Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif

Bruno Blanchet

INRIA Paris-Rocquencourt, France
Bruno.Blanchet@inria.fr

Abstract. After giving general context on the verification of security protocols, we focus on the automatic symbolic protocol verifier ProVerif. This verifier can prove secrecy, authentication, and observational equivalence properties of security protocols, for an unbounded number of sessions of the protocol. It supports a wide range of cryptographic primitives defined by rewrite rules or by equations. The tool takes as input a description of the protocol to verify in a process calculus, an extension of the pi calculus with cryptography. It automatically translates this protocol into an abstract representation of the protocol by Horn clauses, and determines whether the desired security properties hold by resolution on these clauses.

1 Introduction

The verification of security protocols has been a very active research area since the 1990s. The interest of this topic has several motivations. Security protocols are ubiquitous: they are used for e-commerce, wireless networks, credit cards, e-voting, among others. The design of security protocols is notoriously error-prone. This point can be illustrated by attacks found against many published protocols, including the famous attack found by Lowe [59] against the Needham-Schroeder public-key protocol [65] 17 years after its publication. Moreover, security errors cannot be detected by functional testing, since they appear only in the presence of a malicious adversary. These errors can also have serious consequences. Hence, the formal verification or proof of protocols is particularly desirable.

In order to verify protocols, two main models have been considered:

- In the *symbolic model*, often called Dolev-Yao model and due to Needham and Schroeder [65] and Dolev and Yao [46], cryptographic primitives are considered as perfect blackboxes, modeled by function symbols in an algebra of terms, possibly with equations. Messages are terms on these primitives and the adversary can compute only using these primitives.
- In contrast, in the *computational model*, messages are bitstrings, cryptographic primitives are functions from bitstrings to bitstrings, and the adversary is any probabilistic Turing machine. This is the model usually considered by cryptographers.

The symbolic model is an abstract model that makes it easier to build automatic verification tools, and many such tools exist: AVISPA [12], FDR [59], ProVerif [23], Scyther [42], Tamarin [70], for instance. The computational model is closer to the real execution of protocols, but the proofs are more difficult to automate; we refer the reader to [27] for some information on the mechanization of proofs in the computational model. Even though it is closer to reality than the symbolic model, we stress that the computational model is still a model. In particular, it does not take into account side channels, such as timing and power consumption, which may give additional information to an adversary and enable new attacks. Moreover, one often studies specifications of protocols. New attacks may appear when the protocol is implemented, either because the specification has not been faithfully implemented, or because the attacks rely on implementation details that do not appear at the specification level.

In this course, we focus on the verification of specifications of protocols in the symbolic model. Basically, to verify protocols in this case, one computes the set of terms (messages) that the adversary knows. If a message does not belong to this set, then this message is secret. The difficulty is that this set is infinite, for two reasons: the adversary can build terms as large as he wants, and the considered protocol can be executed any number of times. Several approaches can be considered to solve this problem:

- One can bound the size of messages and the number of executions of the protocols. In this case, the state space is finite, and one can apply standard model-checking techniques. This is the approach taken by FDR [59] and in the SATMC [13] back-end of AVISPA [12], for instance.
- If we bound only the number of executions of the protocol, the state space is infinite, but under reasonable assumptions, one can show that the problem of security protocol verification is decidable: protocol insecurity is NP-complete [69]. Basically, the non-deterministic Turing machine guesses an attack and polynomially checks that it is actually an attack against the protocol. There exist practical tools that can verify protocols in this case, using for instance constraint solving as in Cl-AtSe [38] or extensions of model checking as in OFMC [19]; both tools are back-ends of AVISPA [12].
- When the number of executions of the protocol is not bounded, the problem is undecidable [47] for a reasonable model of protocols. Hence, there exists no automatic tool that always terminates and solves this problem. However, there are several approaches that can tackle an undecidable problem:
  - One can rely on help from the user. This is done for example using the interactive theorem prover Isabelle [67], Tamarin [70], which just requires the user to give a few lemmas to help the tool, or Cryptyc [50], which relies on typing with type annotations.
  - One can have incomplete tools, which sometimes answer “I don’t know” but succeed on many practical examples. For instance, one can use abstractions based on tree-automata to represent the knowledge of the adversary [64, 33].
  - One can allow non-termination, as in Maude-NPA [62, 48].

ProVerif uses an abstract representation of protocols by Horn clauses, in the line of ideas by Weidenbach [71], which is more precise than tree-automata because it keeps relational information on messages. However, using this approach, termination is not guaranteed in general.

In this chapter, we will focus on the tool ProVerif. We refer the reader to [28] for a more complete survey of security protocol verification.

2 Structure and Main Features of ProVerif

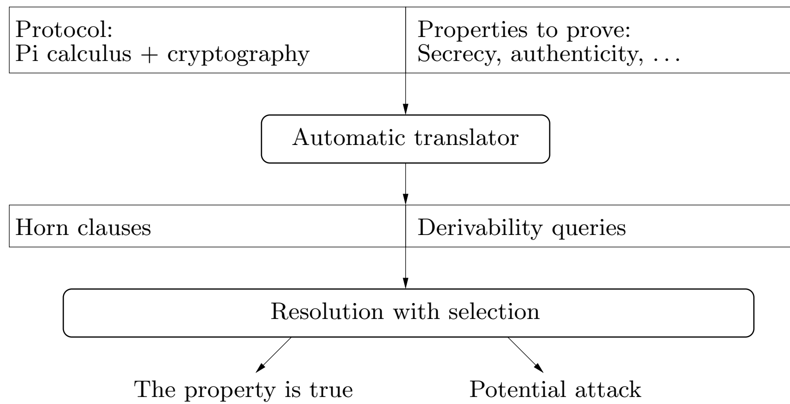

Fig. 1. Structure of ProVerif

The structure of ProVerif is represented in Fig. 1. ProVerif takes as input a model of the protocol in an extension of the pi calculus with cryptography, similar to the applied pi calculus [5] and detailed in the next section. It supports a wide variety of cryptographic primitives, modeled by rewrite rules or by equations. ProVerif also takes as input the security properties that we want to prove. It can verify various security properties, including secrecy, authentication (correspondences), and some observational equivalence properties. It automatically translates this information into an internal representation by Horn clauses: the protocol is translated into a set of Horn clauses, the security properties to prove are translated into derivability queries on these clauses. ProVerif uses an algorithm based on resolution with free selection to determine whether a fact is derivable from the clauses. If the fact is *not* derivable, then the desired security property is proved. If the fact is derivable, then there may be an attack against the considered property: the derivation may correspond to an attack, but it may also correspond to a “false attack”, because the Horn clause representation makes some abstractions. These abstractions are key to the verification of an unbounded number of sessions of protocols.

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\overline{M}\langle N \rangle.P$	output
$M(x).P$	input
$\mathbf{0}$	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu a)P$	restriction
$\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q$	destructor application
$\mathsf{let}\ x = M\ \mathsf{in}\ P$	local definition
$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

Fig. 2. Syntax of the process calculus

Section 3 presents the model of protocols. Section 4 presents the Horn clause representation of protocols and the resolution algorithm. Section 5 gives the translation from the pi calculus model to Horn clauses for secrecy properties. Finally, Sect. 6 summarizes some applications of ProVerif and Sect. 7 concludes.

3 A Formal Model of Security Protocols

This section details the model of protocols used by ProVerif. This calculus was presented in [2]; we adapt that presentation.

3.1 Syntax and Informal Semantics

Figure 2 gives the syntax of terms (data) and processes (programs) of ProVerif's input language. The identifiers a, b, c, k, and similar ones range over names, and x, y, and z range over variables. Names represent atomic data, such as keys and nonces (random numbers). 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 $\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q$ 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 $g(M'_1, \ldots, M'_n) \rightarrow M'$ if and only if there exist a substitution σ and a rewrite rule $g(M_1, \ldots, M_n) \rightarrow M$ 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.)

Using these constructors and destructors, we can represent data structures, such as tuples, and cryptographic operations, for instance as follows:

- $n\mathsf{tuple}(M_1, \ldots, M_n)$ is the tuple of the terms $M_1, \ldots, M_n$, where $n\mathsf{tuple}$ is a constructor. (We sometimes abbreviate $n\mathsf{tuple}(M_1, \ldots, M_n)$ to $(M_1, \ldots, M_n)$.) The n projections are destructors $i\mathsf{th}_n$ for $i \in \{1, \ldots, n\}$, defined by
$$i\mathsf{th}_n(n\mathsf{tuple}(x_1, \ldots, x_n)) \rightarrow x_i$$
- $\mathsf{senc}(M, N)$ is the symmetric (shared-key) encryption of the message M under the key N, where senc is a constructor. The corresponding destructor sdec is defined by
$$\mathsf{sdec}(\mathsf{senc}(x, y), y) \rightarrow x$$
Thus, $\mathsf{sdec}(M', N)$ 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 aenc: $\mathsf{pk}(M)$ builds a public key from a secret key M and $\mathsf{aenc}(M, N)$ encrypts M under the public key N. The corresponding destructor adec is defined by
$$\mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(y)), y) \rightarrow x$$
It decrypts the ciphertext $\mathsf{aenc}(x, \mathsf{pk}(y))$ using the secret key y corresponding to the public $\mathsf{pk}(y)$ used to encrypt this ciphertext.
- As for digital signatures, we may use a constructor sign, and write $\mathsf{sign}(M, N)$ for M signed with the signature key N, and the two destructors check and $\mathsf{getmess}$ with the rewrite rules:
$$\mathsf{check}(\mathsf{sign}(x, y), \mathsf{pk}(y)) \rightarrow x$$
$$\mathsf{getmess}(\mathsf{sign}(x, y)) \rightarrow x$$
The destructor check verifies that the signature $\mathsf{sign}(x, y)$ is a correct signature under the secret key y, using the public key $\mathsf{pk}(y)$. When the signature is correct, it returns the signed message. The destructor $\mathsf{getmess}$ always returns the signed message. (This encoding of signatures assumes that the signature contains the signed message in the clear.)
- 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, modular exponentiation or XOR cannot be directly represented by a constructor or by a destructor. We explain how we can treat some of these primitives in Sect. 5.4.

The other constructs in the syntax of Fig. 2 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. The 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. It makes it possible to represent an unbounded number of executions of the protocol.
- The restriction $(\nu a)P$ creates a new name a, and then executes P. It can model the creation of a fresh key or nonce.
- The local definition $\mathsf{let}\ x = M\ \mathsf{in}\ P$ executes P with x bound to the term M.
- The conditional $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ executes P if M and N reduce to the same term at runtime; otherwise, it executes Q. As usual, we may omit an else branch 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 $M(x).P$, $\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q$, and $\mathsf{let}\ x = M\ \mathsf{in}\ P$. 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 write $\{M_1/x_1, \ldots, M_n/x_n\}$ for the substitution that replaces $x_1, \ldots, x_n$ with $M_1, \ldots, M_n$, respectively. When D is some expression, we write $D\{M_1/x_1, \ldots, M_n/x_n\}$ for the result of applying this substitution to D, but we write σD when the substitution is simply denoted σ. Except when stated otherwise, substitutions always map variables (not names) to expressions.

