Reconstruction of Attacks against Cryptographic Protocols*

Xavier Allamigeon
École Polytechnique
and Corps des Télécommunications, Paris
xavier.allamigeon@polytechnique.org

Bruno Blanchet
CNRS, École Normale Supérieure, Paris
blanchet@di.ens.fr

Abstract

We study an automatic technique for the verification of cryptographic protocols based on a Horn clause model of the protocol. This technique yields proofs valid for an unbounded number of sessions of the protocol. However, up to now, it gave no definite information when the proof failed. In this paper, we present an algorithm for reconstructing an attack against the protocol when the desired security property does not hold. We have proved soundness, termination, as well as a partial completeness result for our algorithm. We have also implemented it in the automatic protocol verifier ProVerif. As an extreme example, we could reconstruct an attack involving 200 parallel sessions against the $f^{200}g^{200}$ protocol [21].

1. Introduction

The verification of cryptographic protocols is a very active research area. Recent progress in this area yields automatic security proofs valid for an unbounded number of executions of the protocol, in order to handle, for instance, the execution of a server which accepts many connections, possibly in parallel. Handling an unbounded number of sessions is important for obtaining actual proofs of security properties of protocols. However, this problem has been shown to be undecidable [16] for a reasonable model of protocols. So, in order to prove properties for an unbounded number of sessions, one needs to perform sound approximations: if the verifier claims that the property is true, then it is. However, the verifier may find "false attacks": situations in which the verifier cannot prove a true property.

One technique that can handle an unbounded number of sessions relies on Horn clauses. The protocol is formalized as a process in an extension of the pi calculus with cryptographic primitives. This process is first translated into a set of Horn clauses. These clauses use a fact $\mathtt{att}(M)$, which means that the attacker may have the message M. So if $\mathtt{att}(M)$ is not derivable from the clauses, then the protocol preserves the secrecy of the message M [2, 5]. Then, we use a resolution-based solving algorithm to determine whether $\mathtt{att}(M)$ is derivable from the clauses [9, 12]. When it is not derivable, we have secrecy. However, when it is derivable, both situations can happen: secrecy may be true (we have a false attack) or false. Our goal in this paper is to obtain more information in this case: we would like to reconstruct an attack against the protocol when secrecy does not hold. Formally, such an attack is an execution trace of the process that models the protocol, in which the attacker obtains the secret M. An exhaustive exploration of all traces smaller than a given size is practical only for small examples. It becomes too costly for more complex ones.

Therefore, we exploit the information provided by the resolution algorithm in order to reconstruct attacks. When the Horn clause technique fails to prove secrecy of M, it outputs a derivation of the fact $\mathtt{att}(M)$ from the Horn clauses. However, because of approximations, reconstructing an attack from this derivation is far from trivial: the Horn clauses do not take into account the number of executions of each step of the protocol and the synchronizations between inputs and outputs. Furthermore, the attack reconstruction necessarily fails in some cases: when secrecy is in fact true, and also sometimes when secrecy is false because of the undecidability of the problem. In fact, we have a formal notion of when a derivation corresponds to a trace, and the trace reconstruction algorithm is allowed to fail when the derivation of $\mathtt{att}(M)$ does not correspond to a trace.

As mentioned above, reconstructing the attack directly from the derivation would be very difficult. Instead, our reconstruction technique relies on the exploration of a restricted, finite set of traces, guided by the derivation of $\mathtt{att}(M)$. This simple idea yields an algorithm which is obviously sound: when it returns a trace, it is really an attack (Section 4.3). Furthermore, our restriction is such that the set of explored traces is finite (Section 4.4) and, provided a minor restriction on the allowed outputs is met, it contains

* This work was partly done while the authors were at Max-Planck-Institut für Informatik, Saarbrücken, Germany.

the desired attack when the derivation actually corresponds to an attack (Section 4.5). Our algorithm is also very fast in practice, because the restricted set of traces often contains only one trace (Section 4.6). We have implemented our algorithm in the protocol verifier ProVerif, available at http://www.di.ens.fr/~blanchet/crypto-eng.html.

