i}}\sigma_M y_{kk',M} \Downarrow E'_{m''}(y_{(j_0,\ldots,j_{k-1}),k'}[a_1,\ldots,a_k])$$

Therefore, for all variables x of N_M defined under k replications, $E_m, \sigma_{\mathrm{i}}\sigma_M x \Downarrow E'_{m''}(x[a_1,\ldots,a_k])$. Since $M = \sigma_M N_M$, we have $E_m, \sigma_{\mathrm{i}}\sigma_M N_M \Downarrow a$, so $E'_{m''}, N_M\{a_1/i_1,\ldots,a_l/i_l\} \Downarrow a$, where $i_1,\ldots,i_l$ are the replication indices of L above $L_{j_0,\ldots,j_l}$. Hence $[\![L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l}]\!]$ sends back a on channel $c_{j_0,\ldots,j_l}[a_1,\ldots,a_l]$ by (Output), which is forwarded on channel $d_{j_0,\ldots,j_l}[a_1,\ldots,a_l]$ by $\mathrm{relay}(L^{a_1,\ldots,a_l}_{j_0,\ldots,j_l})$ by (Output) again, so a is stored in $y[\widetilde{a}]$ by Q''. Thus $E'_{m''}(y[\widetilde{a}]) = a$.

In order to show that the invariant still holds after this step, we remark that, after these outputs, the relay process makes available the following process

$$d_{j_0,\ldots,j_l}[a_1,\ldots,a_l](x_{(j_0,\ldots,j_l),1} : T_{(j_0,\ldots,j_l),1},\ldots,$$
$$x_{(j_0,\ldots,j_l),l'} : T_{(j_0,\ldots,j_l),l'}); \overline{d_{j_0,\ldots,j_l}[a_1,\ldots,a_l]}\langle a\rangle$$

and we have $E_m, \mathrm{convindex}(l, M)\{\widetilde{a}/\widetilde{i}\} \Downarrow a_1,\ldots, a_l$, $E_m, M\{\widetilde{a}/\widetilde{i}\} \Downarrow a$, and $BL(M) = (j_0,\ldots,j_l)$.

– 2) $d_{j_0,\ldots,j_l}[a_1,\ldots,a_l](x_{(j_0,\ldots,j_l),1} : T_{(j_0,\ldots,j_l),1},\ldots, x_{(j_0,\ldots,j_l),l'} : T_{(j_0,\ldots,j_l),l'}); \overline{d_{j_0,\ldots,j_l}[a_1,\ldots,a_l]}\langle r\rangle \in \mathcal{Q}^2_{m'_l}$ and there exist $M' \in \mathcal{M}$ and $\widetilde{a'}$ such that $E_m, \mathrm{convindex}(l, M')\{\widetilde{a'}/\widetilde{i'}\} \Downarrow a_1,\ldots,a_l$, $E_m, M'\{\widetilde{a'}/\widetilde{i'}\} \Downarrow r$, and $BL(M') = (j_0,\ldots,j_l)$, where $\widetilde{i'}$ is the sequence of current replication indices at M'.

We have $E_m, \mathrm{convindex}(l, M)\{\widetilde{a}/\widetilde{i}\} \Downarrow a_1,\ldots,a_l$ by definition of $a_1,\ldots,a_l$. So

$$\mathrm{convindex}(l, M')\{\widetilde{a'}/\widetilde{i'}\} =_{E_m} \mathrm{convindex}(l, M)\{\widetilde{a}/\widetilde{i}\}$$

so, as shown in the proof that $\xrightarrow{\mathrm{var}}_E$ is a function, $\mathrm{index}_l(M')\{\widetilde{a'}/\widetilde{i'}\} =_{E_m} \mathrm{index}_l(M)\{\widetilde{a}/\widetilde{i}\} =_{E_m} \widetilde{b_l}$ and M' and M share the first l sequences of random variables, that is, all sequences of random variables, or $m_l = 0$ and $M = M'$. Moreover, $BL(M) = BL(M') = (j_0,\ldots,j_l)$, so $N_M = N_{M'}$.