ProVerif's calculus resembles the applied pi calculus [5]. 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 ProVerif uses destructors instead of the equational theories of the applied pi calculus. (Section 5.4 contains further material on equational theories.) The second difference is that ProVerif has a built-in error-handling construct (the else branch of the destructor application), whereas in the applied pi calculus the error-handling must be done "by hand".

3.2 Example

We use as a running example a simplified version of the Denning-Sacco key distribution protocol [44], omitting certificates and timestamps:

$$\begin{array}{l}\text{Message 1. } A \rightarrow B : \{\{k\}_{sk_A}\}_{pk_B} \\ \text{Message 2. } B \rightarrow A : \{\mathsf{s}\}_k\end{array}$$

This protocol involves two principals A and B. The key sk_A is the secret key of A, pk_A its public key. Similarly, sk_B and pk_B are the secret and public keys of B, respectively. The key k is a fresh session key created by A. A sends this key signed with its private key sk_A and encrypted under the public key of B, pk_B. When B receives this message, B decrypts it and assumes, seeing the signature, that the key k has been generated by A. Then B sends a secret s encrypted under k. Only A should be able to decrypt the message and get the secret s. (The second message is not really part of the protocol, we use it to check if the key k can really be used to exchange secrets between A and B. In fact, there is an attack against this protocol [7], so s will not remain secret.)

This protocol can be encoded by the following process:

$$
\begin{aligned}
&P_0 = (\nu sk_A)(\nu sk_B)\mathsf{let}\ pk_A = \mathsf{pk}(sk_A)\ \mathsf{in}\ \mathsf{let}\ pk_B = \mathsf{pk}(sk_B)\ \mathsf{in}\ \overline{c}\langle pk_A\rangle.\overline{c}\langle pk_B\rangle.\\
&\qquad (P_A(pk_A, sk_A) \mid P_B(pk_B, sk_B, pk_A))\\
&P_A(pk_A, sk_A) = !\ c(x_pk_B).(\nu k)\overline{c}\langle \mathsf{aenc}(\mathsf{sign}(k, sk_A), x_pk_B)\rangle.\\
&\qquad c(x).\mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ 0\\
&P_B(pk_B, sk_B, pk_A) = !\ c(y).\mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in}\\
&\qquad \mathsf{let}\ x_k = \mathsf{check}(y', pk_A)\ \mathsf{in}\ \overline{c}\langle \mathsf{senc}(\mathsf{s}, x_k)\rangle
\end{aligned}
$$

Such a process can be given as input to ProVerif, in an ASCII syntax. This process first creates the secret keys sk_A and sk_B, computes the corresponding public keys pk_A and pk_B, and sends these keys on the public channel c, so that the adversary has these public keys. Then, it runs the processes P_A and P_B in parallel. These processes correspond respectively to the roles of A and B in the protocol. They both start with a replication, which makes it possible to model an unbounded number of sessions of the protocol.

The process P_A first receives on the public channel c the key x_pk_B, which is the public key of A's interlocutor in the protocol. This message is not strictly speaking part of the protocol; it makes it possible for the adversary to choose with whom A is going to execute a session. In a standard session of the protocol, this key is pk_B, but the adversary can also choose another key, for instance one of his own keys. Then, P_A executes the role of A: it creates a fresh key k, signs it with its secret key sk_A, then encrypts this message under x_pk_B, and sends the obtained message on channel c. P_A then expects the second message of the protocol on channel c, stores it in x and decrypts it. If decryption succeeds, the result (normally the secret s) is stored in z.

The process P_B receives the first message of the protocol on channel c, stores it in y, decrypts it with sk_B, and verifies the signature with pk_A. (The signature is verified with the key pk_A of A and not with an arbitrary key chosen by the adversary since B sends the second message $\{\mathsf{s}\}_k$ only if its interlocutor is the honest participant A.) If these verifications succeed, B believes that x_k is a key shared between A and B, and it sends the secret s encrypted under x_k. If the protocol is correct, s should remain secret.

In the above model, we have assumed for simplicity that A and B each play only one role of the protocol. One could easily write a more general model in

which they play both roles, or one could even provide the adversary with an interface that allows it to dynamically create new protocol participants.

3.3 Formal Semantics

The formal semantics of this calculus can be defined in two ways. We can use a structural congruence and a reduction relation (Fig. 3), which is the most common approach, as in [5]. The main semantic rule is (Red I/O), which performs a communication: the message M is sent on channel N by $\overline{N}\langle M\rangle.Q$ and received by $N(x).P$. After the communication, the process Q remains in parallel with P, in which x is replaced with the received message M. In our calculus, one can communicate on channels that are any term.

However, the process is not always exactly of the form required to perform the communication. Therefore, we use the structural congruence relation $\equiv$ to prepare the process in order to perform reductions. The structural congruence says that the parallel composition is associative, commutative, has $\mathbf{0}$ as neutral element. It allows swapping restrictions and modifying the scope of the restriction. As the name says, structural congruence is a congruence, that is, it is an equivalence relation (reflexive, symmetric, and transitive) and it can be applied under parallel compositions and restrictions. The rule (Red $\equiv$) allows one to apply structural congruence before and after reduction.

The rules (Red Destr 1) and (Red Destr 2) correspond respectively to the success and failure of the destructor application. The rule (Red Let) allows one to evaluate a let binding. The rules (Red Cond 1) and (Red Cond 2) correspond respectively to the success or failure of a conditional. The rule (Red Repl) creates a new copy of a replicated process. Finally, the rules (Red Par) and (Red Res) allow one to apply reductions under parallel compositions and restrictions. We identify processes up to renaming of bound names and variables.

We can also define the semantics by a reduction relation on semantic configurations [26], as in Fig. 4. 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 Fig. 4. The rule (Red Res) is the only one that uses renaming. This second semantics guides the reduction of the process more precisely, which simplifies the computation of the evaluation of a process as well as the proofs of some results on ProVerif. In this tutorial, we will focus on this second semantics.

3.4 Definition of Secrecy

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 [46]. Thus, an adversary can be represented by any process that has a set of public names S in its initial knowledge. (Although the

$$
\begin{array}{ll}
P \mid 0 \equiv P & P \equiv Q \Rightarrow P \mid R \equiv Q \mid R \\
P \mid Q \equiv Q \mid P & P \equiv Q \Rightarrow (\nu a)P \equiv (\nu a)Q \\
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & P \equiv P \\
(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P & Q \equiv P \Rightarrow P \equiv Q \\
(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q \;\text{ if } a \notin \mathit{fn}(P) & P \equiv Q,\, Q \equiv R \Rightarrow P \equiv R
\end{array}
$$

$$
\begin{array}{ll}
\overline{N}\langle M\rangle.Q \mid N(x).P \;\rightarrow\; Q \mid P\{M/x\} & \text{(Red I/O)} \\
\mathsf{let}\ x = g(M_1,\ldots,M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow P\{M'/x\} & \text{(Red Destr 1)} \\
\quad \text{if } g(M_1,\ldots,M_n) \rightarrow M' & \\
\mathsf{let}\ x = g(M_1,\ldots,M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \rightarrow Q & \text{(Red Destr 2)} \\
\quad \text{if there exists no } M' \text{ such that } g(M_1,\ldots,M_n) \rightarrow M' & \\
\mathsf{let}\ x = M\ \mathsf{in}\ P \rightarrow P\{M/x\} & \text{(Red Let)} \\
\mathsf{if}\ M = M\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightarrow P & \text{(Red Cond 1)} \\
\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightarrow Q \text{ if } M \neq N & \text{(Red Cond 2)} \\
!P \;\rightarrow\; P \mid\, !P & \text{(Red Repl)} \\
P \;\rightarrow\; Q \;\Rightarrow\; P \mid R \;\rightarrow\; Q \mid R & \text{(Red Par)} \\
P \;\rightarrow\; Q \;\Rightarrow\; (\nu a)P \;\rightarrow\; (\nu a)Q & \text{(Red Res)} \\
P' \equiv P,\, P \;\rightarrow\; Q,\, Q \equiv Q' \;\Rightarrow\; P' \;\rightarrow\; Q' & \text{(Red} \equiv\text{)}
\end{array}
$$

Fig. 3. Structural congruence and reduction

initial knowledge of the adversary contains only names in S, one can give any terms to the adversary by sending them on a channel in S.)

Definition 1. *Let S be a finite set of names. The closed process Q is an S-*adversary *if and only if $\mathit{fn}(Q) \subseteq S$.*

In this chapter, we only consider the property of secrecy. Intuitively, a process P preserves the secrecy of M when M cannot be output on a public channel, in a run of P with any adversary. Formally, we define that a trace outputs M as follows:

Definition 2. *We say that a trace $\mathcal{T} = E_0, \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ outputs M if and only if $\mathcal{T}$ contains a reduction $E, \mathcal{P} \cup \{\, \overline{c}\langle M\rangle.Q, c(x).P \,\} \rightarrow E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\}$ for some E, $\mathcal{P}$, x, P, Q, and $c \in S$.*

We can finally define secrecy:

Definition 3. *The closed process P* preserves the secrecy of *M from S if and only if for any S-adversary Q, for any E_0 containing $\mathit{fn}(P_0) \cup S \cup \mathit{fn}(M)$, for any trace $\mathcal{T} = E_0, \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, the trace $\mathcal{T}$ does not output M.*

This notion of secrecy is similar to that of [1, 32, 35]: a term M is secret if the adversary cannot get it by listening and sending messages, and performing computations.

$E, \mathcal{P} \cup \{ 0 \} \to E, \mathcal{P}$ (Red Nil)

$E, \mathcal{P} \cup \{ !P \} \to E, \mathcal{P} \cup \{ P, !P \}$ (Red Repl)

$E, \mathcal{P} \cup \{ P \mid Q \} \to E, \mathcal{P} \cup \{ P, Q \}$ (Red Par)

$E, \mathcal{P} \cup \{ (\nu a)P \} \to E \cup \{a'\}, \mathcal{P} \cup \{ P\{a'/a\} \}$ (Red Res)
  where $a' \notin E$.

$E, \mathcal{P} \cup \{ \overline{N}\langle M \rangle.Q, N(x).P \} \to E, \mathcal{P} \cup \{ Q, P\{M/x\} \}$ (Red I/O)

$E, \mathcal{P} \cup \{ \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \} \to E, \mathcal{P} \cup \{ P\{M'/x\} \}$ (Red Destr 1)
  if $g(M_1, \ldots, M_n) \to M'$

$E, \mathcal{P} \cup \{ \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{in}\ P\ \mathsf{else}\ Q \} \to E, \mathcal{P} \cup \{ Q \}$ (Red Destr 2)
  if there exists no M' such that $g(M_1, \ldots, M_n) \to M'$

$E, \mathcal{P} \cup \{ \mathsf{let}\ x = M\ \mathsf{in}\ P \} \to E, \mathcal{P} \cup \{ P\{M/x\} \}$ (Red Let)

$E, \mathcal{P} \cup \{ \mathsf{if}\ M = M\ \mathsf{then}\ P\ \mathsf{else}\ Q \} \to E, \mathcal{P} \cup \{ P \}$ (Red Cond 1)

$E, \mathcal{P} \cup \{ \mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q \} \to E, \mathcal{P} \cup \{ Q \}$ (Red Cond 2)
  if $M \neq N$

Fig. 4. Operational semantics

4 The Horn Clause Representation of Protocols

In this section, we introduce the internal representation of protocols used by ProVerif, based on Horn clauses. We also give and prove a resolution algorithm on these clauses.