Related work Up to now, the main methods that automatically find attacks against protocols are methods that do not perform approximations, such as the constraint solving technique of [22] and the first uses of model-checking [20]. The attack is then a direct result of the verification algorithm. However, these techniques are limited to a small, bounded number of sessions. In extensions of model checking using data independence techniques [25, 13], proofs can be obtained for an unbounded number of sessions but a bounded message size (provided some restrictions on the form of the protocol are met), at the cost of possible false attacks. These extensions are intended to prove protocols more than to find attacks, since attacks can be found with exact model-checking algorithms for a bounded number of sessions, when the state space is not too large. Most other techniques do not output attacks. The typing [17, 4] and theorem proving [24] approaches prove security properties but do not output attacks when the proof fails (although manual inspection of why the proof fails may help in finding an attack). For abstract interpretation techniques, such as [23], we do not know of results on how to reconstruct attacks.

For the Horn clause verification technique itself, the problem of reconstructing an explicit soundness proof when the verifier proves the property has been studied in [26, 18]. However, the problem of reconstructing an attack when the verifier fails to prove the property has not been studied yet as far as we know.

Outline The next section introduces our process calculus with its syntax and semantics. Section 3 recalls the translation of a process into Horn clauses. Section 4 is the core of our contribution: it presents our technique for reconstructing attacks and studies its properties (soundness, completeness, termination, complexity). Finally, Section 5 concludes with extensions and future work.

2. The process calculus

2.1. Syntax and informal semantics

We represent protocols in the calculus described in Figure 1. This calculus is essentially the one of [2, 5]. It distinguishes terms (messages) and processes (programs). It assumes an infinite set of variables denoted $x, y, z, \ldots$ and an infinite set of names denoted $a, b, c, s, \ldots$ It also distinguishes two categories of function symbols, constructors f and destructors g.

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

Figure 1. Syntax of the process calculus

Constructors build new terms, so terms can be variables, names, and constructor applications $f(M_1, \ldots, M_n)$. Destructors manipulate terms. Precisely, each destructor g is defined by a finite set $\mathrm{def}(g)$ of rewrite rules $g(M_1, \ldots, M_n) \rightarrow M$, such that $M_1, \ldots, M_n, M$ do not contain names and all variables of M also occur in $M_1, \ldots, M_n$. We also require that when several rewrite rules apply for the same arguments, they yield the same result. (We could remove this requirement, but this would complicate the work without much practical benefit.) The evaluation of $g(M_1, \ldots, M_n)$ succeeds and returns M when there is a rewrite rule $g(M'_1, \ldots, M'_n) \rightarrow M'$ in $\mathrm{def}(g)$ such that $M_1, \ldots, M_n, M$ is an instance of $M'_1, \ldots, M'_n, M'$. In this case, we write $g(M_1, \ldots, M_n) \rightarrow M$, and the destructor application $\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q$ executes P with x bound to M. Otherwise, it executes Q. The else clause can be omitted when it is 0. The conditional $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ can be defined as syntactic sugar for the destructor application $\mathit{let}\ x = \mathit{equals}(M, N)\ \mathit{in}\ P\ \mathit{else}\ Q$ where the destructor equals is defined by $\mathit{equals}(x, x) \rightarrow x$ and x is a fresh variable. Similarly, we define $\mathit{let}\ x = M\ \mathit{in}\ P$ as $\mathit{let}\ x = \mathit{id}(M)\ \mathit{in}\ P$ where the destructor id is defined by $\mathit{id}(x) \rightarrow x$.

Constructors and destructors can be used to represent most common cryptographic primitives. For instance, shared-key encryption can be encoded by a constructor $\mathit{sencrypt}(M, N)$ which returns the encryption of M under key N, and a destructor $\mathit{sdecrypt}$ defined by $\mathit{sdecrypt}(\mathit{sencrypt}(x, y), y) \rightarrow x$ which returns the cleartext x from the ciphertext $\mathit{sencrypt}(x, y)$ and the secret key y. Public-key encryption can be modeled thanks to two constructors $\mathit{pencrypt}(M, N)$ which returns the encryption of M under key N, and $\mathit{pk}(M)$ which builds a public key from the secret key M, and a destructor $\mathit{pdecrypt}$ defined by

$pdecrypt(pencrypt(x, pk(y)), y) \rightarrow x$. Similarly, signatures are encoded using a binary constructor $sign(M, N)$, which signs M with the key N, and two destructors $checksign$ defined by $checksign(sign(x, y), pk(y)) \rightarrow x$ (which returns the cleartext x from its signature $sign(x, y)$, after it has been checked with the public key $pk(y)$), and $getmess$ defined by $getmess(sign(x, y)) \rightarrow x$ (which returns the cleartext x from its signature without checking its validity). We refer to [10, 11] for other examples.