If $m_l = 0$ and $M = M'$, $\widetilde{a'} = \widetilde{a}$, so $E_m, \sigma_{\mathrm{i}} M \Downarrow r$, so $r = a$.

Otherwise, by Hypothesis H′4.3, there exists a term M_0 such that $M = (\mathrm{index}_l(M))M_0$, $M' = (\mathrm{index}_l(M'))M_0$, and M_0 does not contain the current replication indices at M or M'. Then $a =_{E_m} M\{\widetilde{a}/\widetilde{i}\} =_{E_m} M_0\{\widetilde{b_l}/\widetilde{i''}\} =_{E_m} M'\{\widetilde{a'}/\widetilde{i'}\} =_{E_m} r$ where $\widetilde{i''}$ is the sequence of current replication indices at definition of $z_{lk',M}$ for any k'.

Therefore, in all cases, we obtain $E'_{m''}(y[\widetilde{a}]) = a$, so $\sigma_{\mathrm{i}} C_M[y]$ in the trace of $C'[C[[\![L]\!]]]$ executes in the same way as $\sigma_{\mathrm{i}} C_M[M]$ in the trace of $C'[Q_0]$, which yields the desired invariant.

- The other cases are easy: both sides reduce in the same way.

Conversely, we show that all traces of $C'[C[[\![L]\!]]]$ correspond to a trace of $C'[Q_0]$ with the same relation as above. The proof follows a technique similar to the previous proof.

So $\prod_{z,a'_1,\ldots,a'_{j'}} |I_\eta(T)|$ traces of $C'[Q_0]$, each of probability p_m, correspond to one trace of $C'[C[[\![L]\!]]]$ with probability $p'_{m'} = p_m \times \prod_{z,a'_1,\ldots,a'_{j'}} |I_\eta(T)|$. Moreover, for all channels

c and bitstrings $a, E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ executes $\overline{c}\langle a\rangle$ immediately if and only if $E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ executes $\overline{c}\langle a\rangle$ immediately. So $\Pr[C'[Q_0] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C'[C[\llbracket L\rrbracket]] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$. Hence $Q_0 \approx_0^V C[\llbracket L\rrbracket]$. □

Lemma 12 $Q'_0 \approx_0^V C[\llbracket R\rrbracket]$

Proof sketch The proof uses the same technique as the proof of Lemma 11. The main addition is that, in contrast to L, R may contain functional processes that are more complex than just terms. In order to handle them, we need to define a relation between variables of Q'_0 and variables of R defined by let or new in functional processes: when y is such a variable, $y[a_1, \dots, a_l] \xrightarrow{\text{var}}_E \text{varImR}(y, M)[\widetilde{a'}]$ where for all $k \leq l$, $E, \text{addstart}_{k,M}(\text{num}_{k,M}(\text{im index}_k(M)\{\widetilde{a'}/\widetilde{i}\})) \Downarrow a_k$ and $\widetilde{i}$ is the sequence of current replication indices at M. The relation $\xrightarrow{\text{var}}_E$ is not a function for these variables, but we can show that when $y[a_1, \dots, a_l]$ is related to several variables, these variables hold the same value at runtime.

The most delicate case is that of find functional processes

$$FP = \mathsf{find}\ (\textstyle\bigoplus_{j=1}^m \widetilde{u_j} \leq \widetilde{n_j}\ \mathsf{suchthat}\ \mathsf{defined}(z_{j1}[\widetilde{u_{j1}}], \dots, z_{jl_j}[\widetilde{u_{jl_j}}]) \wedge M_j\ \mathsf{then}\ FP_j)\ \mathsf{else}\ FP'$$

where for each k, $\widetilde{u_{jk}}$ is the concatenation of the prefix of the current replication indices of length l'_0 and of a non-empty prefix of $\widetilde{u_j}$. When executing such a find process, $\llbracket R\rrbracket$ tests the value of $z_{jk}[a_1, \dots, a_{l'_1}]$ for all indices of $a_1, \dots, a_{l'_1}$ such that $a_1, \dots, a_{l'_0}$ correspond to a prefix of the current replication indices. Correspondingly, $\text{transf}_{\phi, C_M}(FP)$ tests the values of all variables that are related to $z_{jk}[a_1, \dots, a_{l'_1}]$ by $\xrightarrow{\text{var}}$. □