4.1 Definition of this Representation

$p ::=$	patterns
x	variable
$a[p_1, \ldots, p_n]$	name
$f(p_1, \ldots, p_n)$	constructor application
$F ::= pred(p_1, \ldots, p_n)$	fact
$R ::= F_1 \wedge \ldots \wedge F_n \Rightarrow F$	Horn clause

Fig. 5. Syntax of ProVerif's internal protocol representation

Internally, ProVerif translates the protocol into a representation by a set of Horn clauses; the syntax of these clauses is given in Fig. 5. In this figure, x

ranges over variables, a over names, f over function symbols, and *pred* over predicate symbols. The patterns p represent messages that are exchanged between participants of the protocol. (Patterns are terms; we use the word patterns to distinguish them from terms of the process calculus.) A variable can represent any pattern. Names represent in particular random numbers. In the process calculus, each principal has the ability of creating new names: fresh names are created at each run of the protocol, and names created in different runs of the protocol are always distinct. In the Horn clause representation, the created names are considered as functions $a[p_1, \ldots, p_n]$ of the messages previously received by the principal that creates the name. Thus, names are distinguished only when the preceding messages are different. As noticed by Martín Abadi (personal communication), this approximation is in fact similar to the approximation done in some type systems (such as [1]): the type of the new name depends on the types in the environment. It is enough to handle many protocols, and can be enriched by adding other parameters to the name. The constructor applications $f(M_1, \ldots, M_n)$ build patterns. A fact $F = \mathit{pred}(p_1, \ldots, p_n)$ expresses a property of the messages $p_1, \ldots, p_n$. Several predicates *pred* can be used but, for a first example, we are going to use a single predicate $\mathsf{attacker}$, such that the fact $\mathsf{attacker}(p)$ means "the attacker may have the message p". A clause $R = F_1 \wedge \ldots \wedge F_n \Rightarrow F$ means that, if all facts $F_1, \ldots, F_n$ are true, then F is also true. A clause with no hypothesis $\Rightarrow F$ is written simply F.

We use illustrate the encoding of a protocol on the example of Sect. 3.2:

$$\begin{array}{l} \text{Message 1. } A \to B : \{\{k\}_{sk_A}\}_{pk_B} \\ \text{Message 2. } B \to A : \{s\}_k \end{array}$$

Representation of the Abilities of the Attacker We first present the encoding of the computation abilities of the attacker. The encoding of the protocol itself will be detailed below.

During its computations, the attacker can apply all constructors and destructors. If f is a constructor of arity n, this leads to the clause:

$$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n)).$$

If g is a destructor, for each rewrite rule $g(M_1, \ldots, M_n) \to M$ in $\mathrm{def}(g)$, we have the clause:

$$\mathsf{attacker}(M_1) \wedge \ldots \wedge \mathsf{attacker}(M_n) \Rightarrow \mathsf{attacker}(M).$$

The destructors never appear in the clauses, they are coded by pattern-matching on their parameters (here $M_1, \ldots, M_n$) in the hypothesis of the clause and generating their result in the conclusion. In the particular case of public-key encryption, this yields:

$$\begin{array}{l} \mathsf{attacker}(m) \wedge \mathsf{attacker}(pk) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(m, pk)), \\ \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(\mathsf{pk}(sk)), \\ \mathsf{attacker}(\mathsf{aenc}(m, \mathsf{pk}(sk))) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(m), \end{array} \tag{1}$$

where the first two clauses correspond to the constructors aenc and pk, and the last clause corresponds to the destructor pdec. When the attacker has an encrypted message $\mathsf{aenc}(m, pk)$ and the decryption key sk, then it also has the cleartext m. (We assume that the cryptography is perfect, hence the attacker can obtain the cleartext from the encrypted message only if it has the key.)

Clauses for signatures (sign, $\mathsf{getmess}$, check) and for shared-key encryption (senc, sdec) are given in Fig. 6.

The clauses above describe the computation abilities of the attacker. Moreover, the attacker initially has the public keys of the protocol participants. Therefore, we add the clauses $\mathsf{attacker}(\mathsf{pk}(sk_A[\,]))$ and $\mathsf{attacker}(\mathsf{pk}(sk_B[\,]))$. We also give a name a to the attacker, that will represent all names it can generate: $\mathsf{attacker}(a[\,])$. In particular, $a[\,]$ can represent the secret key of any dishonest participant, his public key being $\mathsf{pk}(a[\,])$, which the attacker can compute by the clause for constructor pk.

Representation of the Protocol Itself Now, we describe how the protocol itself is represented. We consider that A and B are willing to talk to any principal, A, B but also malicious principals that are represented by the attacker. Therefore, the first message sent by A can be $\mathsf{aenc}(\mathsf{sign}(k, sk_A[\,]), \mathsf{pk}(x))$ for any x. We leave to the attacker the task of starting the protocol with the principal it wants, that is, the attacker will send a preliminary message to A, mentioning the public key of the principal with which A should talk. This principal can be B, or another principal represented by the attacker. Hence, if the attacker has some key $\mathsf{pk}(x)$, it can send $\mathsf{pk}(x)$ to A; A replies with his first message, which the attacker can intercept, so the attacker obtains $\mathsf{aenc}(\mathsf{sign}(k, sk_A[\,]), \mathsf{pk}(x))$. Therefore, we have a clause of the form

$$\mathsf{attacker}(\mathsf{pk}(x)) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(k, sk_A[\,]), \mathsf{pk}(x))).$$

Moreover, a new key k is created each time the protocol is run. Hence, if two different keys $\mathsf{pk}(x)$ are received by A, the generated keys k are certainly different: k depends on $\mathsf{pk}(x)$. The clause becomes:

$$\mathsf{attacker}(\mathsf{pk}(x)) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[\,]), \mathsf{pk}(x))). \tag{2}$$

When B receives a message, he decrypts it with his secret key sk_B, so B expects a message of the form $\mathsf{aenc}(x', \mathsf{pk}(sk_B[\,]))$. Next, B tests whether A has signed x', that is, B evaluates $\mathsf{check}(x', pk_A)$, and this succeeds only when $x' = \mathsf{sign}(y, sk_A[\,])$. If so, he assumes that the key y is only known by A, and sends a secret s (a constant that the attacker does not have a priori) encrypted under y. We assume that the attacker relays the message coming from A, and intercepts the message sent by B. Hence the clause:

$$\mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(y, sk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \mathsf{attacker}(\mathsf{senc}(\mathsf{s}, y)).$$

Remark 1. With these clauses, A cannot play the role of B and vice-versa. In order to model a situation in which all principals play both roles, we can replace

Computation abilities of the attacker:
For each constructor f of arity n:
$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n))$
For each destructor g, for each rewrite rule $g(M_1, \ldots, M_n) \to M$ in def(g):
$\mathsf{attacker}(M_1) \wedge \ldots \wedge \mathsf{attacker}(M_n) \Rightarrow \mathsf{attacker}(M)$
that is

aenc	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(pk) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(m, pk))$
pk	$\mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(\mathsf{pk}(sk))$
pdec	$\mathsf{attacker}(\mathsf{aenc}(m, \mathsf{pk}(sk))) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(m)$
sign	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(\mathsf{sign}(m, sk))$
getmess	$\mathsf{attacker}(\mathsf{sign}(m, sk)) \Rightarrow \mathsf{attacker}(m)$
check	$\mathsf{attacker}(\mathsf{sign}(m, sk)) \wedge \mathsf{attacker}(\mathsf{pk}(sk)) \Rightarrow \mathsf{attacker}(m)$
senc	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(k) \Rightarrow \mathsf{attacker}(\mathsf{senc}(m, k))$
sdec	$\mathsf{attacker}(\mathsf{senc}(m, k)) \wedge \mathsf{attacker}(k) \Rightarrow \mathsf{attacker}(m)$
Name generation:	$\mathsf{attacker}(a[\,])$

Initial knowledge: $\mathsf{attacker}(\mathsf{pk}(sk_A[\,]))$, $\mathsf{attacker}(\mathsf{pk}(sk_B[\,]))$

The protocol:
First message: $\mathsf{attacker}(\mathsf{pk}(x)) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[\,]), \mathsf{pk}(x)))$
Second message: $\mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(y, sk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \mathsf{attacker}(\mathsf{senc}(\mathsf{s}, y))$

Fig. 6. Summary the Horn clause representation of the protocol of Sect. 3.2

all occurrences of $sk_B[\,]$ with $sk_A[\,]$ in the clauses above. Then A plays both roles, and is the only honest principal. A single honest principal is sufficient for proving secrecy properties by [40].

More generally, a protocol that contains n messages is encoded by n sets of clauses. If a principal X sends the ith message, the ith set of clauses contains clauses that have as hypotheses the patterns of the messages previously received by X in the protocol, and as conclusion the pattern of the ith message. There may be several possible patterns for the previous messages as well as for the sent message, in particular when the principal X uses a function defined by several rewrite rules, such as the function exp of Sect. 5.4. In this case, a clause must be generated for each combination of possible patterns. Moreover, the hypotheses of the clauses describe all messages previously received, not only the last one. This is important since in some protocols the fifth message for instance can contain elements received in the first message. The hypotheses summarize the history of the exchanged messages.

Summary To sum up, a protocol can be represented by three sets of Horn clauses, as detailed in Fig. 6 for the protocol of Sect. 3.2:

- Clauses representing the computation abilities of the attacker: constructors, destructors, and name generation.
- Facts corresponding to the initial knowledge of the attacker. In general, there are facts giving the public keys of the participants and/or their names to the attacker.

- Clauses representing the messages of the protocol itself. There is one set of clauses for each message in the protocol. In the set corresponding to the ith message, sent by principal X, the clauses are of the form $\mathsf{attacker}(p_{j_1}) \wedge \ldots \wedge \mathsf{attacker}(p_{j_n}) \Rightarrow \mathsf{attacker}(p_i)$ where $p_{j_1}, \ldots, p_{j_n}$ are the patterns of the messages received by X before sending the ith message, and p_i is the pattern of the ith message.