The other constructs come from the pi calculus: the inactive process 0, the parallel composition $P \mid Q$, the replication $!^k P$ which represents an unbounded number of copies of P in parallel, the restriction $(\nu a)P$ which creates a new name a then executes P, the output $\overline{M}\langle N\rangle.P$ which outputs message N on channel M then executes P, the input $M(x)^k.P$ which receives a message on channel M, binds x to that message and executes P. In this paper, we consider that the input and output can be executed even when the channel M does not reduce to a name at runtime. (We can also handle the other option, in which they block in this case.)

The replication $!^k P$ and the input $M(x)^k$ are labeled with a constant integer k, named occurrence label. We require that in the initial process P_0, each of these integers occurs at most once. These integers are used to track which replication and input in a reduced process comes from which construct in the initial process.

Free names and variables, $fn(P)$ and $fv(P)$, are defined as usual. A closed process is a process without free variables; it may contain free names. We denote by $\{M/x\}$ the substitution which replaces the variable x with the term M.

2.2. Semantics

Our semantics is described in Figure 2. Usually, the semantics of such calculi is defined by a structural equivalence and a reduction relation. In this paper, we adopt another presentation of the semantics, which eliminates the structural equivalence. (This idea was already used in [3].) In our semantics, a configuration is a triple $\mathcal{E}, \mathcal{P}, \mathcal{S}$, where $\mathcal{P}$ is a multiset of processes, $\mathcal{E}$ is the set of free names of $\mathcal{P}$ and of names created by the adversary, and $\mathcal{S}$ is the set of terms known by the adversary. Intuitively, the configuration $\mathcal{E}, \mathcal{P}, \mathcal{S}$ corresponds to the process

$$(\nu a_1) \ldots (\nu a_n)(P_1 \mid \ldots \mid P_m \mid Q)$$

where $\mathcal{E} = \{a_1, \ldots, a_n\}$, $\mathcal{P} = \{P_1, \ldots, P_m\}$, and Q represents an adversary whose current knowledge is $\mathcal{S}$.

The main advantage of eliminating structural congruence for our purpose is that the reduction is more guided than with the standard presentation. For each construct at the top of the process, a single reduction rule among the rules (Red Nil) to (Red Let2) is applicable. The semantics of the restriction is also closer to the intuition that the restriction creates a new name (here a'), and the renamings are strongly limited: they occur only when applying the restriction, while in the standard semantics, they can occur at any application of structural congruence. Our choice makes it easier to track that the same term occurs in several processes of a trace.

The rules (Red Out) to (Red New) represent the actions of the adversary. (Red Out) means that the process sends a message to the adversary, and (Red In) that it receives a message from the adversary. (Red Constr) and (Red Destr) correspond to internal computations of the adversary, applying respectively constructors and destructors, and (Red

$$\mathcal{E}, \mathcal{P} \cup \{0\}, \mathcal{S} \rightarrow \mathcal{E}, \mathcal{P}, \mathcal{S} \qquad \text{(Red Nil)}$$

$$\mathcal{E}, \mathcal{P} \cup \{P \mid Q\}, \mathcal{S} \rightarrow \mathcal{E}, \mathcal{P} \cup \{P, Q\}, \mathcal{S} \qquad \text{(Red Par)}$$

$$\mathcal{E}, \mathcal{P} \cup \{!^k P\}, \mathcal{S} \rightarrow \mathcal{E}, \mathcal{P} \cup \{P, !^k P\}, \mathcal{S} \qquad \text{(Red Repl)}$$

$$\frac{a' \notin \mathcal{E}}{\mathcal{E}, \mathcal{P} \cup \{(\nu a)P\}, \mathcal{S} \rightarrow \mathcal{E} \cup \{a'\}, \mathcal{P} \cup \{P\{a'/a\}\}, \mathcal{S}} \qquad \text{(Red Restr)}$$

$$\frac{M \notin \mathcal{S}}{\begin{array}{c}\mathcal{E}, \mathcal{P} \cup \{M(x)^k.P, \overline{M}\langle N\rangle.Q\}, \mathcal{S} \\ \rightarrow \mathcal{E}, \mathcal{P} \cup \{P\{N/x\}, Q\}, \mathcal{S}\end{array}} \qquad \text{(Red I/O)}$$

$$\frac{g \text{ destructor of arity } n, g(M_1, \ldots, M_n) \rightarrow M}{\begin{array}{c}\mathcal{E}, \mathcal{P} \cup \{let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q\}, \mathcal{S} \\ \rightarrow \mathcal{E}, \mathcal{P} \cup \{P\{M/x\}\}, \mathcal{S}\end{array}} \qquad \text{(Red Let1)}$$