Lemma 13 *Process Q'_0 satisfies Invariant 1.*

Proof Process Q'_0 satisfies Invariant 1 since all newly created definitions concern fresh variables; for variables of Q'_0 that correspond to variables defined by new or by an input in R, there is a single definition for each of them in Q'_0; for variables of Q'_0 that correspond to variables defined by let in R, there are several definitions only when there are several definitions of these variables in R, and since $\llbracket R\rrbracket$ satisfies Invariant 1, these definitions are in different branches of find (or if) in R, so also in Q'_0. □

Lemma 14 *Process Q'_0 satisfies Invariant 2.*

Proof The only variable accesses created in Q'_0 come from $\text{transf}_{\phi_0, C_M}(FP)$. We show by induction on FP that the only variable accesses created by $\text{transf}_{\phi, C_M}(FP)$ and not guarded by a corresponding find are in $\text{im}\ \phi$. (We do not consider variable accesses in C_M, which already existed in Q_0.) So the only variable accesses created by $\text{transf}_{\phi_0, C_M}(FP_M)$ and not guarded by a corresponding find are in $\text{im}\ \phi_0$. Moreover, variable accesses in $\text{im}\ \phi_0$ are of three kinds:

1. $\text{varImR}(x_{j,M}, M)[i'_1, \dots, i'_{l'}]$ which are defined in P'_M, just above $\text{transf}_{\phi_0, C_M}(FP_M)$.
2. $\text{varImR}(y'_{jk,M}, M)[\text{im index}_j(M)]$ where
   (a) either $\text{nNew}_{j,M} > 0$ and $z_{j1,M}[\text{im index}_j(M)]$ is guaranteed to be defined, since it occurs at this point in the initial process Q_0 which satisfies Invariant 2. By the addition of defined conditions in find and the fact that $z'_{jk,M} = \text{varImR}(y'_{jk,M}, M)$ is defined in Q'_0 where $z_{j1,M}$ was defined in Q_0, this implies that $\text{varImR}(y'_{jk,M}, M)[\text{im index}_j(M)]$ is also defined.
   (b) or $\text{nNew}_{j,M} = 0$, then $\text{im index}_j(M)$ is the sequence of current replication indices at M, and $\text{varImR}(y'_{jk,M}, M)[\text{im index}_j(M)]$ is defined just above P'_M.
3. $\text{varImR}(z, M)[i'_1, \dots, i'_{l'}]$ where z is defined by let in FP_M. Since $\llbracket R\rrbracket$ satisfies Invariant 2, accesses to $z[i_1, \dots, i_l]$ in FP_M occur under the definition of $z[i_1, \dots, i_l]$ in FP_M, so accesses to $\text{varImR}(z, M)[i'_1, \dots, i'_{l'}] = \phi_0(z[i_1, \dots, i_l])$ also occur under their definition in $\text{transf}_{\phi_0, C_M}(FP_M)$.

Therefore, Q'_0 satisfies Invariant 2. □

Lemma 15 *Process Q'_0 satisfies Invariant 3.*

Proof The only newly added variable definitions are let $\text{varImR}(x_{j,M}, M) : T_{j,M} = \sigma_M x_{j,M}$ and new $z'_{jk,M} : T'_{jk,M}$. Each variable $\text{varImR}(x_{j,M}, M)$ has at most one definition in Q'_0. For variables $z'_{jk,M}$, when several of these definitions are added for the same variable $z'_{jk,M}$, they are added in place of the definition(s) of $z_{j1,M}$, so by Hypothesis H′3.1.1, they occur under the same replications, so they all have the same type. Therefore, the type environment for Q'_0 is well-defined.