Approximations The reader can notice that the Horn clause representation of protocols is approximate. Specifically, the number of repetitions of each action is ignored, since Horn clauses can be applied any number of times. So a step of the protocol can be completed several times, as long as the previous steps have been completed at least once between the same principals (even when future steps have already been completed). For instance, consider the following protocol (communicated by Véronique Cortier)

First step: A sends $\{(N_1, M)\}_k, \{(N_2, M)\}_k$
Second step: If A receives $\{(x, M)\}_k$, he replies with x
Third step: If A receives N_1, N_2, he replies with s

where N_1, N_2, and M are nonces. In an exact model, A never sends s, since $\{(N_1, M)\}_k$ or $\{(N_2, M)\}_k$ can be decrypted, but not both. In the Horn clause model, even though the first step is executed once, the second step may be executed twice for the same M (that is, the corresponding clause can be applied twice), so that both $\{(N_1, M)\}_k$ and $\{(N_2, M)\}_k$ can be decrypted, and A may send s. We have a false attack against the secrecy of s.

However, the important point is that the approximations are sound: if an attack exists in a more precise model, such as the applied pi calculus [5] or multiset rewriting [43], then it also exists in the Horn clause representation. This is shown for the applied pi calculus in [2] and for multiset rewriting in [24]. In particular, [24] shows formally that the only approximation with respect to the multiset rewriting model is that the number of repetitions of actions is ignored. Performing approximations enables us to build a much more efficient verifier, which will be able to handle larger and more complex protocols. Another advantage is that the verifier does not have to limit the number of runs of the protocol. The price to pay is that false attacks may be found by the verifier: sequences of clause applications that do not correspond to a protocol run, as illustrated above. False attacks appear in particular for protocols with temporary secrets: when some value first needs to be kept secret and is revealed later in the protocol, the Horn clause model considers that this value can be reused in the beginning of the protocol, thus breaking the protocol. When a false attack is found, we cannot know whether the protocol is secure or not: a real attack may also exist. A more precise analysis is required in this case. Fortunately, the Horn clause representation is precise enough so that false attacks are rare. (This is demonstrated by the experiments, see Sect. 6.)

Secrecy Criterion A basic goal is to determine secrecy properties: for instance, can the attacker get the secret s? That is, can the fact $\mathsf{attacker}(\mathsf{s})$ be derived from the clauses? If $\mathsf{attacker}(\mathsf{s})$ can be derived, the sequence of clauses applied to derive $\mathsf{attacker}(\mathsf{s})$ will lead to the description of an attack. This is the notion of secrecy of Sect. 3.4.

In our running example, $\mathsf{attacker}(\mathsf{s})$ is derivable from the clauses. The derivation is as follows. The attacker generates a fresh name $a[\,]$ (considered as a secret key), it computes $\mathsf{pk}(a[\,])$ by the clause for pk, obtains $\mathsf{aenc}(\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,]), \mathsf{pk}(a[\,]))$ by the clause for the first message. It decrypts this message using the clause for pdec and its knowledge of $a[\,]$, thus obtaining $\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,])$. It reencrypts the signature under $\mathsf{pk}(sk_B[\,])$ by the clause for aenc (using its initial knowledge of $\mathsf{pk}(sk_B[\,])$), thus obtaining $\mathsf{aenc}(\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,]), \mathsf{pk}(sk_B[\,]))$. By the clause for the second message, it obtains $\mathsf{senc}(\mathsf{s}, k[\mathsf{pk}(a[\,])])$. On the other hand, from $\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,])$, it obtains $k[\mathsf{pk}(a[\,])]$ by the clause for $\mathsf{getmess}$, so it can decrypt $\mathsf{senc}(\mathsf{s}, k[\mathsf{pk}(a[\,])])$ by the clause for sdec, thus obtaining s. In other words, the attacker starts a session between A and a dishonest participant of secret key $a[\,]$. It gets the first message $\mathsf{aenc}(\mathsf{sign}(k, sk_A[\,]), \mathsf{pk}(a[\,]))$, decrypts it, reencrypts it under $\mathsf{pk}(sk_B[\,])$, and sends it to B. For B, this message looks like the first message of a session between A and B, so B replies with $\mathsf{senc}(\mathsf{s}, k)$, which the attacker can decrypt since it obtains k from the first message. The obtained derivation corresponds to the known attack against this protocol. In contrast, if we fix the protocol by adding the public key of B in the first message $\{\{(pk_B, k)\}_{sk_A}\}_{pk_B}$, $\mathsf{attacker}(\mathsf{s})$ is not derivable from the clauses, so the fixed protocol preserves the secrecy of s.

Next, we formally define when a given fact can be derived from a given set of clauses. We shall see in the next section how we determine that. Technically, the hypotheses $F_1, \ldots, F_n$ of a clause are considered as a multiset. This means that the order of the hypotheses is irrelevant, but the number of times a hypothesis is repeated is important. (This is not related to multiset rewriting models of protocols: the semantics of a clause does not depend on the number of repetitions of its hypotheses, but considering multisets is necessary in the proof of the resolution algorithm.) We use R for clauses (logic programming *rules*), H for hypothesis, and C for conclusion.

Definition 4 (Subsumption). *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 $\sigma H_1 \subseteq H_2$ (multiset inclusion).*

We write $R_1 \sqsupseteq R_2$ when R_2 can be obtained by adding hypotheses to a particular instance of R_1. In this case, all facts that can be derived by R_2 can also be derived by R_1.

A derivation is defined as follows, as illustrated in Fig. 7.

Definition 5 (Derivability). *Let F be a closed fact, that is, a fact without variable. Let $\mathcal{R}$ be a set of clauses. F is derivable from $\mathcal{R}$ if and only if there exists a derivation of F from $\mathcal{R}$, that 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;*
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 $R \sqsupseteq F_1 \wedge \dots \wedge F_n \Rightarrow F_0$.*
4. *The root has one outgoing edge, labeled by F. The unique son of the root is named the* subroot.

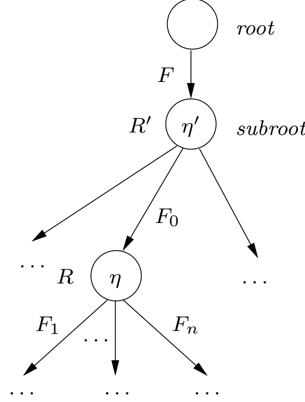

Fig. 7. Derivation of F

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 F_0 can be derived from $F_1, \dots, F_n$ by the clause R. Therefore, there exists a derivation of F from $\mathcal{R}$ if and only if F can be derived from clauses in $\mathcal{R}$ (in classical logic).

4.2 Resolution Algorithm

The internal protocol representation is a set of Horn clauses, and our goal is to determine whether a given fact can be derived from these clauses or not. This is exactly the problem solved by usual Prolog systems. However, we cannot use such systems here, because they would not terminate. For instance, the clause

$$\mathsf{attacker}(\mathsf{aenc}(m, \mathsf{pk}(sk))) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(m)$$

leads to considering more and more complex terms, with an unbounded number of encryptions. We could of course limit arbitrarily the depth of terms to solve the problem, but we can do much better than that.

As detailed below, the main idea is to combine pairs of clauses by resolution, and to guide this resolution process by a selection function: ProVerif's resolution algorithm is resolution with free selection [66, 61, 14]. This algorithm is similar

to ordered resolution with selection, used by [71], but without the ordering constraints.

Notice that, since a term is secret when a fact is *not* derivable from the clauses, soundness in terms of security (if the verifier claims that there is no attack, then there is no attack) corresponds to the completeness of the resolution algorithm in terms of logic programming (if the algorithm claims that a fact is not derivable, then it is not). The resolution algorithm that we use must therefore be complete.

Let us first define resolution: when the conclusion of a clause R unifies with a hypothesis of another (or the same) clause R', resolution infers a new clause that corresponds to applying R and R' one after the other. Formally, resolution is defined as follows:

Definition 6. *Let R and R' be two clauses, $R = H \Rightarrow C$, and $R' = H' \Rightarrow C'$. 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 $R \circ_{F_0} R' = \sigma(H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma C'$. The clause $R \circ_{F_0} R'$ is the result of resolving R' with R upon F_0; it can be inferred from R and R':*

$$\frac{R = H \Rightarrow C \qquad R' = H' \Rightarrow C'}{R \circ_{F_0} R' = \sigma(H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma C'}$$

For example, if R is the clause (2), R' is the clause (1), and the fact F_0 is $F_0 = \mathsf{attacker}(\mathsf{aenc}(m, \mathsf{pk}(sk)))$, then $R \circ_{F_0} R'$ is

$$\mathsf{attacker}(\mathsf{pk}(x)) \wedge \mathsf{attacker}(x) \Rightarrow \mathsf{attacker}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[\,]))$$

with the substitution $\sigma = \{sk \mapsto x, m \mapsto \mathsf{sign}(k[\mathsf{pk}(x)], sk_A[\,])\}$.

We guide the resolution by a selection function:

Definition 7. *A* selection function *sel is a function from clauses to sets of facts, such that $sel(H \Rightarrow C) \subseteq H$. If $F \in sel(R)$, we say that F is selected in R. If $sel(R) = \emptyset$, we say that no hypothesis is selected in R, or that the conclusion of R is selected.*

The resolution algorithm is correct (sound and complete) with any selection function, as we show below. However, the choice of the selection function can change dramatically the behavior of the algorithm. The essential idea of the algorithm is to combine clauses by resolution only when the facts unified in the resolution are selected. We will therefore 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 $\mathsf{attacker}(x)$, with x variable, can be unified will all facts of the form $\mathsf{attacker}(p)$. Therefore, we should avoid selecting them. So a basic selection function is a function sel_0 that satisfies the constraint

$$sel_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if } \forall F \in H, \exists x \text{ variable}, F = \mathsf{attacker}(x) \\ \{F_0\} & \text{where } F_0 \in H \text{ and } \forall x \text{ variable}, F_0 \neq \mathsf{attacker}(x) \end{cases} \tag{3}$$

$\mathsf{saturate}(\mathcal{R}_0) =$
1. $\mathcal{R} \leftarrow \emptyset$.
   For each $R \in \mathcal{R}_0$, $\mathcal{R} \leftarrow \mathsf{elim}(\{R\} \cup \mathcal{R})$.
2. Repeat until a fixpoint is reached
   for each $R \in \mathcal{R}$ such that $sel(R) = \emptyset$,
   for each $R' \in \mathcal{R}$, for each $F_0 \in sel(R')$ such that $R \circ_{F_0} R'$ is defined,
   $\mathcal{R} \leftarrow \mathsf{elim}(\{R \circ_{F_0} R'\} \cup \mathcal{R})$.
3. Return $\{R \in \mathcal{R} \mid sel(R) = \emptyset\}$.

Fig. 8. Resolution algorithm

The resolution algorithm is described in Fig. 8. It transforms the initial set of clauses into a new one that derives the same facts.

The resolution algorithm, $\mathsf{saturate}(\mathcal{R}_0)$, 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 elimination of subsumed clauses by elim: if R' subsumes R, and R and R' are in $\mathcal{R}$, then R is removed by $\mathsf{elim}(\mathcal{R})$.
- The second step is a fixpoint iteration that adds clauses created by resolution. The resolution 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 $\mathcal{R}$. 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, saturate preserves derivability (it is both sound and complete):

Theorem 1 (Correctness of saturate). *Let F be a closed fact. F is derivable from $\mathcal{R}_0$ if and only if it is derivable from* $\mathsf{saturate}(\mathcal{R}_0)$.