Assume that $M \in \mathcal{M}$ and $P_M = C_M[M]$ is the smallest process containing M. Let $\mathcal{E}_L$ be the type environment at $P_M = C_M[M]$ in Q_0; let $\mathcal{E}_R$ be the type environment at P'_M in Q'_0; let $\mathcal{E}'_L$ be the type environment at N_M in L; let $\mathcal{E}'_R$ be the type environment at FP_M in R. We know that $\mathcal{E}_L \vdash P_M$, and show that $\mathcal{E}_R \vdash P'_M$. It is then easy to see that Q'_0 is well-typed knowing that Q_0 is well-typed. We note that $\mathcal{E}_R$ is an extension of $\mathcal{E}_L$ with types for variables $\text{varImR}(y'_{jk,M'}, M')$, $\text{varImR}(x_{j,M'}, M')$, and $\text{varImR}(z, M')$ when z is defined by let in $FP_{M'}$, for each $M' \in \mathcal{M}$. By Hypothesis H′3.2, $\mathcal{E}_L \vdash \sigma_M x_{j,M} : T_{j,M}$, so $\mathcal{E}_R \vdash \sigma_M x_{j,M} : T_{j,M}$, since $\mathcal{E}_R$ is an extension of $\mathcal{E}_L$. Then, in order to show $\mathcal{E}_R \vdash P'_M$, it is enough to show $\mathcal{E}_R \vdash \text{transf}_{\phi_0, C_M}(FP_M)$.

We say that ϕ is well-typed when $z[\widetilde{M}] \in \text{Dom}(\phi)$ and $\mathcal{E}'_R \vdash z[\widetilde{M}] : T'$ implies $\mathcal{E}_R \vdash \phi(z[\widetilde{M}]) : T'$.

First, it is easy to show by induction on M' that for all well-typed ϕ, for all M' such that $\mathcal{E}'_R \vdash M' : T$, we have $\mathcal{E}_R \vdash \phi(M') : T$.

Next, we show that for all well-typed ϕ, if $\mathcal{E}'_R \vdash \llbracket FP'\rrbracket_{\widetilde{i}}^{\widetilde{j}}$ and the type of the result of FP' is the type of N_M, then $\mathcal{E}_R \vdash \text{transf}_{\phi, C_M}(FP')$, by induction on FP'.

- If $FP' = M'$, we have to show that $\mathcal{E}_R \vdash C_M[\phi(M')]$. Let T such that $\mathcal{E}_L \vdash M : T$.

  We have $M = \sigma_M N_M$, so if N_M contains a function symbol, $\mathcal{E}'_L \vdash N_M : T$. If $N_M = x_{j,M}$, $M = \sigma_M x_{j,M}$ is of type $T_{j,M}$ by Hypothesis H′3.2, so $T = T_{j,M}$,

hence we also have $\mathcal{E}'_L \vdash N_M : T$. If $N_M = y_{jk,M}$, $M = \sigma_M y_{jk,M} = z_{jk,M}[\text{im index}_j(M)]$ is of type $T_{jk,M}$ by Hypothesis H′3.1.1, so $T = T_{jk,M}$ and we also have $\mathcal{E}'_L \vdash N_M : T$.

By hypothesis, we have then $\mathcal{E}'_R \vdash M' : T$, so $\mathcal{E}_R \vdash \phi(M') : T$. Since $\mathcal{E}_L \vdash C_M[M]$ with $\mathcal{E}_L \vdash M : T$, by a substitution lemma, we conclude that $\mathcal{E}_R \vdash C_M[\phi(M')]$.

- The inductive cases follow easily using $\mathcal{E}'_R \vdash [\![FP']\!]^{\widetilde{j}}_{\widetilde{i}}$ and the property proved above to type terms.

  In the case of a find branch with non-empty defined conditions, we extend ϕ into ϕ' as follows. Let $\widetilde{i'}$ be the sequence of current replication indices at M' and $\widetilde{u'}$ be a sequence formed with a fresh variable for each variable in $\widetilde{i'}$.

  – If $z_k = y'_{jk',M'}$ for some k', then

  $$\phi'(z_k[M_{k1}, \dots, M_{kl'_k}]) = \text{varImR}(z_k, M')[\text{im index}_j(M')\{\widetilde{u'}/\widetilde{i'}\}].$$

  Since $\text{varImR}(z_k, M')$ is defined where $z_{j1,M'}$ is defined, the indices of $\text{varImR}(z_k, M')$ are the indices of $z_{j1,M'}$, so $\text{im index}_j(M')$ is of the suitable type. Moreover, $\widetilde{u'}$ and $\widetilde{i'}$ have the same types, so by a substitution lemma, $\text{im index}_j(M')\{\widetilde{u'}/\widetilde{i'}\}$ is of the suitable type. Moreover z_k in R and $\text{varImR}(z_k, M')$ in Q'_0 are both declared of type $T'_{jk',M'}$, so $\mathcal{E}'_R \vdash z_k[M_{k1}, \dots, M_{kl'_k}] : T'_{jk',M'}$ and $\mathcal{E}_R \vdash \text{varImR}(z_k, M')[\text{im index}_j(M')\{\widetilde{u'}/\widetilde{i'}\}] : T'_{jk',M'}$.

  – If z_k is defined by let or by a function input, $\phi'(z_k[M_{k1}, \dots, M_{kl'_k}]) = \text{varImR}(z_k, M')[\widetilde{u'}]$. $\text{varImR}(z_k, M')$ is declared under the same replications as M', so $\widetilde{u'}$ is of the suitable type. The variables z_k in R and $\text{varImR}(z_k, M')$ in Q'_0 are declared of the same type, so if $\mathcal{E}'_R \vdash z_k[M_{k1}, \dots, M_{kl'_k}] : T'$ then $\mathcal{E}_R \vdash \text{varImR}(z_k, M')[\widetilde{u'}] : T'$.

  So ϕ' is well-typed.

  Moreover, we show that $\mathcal{E}_R \vdash \text{im index}_{j_1}(M')\{\widetilde{u'}/\widetilde{i'}\} = \text{im index}_{j_1}(M) : \mathit{bool}$. We have $z_{j_1k,M} = z_{j_1k,M'}$ since M and M' share the j_1 first sequences of random variables, so $\text{im index}_{j_1}(M')$ and $\text{im index}_{j_1}(M)$ are of the same type, since they are both used as indices of $z_{j_1k,M}$. Since $\widetilde{u'}$ and $\widetilde{i'}$ are of the same type, by a substitution lemma, $\text{im index}_{j_1}(M')\{\widetilde{u'}/\widetilde{i'}\}$ and $\text{im index}_{j_1}(M)$ are of the same type, which yields the desired result.

It is easy to see that ϕ_0 is well-typed. Moreover $\mathcal{E}'_R \vdash [\![FP_M]\!]^{\widetilde{j}}_{\widetilde{i}}$ and the type of the result of FP_M is the type of N_M by Hypothesis H0, so $\mathcal{E}_R \vdash \text{transf}_{\phi_0,C_M}(FP_M)$. □

Proof of Proposition 3 Invariants 1, 2, and 3 have been proved in Lemmas 13, 14, and 15 respectively. Finally, we show that $Q_0 \approx^V Q'_0$. After renaming variables so that V and C do not contain variables of L and R, by Lemmas 1, 11, and 12, $Q_0 \approx^V_0 C[[\![L]\!]] \approx^V C[[\![R]\!]] \approx^V_0 Q'_0$, so by transitivity $Q_0 \approx^V Q'_0$. □

E.5 Proofs for Section 4

Proof of Proposition 4 Let C be an acceptable context for $Q \mid Q_x$, $Q \mid Q'_x$, $\emptyset$. We relate the traces of $C[Q \mid Q_x]$ and $C[Q \mid Q'_x]$ as follows:

- If a trace of $C[Q \mid Q_x]$ never executes the subprocess $\overline{c}\langle x[u_1, \dots, u_m]\rangle$ of Q_x, then we obtain a trace of $C[Q \mid Q'_x]$ with the same probability, by just replacing Q_x with Q'_x and subprocesses of Q_x with the corresponding subprocess of Q'_x.

- Otherwise, the considered trace of $C[Q \mid Q_x]$ executes the subprocess $\overline{c}\langle x[u_1, \dots, u_m]\rangle$ of Q_x exactly once, with $E(u_1) = a_1, \dots, E(u_m) = a_m$, and $E(x[a_1, \dots, a_m]) = a$, where E is the environment when $\overline{c}\langle x[u_1, \dots, u_m]\rangle$ is executed. By hypothesis, the definition of $x[a_1, \dots, a_m]$ in this trace is either a restriction new $x[a_1, \dots, a_m] : T$, or an assignment let $x[a_1, \dots, a_m] : T = z[M_1, \dots, M_l]$ with $E, M_k \Downarrow a'_k$ for all $k \leq l$, and the definition of $z[a'_1, \dots, a'_l]$ in this trace is new $z[a'_1, \dots, a'_l] : T$.