This result is proved by transforming a derivation of F from $\mathcal{R}_0$ into a derivation of F from $\mathsf{saturate}(\mathcal{R}_0)$. Basically, when the derivation contains a clause R' with $sel(R') \neq \emptyset$, we replace in this derivation two clauses R, with $sel(R) = \emptyset$, 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 $sel(R') = \emptyset$ so they are in $\mathsf{saturate}(\mathcal{R}_0)$. A detailed proof is given in Sect. 4.3.

Usually, resolution with selection is used for proofs by refutation. That is, the negation of the goal F is added to the clauses, under the form of a clause without conclusion: $F \Rightarrow$. The goal F is derivable if and only if the empty clause "$\Rightarrow$" can be derived. Here, for non-closed goals, we also want to be able to know which instances of the goal can be derived. That is why we prove that the clauses in $\mathsf{saturate}(\mathcal{R}_0)$ derive the same facts as the clauses in $\mathcal{R}_0$.

We can determine which instances of $pred(p_1, \ldots, p_n)$ are derivable, as follows:

Corollary 1. *Let* $\mathsf{solve}_{\mathcal{R}_0}(pred(p_1, \ldots, p_n)) = \{H \Rightarrow pred(p'_1, \ldots, p'_n) \mid H \Rightarrow pred'(p'_1, \ldots, p'_n) \in \mathsf{saturate}(\mathcal{R}'_0)\}$*, where* $pred'$ *is a new predicate and* $\mathcal{R}'_0 = \mathcal{R}_0 \cup \{pred(p_1, \ldots, p_n) \Rightarrow pred'(p_1, \ldots, p_n)\}$*.*

The fact $\sigma pred(p_1, \ldots, p_n)$ *is derivable from* $\mathcal{R}_0$ *if and only if there exists a clause* $H \Rightarrow pred(p'_1, \ldots, p'_n)$ *in* $\mathsf{solve}_{\mathcal{R}_0}(pred(p_1, \ldots, p_n))$ *and a substitution* σ' *such that* $\sigma' pred(p'_1, \ldots, p'_n) = \sigma pred(p_1, \ldots, p_n)$ *and* $\sigma' H$ *is derivable from* $\mathcal{R}'_0$*.*

Proof. The fact $\sigma pred(p_1, \ldots, p_n)$ is derivable from $\mathcal{R}_0$ if and only if $\sigma pred'(p_1, \ldots, p_n)$ is derivable from $\mathcal{R}'_0$, so by Theorem 1, if and only if $\sigma pred'(p_1, \ldots, p_n)$ is derivable from $\mathsf{saturate}(\mathcal{R}'_0)$, so if and only if there exists a clause $H \Rightarrow pred(p'_1, \ldots, p'_n)$ in $\mathsf{solve}_{\mathcal{R}_0}(pred(p_1, \ldots, p_n))$ and a substitution σ' such that $\sigma' pred(p'_1, \ldots, p'_n) = \sigma pred(p_1, \ldots, p_n)$ and $\sigma' H$ is derivable from $\mathsf{saturate}(\mathcal{R}'_0)$, that is, from $\mathcal{R}'_0$. □

In particular, if $\mathsf{solve}_{\mathcal{R}_0}(\mathsf{attacker}(p)) = \emptyset$, then $\mathsf{attacker}(p)$ is not derivable from $\mathcal{R}_0$. Moreover, if $\mathsf{solve}_{\mathcal{R}_0}(\mathsf{attacker}(p))$ is not empty for the selection function sel_0, at least one instance of $\mathsf{attacker}(p)$ is derivable, since H will contain facts of the form $\mathsf{attacker}(x)$, an instance of which is derivable by $\mathsf{attacker}(a[\,])$.

4.3 Proofs

In this section, we detail the proof of Theorem 1. We first need to prove a few preliminary lemmas. The first one shows that two nodes in a derivation can be replaced by one when combining their clauses by resolution.

Lemma 1 (Resolution). *Consider a 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* $\eta' \to \eta$ *is labeled by an instance of* F_0*,* $R \circ_{F_0} R'$ *is defined, and one obtains a 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 Fig. 9. 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 a substitution σ such that $\sigma H' \subseteq H'_1$ and $\sigma C' = C'_1$. Since $F_0 \in H'$, $\sigma F_0 \in H'_1$, so there is an outgoing edge of η' labeled σF_0. 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. So $R \sqsupseteq (H_1 \Rightarrow \sigma F_0)$. By the above choice of distinct variables, we can then extend σ so that $\sigma H \subseteq H_1$ and $\sigma C = \sigma F_0$.

The edge $\eta' \to \eta$ 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

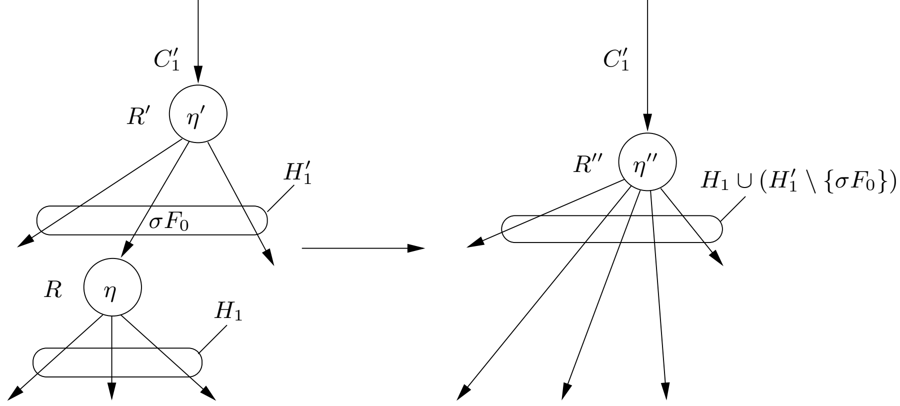

Fig. 9. Merging of nodes of Lemma 1

$\sigma''\sigma'C' = \sigma C' = C'_1$. 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''. □

Lemma 2 (Subsumption). *If a node η of a derivation D is labeled by R, then one obtains a 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'. □

Lemma 3 (Saturation). *At the end of* saturate*, $\mathcal{R}$ satisfies the following properties:*

1. *For all $R \in \mathcal{R}_0$, R is subsumed by a clause in $\mathcal{R}$;*
2. *Let $R \in \mathcal{R}$ and $R' \in \mathcal{R}$. Assume that $sel(R) = \emptyset$ and there exists $F_0 \in sel(R')$ such that $R \circ_{F_0} R'$ is defined. In this case, $R \circ_{F_0} R'$ is subsumed by a clause in $\mathcal{R}$.*

Proof. To prove the first property, let $R \in \mathcal{R}_0$. We show that, after the addition of R to $\mathcal{R}$, R is subsumed by a clause in $\mathcal{R}$.

In the first step of saturate, we execute the instruction $\mathcal{R} \leftarrow \mathsf{elim}(\{R\} \cup \mathcal{R})$. After execution of this instruction, R is subsumed by a clause in $\mathcal{R}$.

Assume that we execute $\mathcal{R} \leftarrow \mathsf{elim}(\{R''\} \cup \mathcal{R})$ for some clause R'' and that, before this execution, R is subsumed by a clause in $\mathcal{R}$, say R'. If R' is removed by this instruction, there exists a clause R'_1 in $\mathcal{R}$ that subsumes R', so by transitivity of subsumption, R'_1 subsumes R, hence R is subsumed by the clause $R'_1 \in \mathcal{R}$ after this instruction. If R' is not removed by this instruction, then R is subsumed by the clause $R' \in \mathcal{R}$ after this instruction.

Hence, at the end of saturate, R is subsumed by a clause in $\mathcal{R}$, which proves the first property.

In order to prove the second property, we just need to notice that the fixpoint is reached at the end of saturate, so $\mathcal{R} = \mathsf{elim}(\{R \circ_{F_0} R'\} \cup \mathcal{R})$. Hence, $R \circ_{F_0} R'$ is eliminated by elim, so it is subsumed by some clause in $\mathcal{R}$. □

Proof of Theorem 1: Assume that F is derivable from $\mathcal{R}_0$ and consider a derivation of F from $\mathcal{R}_0$. We show that F is derivable from $\mathsf{saturate}(\mathcal{R}_0)$.

We consider the value of the set of clauses $\mathcal{R}$ at the end of saturate. For each clause R in $\mathcal{R}_0$, R is subsumed by a clause in $\mathcal{R}$ (Lemma 3, Property 1). So, by Lemma 2, we can replace all clauses R in the considered derivation with a clause in $\mathcal{R}$. Therefore, we obtain a derivation D of F from $\mathcal{R}$.

Next, we build a derivation of F from $\mathcal{R}_1$, where $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$. If D contains a node labeled by a clause not in $\mathcal{R}_1$, we can transform D as follows. Let η' be a lowest node of D labeled by a clause not in $\mathcal{R}_1$. So all sons of η' are labeled by elements of $\mathcal{R}_1$. Let R' be the clause labeling η'. Since $R' \notin \mathcal{R}_1$, $sel(R') \neq \emptyset$. Take $F_0 \in sel(R')$. By Lemma 1, there exists a son of η of η' labeled by R, such that $R \circ_{F_0} R'$ is defined, and we can replace η and η' with a node η'' labeled by $R \circ_{F_0} R'$. Since all sons of η' are labeled by elements of $\mathcal{R}_1$, $R \in \mathcal{R}_1$. Hence $sel(R) = \emptyset$. So, by Lemma 3, Property 2, $R \circ_{F_0} R'$ is subsumed by a clause R'' in $\mathcal{R}$. By Lemma 2, we can relabel η'' with R''. The total number of nodes strictly decreases since η and η' are replaced with a single node η''.

So we obtain a derivation D' of F from $\mathcal{R}$, such that the total number of nodes strictly decreases. Hence, this replacement process terminates. Upon termination, all clauses are in $\mathcal{R}_1$. So we obtain a derivation of F from $\mathcal{R}_1$, 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: if a fact is derivable by applying the clause $R \circ_{F_0} R'$, then it is also derivable by applying R and R'. □

4.4 Optimizations

The resolution algorithm uses several optimizations, in order to speed up resolution. The first two are standard, while the last three are specific to protocols.

Elimination of duplicate hypotheses If a clause contains several times the same hypotheses, the duplicate hypotheses are removed, so that at most one occurrence of each hypothesis remains.

Elimination of tautologies If a clause has a conclusion that is already in the hypotheses, this clause is a tautology: it does not derive new facts. Such clauses are removed.

Elimination of hypotheses $\mathsf{attacker}(x)$ If a clause $H \Rightarrow C$ contains in its hypotheses $\mathsf{attacker}(x)$, where x is a variable that does not appear elsewhere in the clause, then the hypothesis $\mathsf{attacker}(x)$ is removed. Indeed, the attacker always has at least one message, so $\mathsf{attacker}(x)$ is always satisfied for some value of x.