  We build $|I_\eta(T)|$ traces of $C[Q \mid Q'_x]$ from this trace, by choosing any value of $I_\eta(T)$ for the restriction new $x[a_1, \dots, a_m] : T$ or new $z[a'_1, \dots, a'_l] : T$ defined above, and the value a for the restriction new $y : T$ of Q'_x. By definition of S, these traces are the same as the trace of $C[Q \mid Q_x]$ except perhaps for values of variables in S, and for the process Q'_x instead of Q_x. The probability of each of these traces is $1/|I_\eta(T)|$ times the probability of the considered trace of $C[Q \mid Q_x]$, since these traces choose one more random number in $I_\eta(T)$ than the trace of $C[Q \mid Q_x]$.

Moreover, all traces of $C[Q \mid Q'_x]$ are obtained by the previous construction. (To show that, we rebuild a trace of $C[Q \mid Q_x]$ from the trace of $C[Q \mid Q'_x]$ by the reverse construction of the one detailed above.)

For each configuration $E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ of the trace of $C[Q \mid Q_x]$, and corresponding configuration $E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ of the trace of $C[Q \mid Q'_x]$, for all channels c and bitstrings a, $E_m, P_m, \mathcal{Q}_m, \mathcal{C}_m$ executes $\overline{c}\langle a\rangle$ immediately if and only if $E'_{m'}, P'_{m'}, \mathcal{Q}'_{m'}, \mathcal{C}'_{m'}$ executes $\overline{c}\langle a\rangle$ immediately.

Therefore, $\Pr[C[Q \mid Q_x] \rightsquigarrow_\eta \overline{c}\langle a\rangle] = \Pr[C[Q \mid Q'_x] \rightsquigarrow_\eta \overline{c}\langle a\rangle]$, so $Q \mid Q_x \approx_0 Q \mid Q'_x$. □

Proof sketch of Proposition 5 Let C be an acceptable context for $Q \mid Q_x$, $Q \mid Q'_x$, $\emptyset$.

We first exclude traces $\mathcal{T}$ such that $\text{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) = \text{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])$ and $\widetilde{a} \neq \widetilde{a'}$. These traces have negligible probability by hypothesis, since $C[_ \mid Q_x]$ is an acceptable context for Q, 0, $\{x\}$. So this removal does not change the result.

For the remaining traces, when $\widetilde{a} \neq \widetilde{a'}$, $\text{defRestr}_{\mathcal{T}}(x[\widetilde{a}]) \neq \text{defRestr}_{\mathcal{T}}(x[\widetilde{a'}])$, so the definitions of $x[\widetilde{a}]$ and $x[\widetilde{a'}]$ do not come from a single execution of the same restriction. (So $x[\widetilde{a}]$ and $x[\widetilde{a'}]$ are independent random numbers.) Then we can apply a proof similar to that of Proposition 4, except that we replace each tested value of $x[\widetilde{a'}]$ with independent random numbers instead of single one. □

Proof of Lemma 2 Let us prove the result for one-session secrecy. (The proof is essentially the same for secrecy.) The contexts $[\,] \mid Q_x$ and $[\,] \mid Q'_x$ are acceptable contexts for Q, Q', $\{x\}$ (after renaming $u_1, \dots, u_m, y$ so that they do not occur in Q and Q'). We have $Q \approx^{\{x\}} Q'$. So, by Lemma 1, $Q \mid Q_x \approx Q' \mid Q_x$ and $Q \mid Q'_x \approx Q' \mid Q'_x$. Since Q preserves the one-session secrecy of x, $Q \mid Q_x \approx Q \mid Q'_x$. So, by transitivity of $\approx$, $Q' \mid Q_x \approx Q' \mid Q'_x$. Therefore, Q' preserves the one-session secrecy of x. $\square$