Decomposition of data constructors A data constructor is a constructor f of arity n that comes with associated destructors g_i for $i \in \{1, \ldots, n\}$ defined by $g_i(f(x_1, \ldots, x_n)) \rightarrow x_i$. 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:

$$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n)) \qquad \text{(Rf)}$$

$$\mathsf{attacker}(f(x_1, \ldots, x_n)) \Rightarrow \mathsf{attacker}(x_i) \qquad \text{(Rg)}$$

Therefore, $\mathsf{attacker}(f(p_1, \ldots, p_n))$ is derivable if and only if $\forall i \in \{1, \ldots, n\}$, $\mathsf{attacker}(p_i)$ is derivable. When a fact of the form $\mathsf{attacker}(f(p_1, \ldots, p_n))$ is met, it is replaced with $\mathsf{attacker}(p_1) \wedge \ldots \wedge \mathsf{attacker}(p_n)$. If this replacement is done in the conclusion of a clause $H \Rightarrow \mathsf{attacker}(f(p_1, \ldots, p_n))$, n clauses are created: $H \Rightarrow \mathsf{attacker}(p_i)$ for each $i \in \{1, \ldots, n\}$. This replacement is of course done recursively: if p_i itself is a data constructor application, it is replaced again. The clauses (Rf) and (Rg) for data constructors are left unchanged. (When $\mathsf{attacker}(x)$ 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 $\mathsf{attacker}(x)$ can be selected, which cannot be excluded with certain extensions, these clauses may become necessary for soundness.)

Secrecy assumptions When the user knows that a fact will not be derivable, he can tell it to the verifier. (When this fact is of the form $\mathsf{attacker}(p)$, the user tells that p remains secret.) The tool then removes all clauses which have this fact in their hypotheses. At the end of the computation, the tool checks that the fact is indeed underivable from the obtained clauses. If the user has given erroneous information, an error message is displayed. Even in this case, 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 resolution algorithm. In most cases, the secret keys of the principals cannot be known by the attacker. So, examples of underivable facts are $\mathsf{attacker}(sk_A[\,])$, $\mathsf{attacker}(sk_B[\,])$, ...

For simplicity, the proofs given in Sect. 4.3 do not take into account these optimizations. For a full proof, we refer the reader to [25, Appendix C].

4.5 Termination

In general, the resolution algorithm may not terminate. (The derivability problem is undecidable.) In practice, however, it terminates in most examples.

Blanchet and Podelski have shown that it always terminates on a large and interesting class of protocols, the *tagged protocols* [31]. They consider protocols that use as cryptographic primitives only public-key encryption and signatures with atomic keys, shared-key encryption, message authentication codes, and hash functions. Basically, a protocol is tagged when each application of a cryptographic primitive is marked with a distinct constant tag. It is easy to transform a protocol into a tagged protocol by adding tags. For instance, our example of protocol can be transformed into a tagged protocol, by adding the tags c_0, c_1, c_2 to distinguish the encryptions and signature:

$$\text{Message 1. } A \to B : \{(c_1, \{(c_0, k)\}_{sk_A})\}_{pk_B}$$
$$\text{Message 2. } B \to A : \{(c_2, s)\}_k$$

Adding tags preserves the expected behavior of the protocol, that is, the attack-free executions are unchanged. In the presence of attacks, the tagged protocol may be more secure. Hence, tagging is a feature of good protocol design, as explained e.g. in [7]: the tags are checked when the messages are received; they facilitate the decoding of the received messages and prevent confusions between messages. More formally, tagging prevents type-flaw attacks [53], which occur when a message is taken for another message. However, the tagged protocol is potentially more secure than its untagged version, so, in other words, a proof of security for the tagged protocol does not imply the security of its untagged version.

To illustrate the effect of tagging, we consider the Needham-Schroeder shared-key protocol [65]. The algorithm does not terminate on its original version, which is untagged. It terminates after adding tags. In this protocol, we have two messages of the form:

$$\begin{array}{lll} \text{Message 4.} & B \to A: & \{N_B\}_K \\ \text{Message 5.} & A \to B: & \{N_B - 1\}_K \end{array}$$

where N_B is a nonce. Representing $N_B - 1$ using a function $\mathsf{minusone}(x) = x - 1$, the algorithm does not terminate.

Indeed, message 5 is represented by a clause of the form:

$$H \wedge \mathsf{attacker}(\mathsf{senc}(n, k)) \Rightarrow \mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(n), k))$$

where the hypothesis H describes other messages previously received by A. After some resolution steps, we obtain a clause of the form

$$\mathsf{attacker}(\mathsf{senc}(n, K)) \Rightarrow \mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(n), K)) \qquad \text{(Loop)}$$

for some term K. The fact $\mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(N_B), K))$ is also derived, so a resolution step with (Loop) yields: $\mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}(\mathsf{minusone}(N_B)), K))$. This fact can again be resolved with (Loop), so that we finally have a cycle that derives $\mathsf{attacker}(\mathsf{senc}(\mathsf{minusone}^n(N_B), K))$ for all n.

When tags are added, the rule (Loop) becomes:

$$\mathsf{attacker}(\mathsf{senc}((c_1, n), K)) \Rightarrow \mathsf{attacker}(\mathsf{senc}((c_2, \mathsf{minusone}(n)), K)) \qquad \text{(NoLoop)}$$

and the previous loop is removed because c_2 does not unify with c_1. The fact $\mathsf{attacker}(\mathsf{senc}((c_2, \mathsf{minusone}(N_B)), K))$ is derived, but this does not yield a loop.

Other authors have proved related results: Ramanujan and Suresh [68] have shown that secrecy is decidable for tagged protocols. However, their tagging scheme is stronger since it forbids blind copies. A blind copy happens when a protocol participant sends back part of a message he received without looking at what is contained inside this part. On the other hand, they obtain a decidability result, while [31] obtains a termination result for an algorithm which is sound, efficient in practice, but approximate. Arapinis and Duflot [11] extend this result but still forbid blind copies. Comon-Lundh and Cortier [39] show that an algorithm using ordered binary resolution, ordered factorization and splitting terminates on protocols that blindly copy at most one term in each message. In contrast, the result of [31] puts no limit on the number of blind copies, but requires tagging.

For protocols that are not tagged, heuristics have been designed to adapt the selection function in order to obtain termination more often. We refer the reader to [26, Sect. 8.2] for more details.

It is also possible to obtain termination in all cases at the cost of additional abstractions. For instance, Goubault-Larrecq shows that one can abstract the clauses into clauses in the decidable class $\mathcal{H}_1$ [51], by losing some relational information on the messages.

5 Translation from the Pi Calculus

Given a closed process P_0 in the language of Sect. 3 and a set of names S, ProVerif builds a set of Horn clauses, representing the protocol P_0 in parallel with any S-adversary, in the same style as the clauses presented in the previous section. This translation was originally given in [2]. The clauses use *facts* defined by the following grammar:

$F ::=$	facts
$\quad \mathsf{attacker}(p)$	attacker knowledge
$\quad \mathsf{mess}(p, p')$	message on a channel

The fact $\mathsf{attacker}(p)$ means that the attacker may have p, and the fact $\mathsf{mess}(p, p')$ means that the message p' may appear on channel p. 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 $\mathcal{R}_{P_0,S}$.

5.1 Clauses for the Attacker

The abilities of the attacker are represented by the following clauses:

$$\text{For each } a \in S,\ \mathsf{attacker}(a[\,]) \tag{Init}$$

$$\mathsf{attacker}(b_0[\,]) \tag{Rn}$$

For each constructor f of arity n,
$\text{attacker}(x_1) \wedge \ldots \wedge \text{attacker}(x_n) \Rightarrow \text{attacker}(f(x_1, \ldots, x_n))$ (Rf)

For each destructor g,
for each rewrite rule $g(M_1, \ldots, M_n) \rightarrow M$ in $\text{def}(g)$, (Rg)
$\text{attacker}(M_1) \wedge \ldots \wedge \text{attacker}(M_n) \Rightarrow \text{attacker}(M)$

$\text{mess}(x, y) \wedge \text{attacker}(x) \Rightarrow \text{attacker}(y)$ (Rl)

$\text{attacker}(x) \wedge \text{attacker}(y) \Rightarrow \text{mess}(x, y)$ (Rs)

The clause (Init) represents the initial knowledge of the attacker. The clause (Rn) means that the attacker can generate 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 $\text{def}(g)$ 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 $c \in S$, we can replace all occurrences of $\text{mess}(c[\,], p)$ with $\text{attacker}(p)$ in the clauses. Indeed, these facts are equivalent by the clauses (Rl) and (Rs).

5.2 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 $\text{mess}(p, p')$. The environment ρ maps each variable and name to its associated pattern representation. The sequence H keeps track 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 $H \wedge F$.

$[\![0]\!]\rho H = \emptyset$

$[\![P \mid Q]\!]\rho H = [\![P]\!]\rho H \cup [\![Q]\!]\rho H$

$[\![!P]\!]\rho H = [\![P]\!]\rho H$

$[\![(\nu a)P]\!]\rho H = [\![P]\!](\rho[a \mapsto a[p'_1, \ldots, p'_n]\,])H$
where $H = \text{mess}(p_1, p'_1) \wedge \ldots \wedge \text{mess}(p_n, p'_n)$

$[\![M(x).P]\!]\rho H = [\![P]\!](\rho[x \mapsto x])(H \wedge \text{mess}(\rho(M), x))$

$[\![\overline{M}\langle N\rangle.P]\!]\rho H = [\![P]\!]\rho H \cup \{H \Rightarrow \text{mess}(\rho(M), \rho(N))\}$

$[\![\text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q]\!]\rho H = \bigcup\{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma H)$
$\mid g(p'_1, \ldots, p'_n) \rightarrow p'$ is in $\text{def}(g)$ and (σ, σ') is a most general pair of substitutions such that $\sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho H$

$[\![\mathsf{let}\ x = M\ \mathsf{in}\ P]\!]\rho H = [\![P]\!](\rho[x \mapsto \rho(M)])H$
$[\![\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]\rho H =$

$$\begin{cases} [\![P]\!](\sigma\rho)(\sigma H) \cup [\![Q]\!]\rho H \\ \quad \text{when } \rho(M) \text{ and } \rho(N) \text{ unify and } \sigma \text{ is their most general unifier} \\ [\![Q]\!]\rho H \text{ when } \rho(M) \text{ and } \rho(N) \text{ do not unify} \end{cases}$$

The translation of a process is a set of Horn clauses that express that it may send certain messages.

- 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 is ignored, because all Horn clauses are applicable arbitrarily many times.
- For the restriction, we replace the restricted name a in question with the pattern $a[p'_1, \ldots, p'_n]$, where $p'_1, \ldots, p'_n$ are the previous inputs.
- 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. For a more precise treatment, see [26, Sect. 9.2].
- The local definition let $x = M$ in P is in fact equivalent to let $x = id(M)$ in P else 0, where the destructor id is defined by $id(x) \to x$. The conditional if $M = N$ then P else Q is in fact equivalent to let $x = \mathsf{equal}(M, N)$ in P else Q, where the destructor equal is defined by $\mathsf{equal}(x, x) \to x$. So the translations of these constructs is a particular case of the destructor application. We give them explicitly since they are particularly simple.

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.

5.3 Summary and Correctness

Let $\rho = \{a \mapsto a[\,] \mid a \in \mathit{fn}(P_0)\}$. We define the clauses corresponding to the process P_0 as:

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

Theorem 2 (Correctness of the clauses). *Let P_0 be a closed process. Let M be a closed term and p be the pattern obtained from the term M by replacing all names a with $a[\,]$. If $\mathsf{attacker}(p)$ is not derivable from $\mathcal{R}_{P_0,S}$, then P_0 preserves the secrecy of M from S.*

The proof of this result 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. This technique was introduced in [2] where a similar result is proved. [2] also shows an equivalence between an instance of a generic type system for proving secrecy properties of protocols and the Horn clause verification method. This instance is the most precise instance of this generic type system.

By combining Theorem 2 with Corollary 1, we obtain:

Corollary 2. *Let P_0 be a closed process. Let M be a closed term and p be the pattern obtained from the term M by replacing all names a with $a[\,]$. If $\mathsf{solve}_{\mathcal{R}_{P_0,S}}(\mathsf{attacker}(p)) = \emptyset$, then P_0 preserves the secrecy of M from S.*

5.4 Extension to Equational Theories

ProVerif has been extended to handle primitives defined by equational theories [29]. The term algebra consists of constructors equipped with an equational theory, defined by a finite set of equations. For example, we can model a symmetric encryption scheme in which decryption always succeeds (but may return a meaningless message) by the equations

$$\begin{aligned} \mathsf{sdec}(\mathsf{senc}(x,y),y) &= x \\ \mathsf{senc}(\mathsf{sdec}(x,y),y) &= x \end{aligned} \tag{4}$$

where senc and sdec are constructors. The first equation is standard; the second one avoids that the equality test $\mathsf{senc}(\mathsf{sdec}(M,N),N) = M$ reveals that M is a ciphertext under N: in the presence of the second equation, this equality always holds, even when M is not a ciphertext under N. These equations are satisfied by block ciphers, which are bijective.

We can also model the Diffie-Hellman key agreement [45] using equations. The Diffie-Hellman key agreement relies on the following property of modular exponentiation: $(g^a)^b = (g^b)^a = g^{ab}$ in a cyclic multiplicative subgroup G of $\mathbb{Z}_p^*$, where p is a large prime number and g is a generator of G, and on the assumption that it is difficult to compute g^{ab} from g^a and g^b, without knowing the random numbers a and b (computational Diffie-Hellman assumption), or on the stronger

assumption that it is difficult to distinguish g^a, g^b, g^{ab} from g^a, g^b, g^c without knowing the random numbers a, b, and c (decisional Diffie-Hellman assumption). These properties are exploited to establish a shared key between two participants A and B of a protocol: A chooses randomly a and sends g^a to B; symmetrically, B chooses randomly b and sends g^b to A. A can then compute $(g^b)^a$, since it has a and receives g^b, while B computes $(g^a)^b$. These two values being equal, they can be used to compute the shared key. The adversary, on the other hand, has g^a and g^b but not a and b so by the computational Diffie-Hellman assumption, it cannot compute the key. (This exchange resists passive attacks only; to resist active attacks, we need additional ingredients, for instance signatures.) We can model the Diffie-Hellman key agreement by the equation [5, 4]

$$\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) = \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x) \tag{5}$$

where g is a constant and exp is modular exponentiation. Obviously, this is a basic model: it models the main functional equation but misses many algebraic relations that exist in the group G.

The main idea of our extension to equations is to translate these equations into a set of rewrite rules associated to constructors. For instance, the equations (4) are translated into the rewrite rules

$$\begin{array}{ll} \mathsf{senc}(x, y) \rightarrow \mathsf{senc}(x, y) & \mathsf{sdec}(x, y) \rightarrow \mathsf{sdec}(x, y) \\ \mathsf{senc}(\mathsf{sdec}(x, y), y) \rightarrow x & \mathsf{sdec}(\mathsf{senc}(x, y), y) \rightarrow x \end{array} \tag{6}$$

while the equation (5) is translated into

$$\mathsf{exp}(x, y) \rightarrow \mathsf{exp}(x, y) \qquad \mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) \rightarrow \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x) \tag{7}$$

Intuitively, these rewrite rules allow one, by applying them *exactly once* for each constructor, to obtain the various forms of the terms modulo the considered equational theory.[1] The constructors are then simply evaluated like destructors in the calculus above. With Abadi and Fournet, we have formally defined when a set of rewrite rules models an equational theory, and designed algorithms that compute translate equations into rewrite rules that model them [29, Sect. 5]. Then, each trace in the calculus with equational theory corresponds to a trace in the calculus with rewrite rules, and conversely [29, Lemma 1].[2] We are then reduced to the simpler case in which there are no equations. The main advantage of this technique is that resolution can still use ordinary syntactic unification (instead of having to use unification modulo the equational theory), and therefore remains efficient.

This extension to equations still has limitations: it does not allow us to model associative operations, such as exclusive or, since this would require an infinite

[1] The rewrite rules like $\mathsf{sdec}(x, y) \rightarrow \mathsf{sdec}(x, y)$ are necessary so that sdec always succeeds. Thanks to this rule, the evaluation of $\mathsf{sdec}(M, N)$ succeeds and leaves this term unchanged when M is not of the form $\mathsf{senc}(M', N)$.

[2] More precisely, the inequality tests of (Red Destr 2) must still be performed modulo the equational theory, even in the calculus with rewrite rules.

number of rewrite rules. It may be possible to handle these symbols using unification modulo the equational theory instead of syntactic unification, at the cost of a larger complexity. In the case of a bounded number of sessions, exclusive or is handled in [41, 37] and a more complete theory of modular exponentiation is handled in [36]. A unification algorithm for modular exponentiation is presented in [63]. For an unbounded number of sessions, extensions of the Horn clause approach that can handle XOR and Diffie-Hellman key agreements with more detailed algebraic relations (including equations of the multiplicative group modulo p) have been proposed by Küsters and Truderung: they handle XOR provided one of its two arguments is a constant in the clauses that model the protocol [57] and Diffie-Hellman key agreements provided the exponents are constants in the clauses that model the protocol [58]; they proceed by transforming the initial clauses into richer clauses on which the standard resolution algorithm is applied.

6 Applications

The automatic protocol verifier ProVerif is available at http://proverif.inria.fr/. Even though we focused only on secrecy in this chapter, ProVerif can also verify authentication [26] and some observational equivalence properties [29]. It can also reconstruct attacks against protocols [10] from the Horn clause derivation, when the desired property does not hold. It was successfully applied to many protocols of the literature, to prove secrecy and authentication properties [26]: no false attack was found in the 19 protocols tested in [26]. It was also used in more substantial case studies:

- With Abadi [3], we applied it to the verification of a certified email protocol [6]. We used correspondence properties to prove that the receiver receives the message if and only if the sender has a receipt for the message. (We used simple manual arguments to take into account that the reception of sent messages is guaranteed.) One of the tested versions includes the SSH transport layer in order to establish a secure channel.
- With Abadi and Fournet [4], we studied the JFK protocol (*Just Fast Keying*) [8], which was one of the candidates to the replacement of IKE as key exchange protocol in IPSec. We combined manual proofs and ProVerif to prove correspondences and equivalences.
- With Chaudhuri [30], we studied the secure filesystem Plutus [54] with ProVerif, which allowed us to discover and fix weaknesses of the initial system.
- ProVerif was also used for verifying a certified email web service [60], a certified mailing-list protocol [55], e-voting protocols [56, 15], the ad-hoc routing protocol ARAN (*Authenticated Routing for Adhoc Networks*) [49], and zero-knowledge protocols [16], for instance.

It was also used as a back-end for building other verification tools:

- Bhargavan et al. [21] use it to build the Web services verification tool TulaFale: Web services are protocols that send XML messages; TulaFale translates them into the input format of ProVerif and uses ProVerif to prove the desired security properties.
- Bhargavan et al. [22] use ProVerif for verifying implementations of protocols in F# (a functional language of the Microsoft .NET environment): a subset of F# large enough for expressing security protocols is translated into the input format of ProVerif. The TLS protocol, in particular, was studied using this technique [20].
- Aizatulin et al. [9] use symbolic execution in order to extract ProVerif models from pre-existing protocol implementations in C. This technique currently analyzes a single execution path of the protocol, so it is limited to protocols without branching. An earlier related approach is that of Goubault-Larrecq and Parrennes [52]: they also use the Horn clause method for analyzing implementations of protocols written in C. However, they translate protocols into clauses of the $\mathcal{H}_1$ class and use the $\mathcal{H}_1$ prover by Goubault-Larrecq [51] rather than ProVerif to prove secrecy properties of the protocol.
- Bansal et al. [17] built the Web-spi library which allows one to model web security mechanisms and protocols and verify them using ProVerif.

Canetti and Herzog [34] use ProVerif for verifying protocols in the computational model: they show that, for a restricted class of protocols that use only public-key encryption, a proof in the Dolev-Yao model implies security in the computational model, in the universal composability framework.

7 Conclusion

ProVerif is an automatic protocol verifier that relies on the symbolic model of cryptography. Its main strengths are that it supports a wide range of cryptographic primitives, defined by rewrite rules and equations, that it can prove various security properties, including secrecy, authentication, and some observational equivalences, and that it handles an unbounded number of protocol executions. This is possible thanks to an abstract representation of the protocol by Horn clauses. Its main limitations are that it may fail to prove some security properties that actually hold, and that it may not terminate. However, it is precise and efficient on many practical examples. Other limitations concern the treatment of equations and the class of observational equivalences that it can prove.

ProVerif verifies specifications of protocols in the symbolic model, which can also be seen as a limitation, since the symbolic model abstracts away the details of cryptographic operations, and specifications do not take into account all implementation details. Going further is a topic of active research. Some tools, such as EasyCrypt [18] and CryptoVerif [27], already tackle the more difficult problem of verifying protocols in the computational model. Other tools verify implementations of protocols rather than specifications, some of them by translating the implementation into a ProVerif model, as mentioned in Section 6.

References

1. Abadi, M., Blanchet, B.: Secrecy types for asymmetric communication. Theoretical Computer Science 298(3), 387–415 (Apr 2003), special issue FoSSaCS'01.
2. Abadi, M., Blanchet, B.: Analyzing security protocols with secrecy types and logic programs. Journal of the ACM 52(1), 102–146 (Jan 2005)
3. Abadi, M., Blanchet, B.: Computer-assisted verification of a protocol for certified email. Science of Computer Programming 58(1–2), 3–27 (Oct 2005), special issue SAS'03.
4. Abadi, M., Blanchet, B., Fournet, C.: Just Fast Keying in the pi calculus. ACM TISSEC 10(3), 1–59 (Jul 2007)
5. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL'01. pp. 104–115. ACM Press, New York (Jan 2001)
6. Abadi, M., Glew, N., Horne, B., Pinkas, B.: Certified email with a light on-line trusted third party: Design and implementation. In: 11th International World Wide Web Conference. pp. 387–395. ACM, New York (May 2002)
7. Abadi, M., Needham, R.: Prudent engineering practice for cryptographic protocols. IEEE Transactions on Software Engineering 22(1), 6–15 (Jan 1996)
8. Aiello, W., Bellovin, S.M., Blaze, M., Canetti, R., Ioannidis, J., Keromytis, K., Reingold, O.: Just Fast Keying: Key agreement in a hostile Internet. ACM TISSEC 7(2), 242–273 (May 2004)
9. Aizatulin, M., Gordon, A.D., Jürjens, J.: Extracting and verifying cryptographic models from C protocol code by symbolic execution. In: CCS'11. pp. 331–340. ACM, New York (Oct 2011)
10. Allamigeon, X., Blanchet, B.: Reconstruction of attacks against cryptographic protocols. In: CSFW'05. pp. 140–154. IEEE, Los Alamitos (Jun 2005)
11. Arapinis, M., Duflot, M.: Bounding messages for free in security protocols. In: Arvind, V., Prasad, S. (eds.) FSTTCS'07. LNCS, vol. 4855, pp. 376–387. Springer, Heidelberg (Dec 2007)
12. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P.H., Héam, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganó, L., Vigneron, L.: The AVISPA tool for automated validation of Internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV'05. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (Jul 2005)
13. Armando, A., Compagna, L., Ganty, P.: SAT-based model-checking of security protocols using planning graph analysis. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME'03. LNCS, vol. 2805, pp. 875–893. Springer, Heidelberg (Sep 2003)
14. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, chap. 2, pp. 19–100. North Holland (2001)
15. Backes, M., Hritcu, C., Maffei, M.: Automated verification of remote electronic voting protocols in the applied pi-calculus. In: CSF'08. pp. 195–209. IEEE, Los Alamitos (Jun 2008)
16. Backes, M., Maffei, M., Unruh, D.: Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In: S& P'08. pp. 202–215. IEEE, Los Alamitos (May 2008), technical report version available at http://eprint.iacr.org/2007/289
17. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: CSF'12. pp. 247–262. IEEE, Los Alamitos (Jun 2012)

18. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO'11. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (Aug 2011)
19. Basin, D., Mödersheim, S., Viganò, L.: An on-the-fly model-checker for security protocol analysis. In: Snekkenes, E., Gollman, D. (eds.) ESORICS'03. LNCS, vol. 2808, pp. 253–270. Springer, Heidelberg (Oct 2003)
20. Bhargavan, K., Corin, R., Fournet, C., Zălinescu, E.: Cryptographically verified implementations for TLS. In: CCS'08. pp. 459–468. ACM, New York (Oct 2008)
21. Bhargavan, K., Fournet, C., Gordon, A.: Verifying policy-based security for web services. In: CCS'04. pp. 268–277. ACM, New York (Oct 2004)
22. Bhargavan, K., Fournet, C., Gordon, A., Tse, S.: Verified interoperable implementations of security protocols. In: CSFW'06. pp. 139–152. IEEE, Los Alamitos (Jul 2006)
23. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: CSFW-14. pp. 82–96. IEEE, Los Alamitos (Jun 2001)
24. Blanchet, B.: Security protocols: From linear to classical logic by abstract interpretation. Information Processing Letters 95(5), 473–479 (Sep 2005)
25. Blanchet, B.: Automatic verification of correspondences for security protocols. Report arXiv:0802.3444v1 (2008), available at http://arxiv.org/abs/0802.3444v1
26. Blanchet, B.: Automatic verification of correspondences for security protocols. Journal of Computer Security 17(4), 363–434 (Jul 2009)
27. Blanchet, B.: Mechanizing game-based proofs of security protocols. In: Nipkow, T., Grumberg, O., Hauptmann, B. (eds.) Software Safety and Security - Tools for Analysis and Verification, NATO Science for Peace and Security Series – D: Information and Communication Security, vol. 33, pp. 1–25. IOS Press (May 2012), proceedings of the 2011 MOD summer school
28. Blanchet, B.: Security protocol verification: Symbolic and computational models. In: Degano, P., Guttman, J. (eds.) POST'12. LNCS, vol. 7215, pp. 3–29. Springer, Heidelberg (Mar 2012)
29. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. Journal of Logic and Algebraic Programming 75(1), 3–51 (Feb–Mar 2008)
30. Blanchet, B., Chaudhuri, A.: Automated formal analysis of a protocol for secure file sharing on untrusted storage. In: S& P'08. pp. 417–431. IEEE, Los Alamitos (May 2008)
31. Blanchet, B., Podelski, A.: Verification of cryptographic protocols: Tagging enforces termination. Theoretical Computer Science 333(1-2), 67–90 (Mar 2005), special issue FoSSaCS'03.
32. Bodei, C.: Security Issues in Process Calculi. Ph.D. thesis, Università di Pisa (Jan 2000)
33. Boichut, Y., Kosmatov, N., Vigneron, L.: 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 (Mar 2006)
34. Canetti, R., Herzog, J.: Universally composable symbolic analysis of mutual authentication and key exchange protocols. In: Halevi, S., Rabin, T. (eds.) TCC'06. LNCS, vol. 3876, pp. 380–403. Springer, New York (Mar 2006), extended version available at http://eprint.iacr.org/2004/334
35. Cardelli, L., Ghelli, G., Gordon, A.D.: Secrecy and group creation. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 365–379. Springer, Heidelberg (Aug 2000)

36. Chevalier, Y., Küsters, R., Rusinowitch, M., Turuani, M.: Deciding the security of protocols with Diffie-Hellman exponentiation and products in exponents. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS'03. LNCS, vol. 2914, pp. 124–135. Springer, Heidelberg (Dec 2003)
37. Chevalier, Y., Küsters, R., Rusinowitch, M., Turuani, M.: An NP decision procedure for protocol insecurity with XOR. Theoretical Computer Science 338(1–3), 247–274 (Jun 2005)
38. Chevalier, Y., Vigneron, L.: A tool for lazy verification of security protocols. In: ASE'01. pp. 373–376. IEEE, Los Alamitos (Nov 2001)
39. Comon-Lundh, H., Cortier, V.: New decidability results for fragments of first-order logic and application to cryptographic protocols. In: Nieuwenhuis, R. (ed.) RTA'2003. LNCS, vol. 2706, pp. 148–164. Springer, Heidelberg (Jun 2003)
40. Comon-Lundh, H., Cortier, V.: Security properties: two agents are sufficient. Science of Computer Programming 50(1–3), 51–71 (Feb 2004)
41. Comon-Lundh, H., Shmatikov, V.: Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In: LICS'03. pp. 271–280. IEEE, Los Alamitos (Jun 2003)
42. Cremers, C.J.F.: Scyther - Semantics and Verification of Security Protocols. Ph.D. dissertation, Eindhoven University of Technology (Nov 2006)
43. Denker, G., Meseguer, J., Talcott, C.: Protocol specification and analysis in Maude. In: FMSP'98 (Jun 1998)
44. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. Commun. ACM 24(8), 533–536 (Aug 1981)
45. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory IT-22(6), 644–654 (Nov 1976)
46. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory IT-29(12), 198–208 (Mar 1983)
47. Durgin, N., Lincoln, P., Mitchell, J.C., Scedrov, A.: Multiset rewriting and the complexity of bounded security protocols. Journal of Computer Security 12(2), 247–311 (2004)
48. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. Theoretical Computer Science 367(1-2), 162–202 (2006)
49. Godskesen, J.C.: Formal verification of the ARAN protocol using the applied pi-calculus. In: WITS'06. pp. 99–113 (Mar 2006)
50. Gordon, A., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. Journal of Computer Security 12(3/4), 435–484 (2004)
51. Goubault-Larrecq, J.: Deciding $\mathcal{H}_1$ by resolution. Information Processing Letters 95(3), 401–408 (Aug 2005)
52. Goubault-Larrecq, J., Parrennes, F.: Cryptographic protocol analysis on real C code. In: Cousot, R. (ed.) VMCAI'05. LNCS, vol. 3385, pp. 363–379. Springer, Heidelberg (Jan 2005)
53. Heather, J., Lowe, G., Schneider, S.: How to prevent type flaw attacks on security protocols. In: CSFW'00. pp. 255–268. IEEE, Los Alamitos (Jul 2000)
54. Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K.: Plutus: Scalable secure file sharing on untrusted storage. In: FAST'03. pp. 29–42. Usenix, Berkeley (Apr 2003)
55. Khurana, H., Hahm, H.S.: Certified mailing lists. In: ASIACCS'06. pp. 46–58. ACM, New York (Mar 2006)

56. Kremer, S., Ryan, M.D.: Analysis of an electronic voting protocol in the applied pi calculus. In: Sagiv, M. (ed.) ESOP'05. LNCS, vol. 3444, pp. 186–200. Springer, Heidelberg (Apr 2005)
57. Küsters, R., Truderung, T.: Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach. In: CCS'08. pp. 129–138. ACM, New York (Oct 2008)
58. Küsters, R., Truderung, T.: Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In: CSF'09. pp. 157–171. IEEE, Los Alamitos (Jul 2009)
59. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: TACAS'96. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
60. Lux, K.D., May, M.J., Bhattad, N.L., Gunter, C.A.: WSEmail: Secure internet messaging based on web services. In: ICWS'05. pp. 75–82. IEEE, Los Alamitos (Jul 2005)
61. Lynch, C.: Oriented equational logic programming is complete. Journal of Symbolic Computation 21(1), 23–45 (1997)
62. Meadows, C.A.: The NRL protocol analyzer: An overview. Journal of Logic Programming 26(2), 113–131 (1996)
63. Meadows, C., Narendran, P.: A unification algorithm for the group Diffie-Hellman protocol. In: WITS'02 (Jan 2002)
64. Monniaux, D.: Abstracting cryptographic protocols with tree automata. Science of Computer Programming 47(2–3), 177–202 (2003)
65. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM 21(12), 993–999 (Dec 1978)
66. de Nivelle, H.: Ordering Refinements of Resolution. Ph.D. thesis, Technische Universiteit Delft (Oct 1995)
67. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6(1–2), 85–128 (1998)
68. Ramanujam, R., Suresh, S.: Tagging makes secrecy decidable with unbounded nonces as well. In: Pandya, P., Radhakrishnan, J. (eds.) FSTTCS'03. LNCS, vol. 2914, pp. 363–374. Springer, Heidelberg (Dec 2003)
69. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions is NP-complete. Theoretical Computer Science 299(1–3), 451–475 (Apr 2003)
70. Schmidt, B., Meier, S., Cremers, C., Basin, D.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: CSF'12. pp. 78–94. IEEE, Los Alamitos (Jun 2012)
71. Weidenbach, C.: Towards an automatic analysis of security protocols in first-order logic. In: Ganzinger, H. (ed.) CADE-16. LNAI, vol. 1632, pp. 314–328. Springer, Heidelberg (Jul 1999)