$$\frac{\begin{array}{c}g \text{ destructor of arity } n, \\ g(M_1, \ldots, M_n) \not\rightarrow M \text{ for all terms } M\end{array}}{\begin{array}{c}\mathcal{E}, \mathcal{P} \cup \{let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q\}, \mathcal{S} \\ \rightarrow \mathcal{E}, \mathcal{P} \cup \{Q\}, \mathcal{S}\end{array}} \qquad \text{(Red Let2)}$$

$$\frac{M \in \mathcal{S}}{\mathcal{E}, \mathcal{P} \cup \{\overline{M}\langle N\rangle.P\}, \mathcal{S} \rightarrow \mathcal{E}, \mathcal{P} \cup \{P\}, \mathcal{S} \cup \{N\}} \qquad \text{(Red Out)}$$

$$\frac{M, N \in \mathcal{S}}{\mathcal{E}, \mathcal{P} \cup \{M(x)^k.P\}, \mathcal{S} \rightarrow \mathcal{E}, \mathcal{P} \cup \{P\{N/x\}\}, \mathcal{S}} \qquad \text{(Red In)}$$

$$\frac{f \text{ constructor of arity } n, M_1, \ldots, M_n \in \mathcal{S}}{\mathcal{E}, \mathcal{P}, \mathcal{S} \rightarrow \mathcal{E}, \mathcal{P}, \mathcal{S} \cup \{f(M_1, \ldots, M_n)\}} \qquad \text{(Red Constr)}$$

$$\frac{\begin{array}{c}g \text{ destructor of arity } n, M_1, \ldots, M_n \in \mathcal{S}, \\ g(M_1, \ldots, M_n) \rightarrow M\end{array}}{\mathcal{E}, \mathcal{P}, \mathcal{S} \rightarrow \mathcal{E}, \mathcal{P}, \mathcal{S} \cup \{M\}} \qquad \text{(Red Destr)}$$

$$\frac{a' \notin \mathcal{E}}{\mathcal{E}, \mathcal{P}, \mathcal{S} \rightarrow \mathcal{E} \cup \{a'\}, \mathcal{P}, \mathcal{S} \cup \{a'\}} \qquad \text{(Red New)}$$

Figure 2. Reduction rules of the semantics

New) to the creation of a new name by the adversary. Note that, for (Red I/O), we require that the adversary does not have the channel $M \notin \mathcal{S}$, since otherwise, this reduction can be simulated by steps (Red Out) followed by (Red In) (the adversary first receives the message from the process, then sends it back). This version of (Red I/O) avoids having two choices for each communication on a public channel, so it reduces the number of traces to consider.

2.3. Example

As a running example, we will consider a simplified version of the Denning-Sacco key distribution protocol [15]:

$$\begin{array}{ll} \text{Message 1.} & A \rightarrow B : \left\{\{k\}_{sk_A}\right\}_{pk_B} \\ \text{Message 2.} & B \rightarrow A : \{s\}_k \end{array}$$

In this protocol, two principals A and B wish to establish a shared key k. A creates a fresh key k, signs it with its secret key sk_A, encrypts it under the public key pk_B of B, and finally sends it to B (message 1). When he receives the message, B can decrypt it with its secret key sk_B, and assumes, checking the signature with the public key pk_A of A, that the key k has been created by A. As a result, A and B share this key, and B can send a secret s under k to A (message 2). We use the second message to check if the shared key k can be used to encrypt secret data shared by A and B. This protocol is subject to the following attack, described in [14]:

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

The principal A starts a session with the attacker C by sending the first message. When receiving this message, C decrypts it, and encrypts the result with the public key of B. The obtained message (Message 1') looks like a legitimate message for a session between A and B. C, impersonating A, sends it to B. B replies with the second message, as if B was talking to A. Since C also obtains k from the first message, it can decrypt B's message and obtain the secret s.

We can represent this protocol by the following processes:

$$\begin{array}{l} P_A(sk_A) = c(x_{pk_B})^1.(\nu k) \\ \quad \overline{c}\langle pencrypt(sign(k, sk_A), x_{pk_B})\rangle \\ P_B(pk_A, sk_B) = c(x_m)^2.let\ x_p = pdecrypt(x_m, sk_B)\ in \\ \quad let\ x_k = checksign(x_p, pk_A)\ in\ \overline{c}\langle sencrypt(s, x_k)\rangle \\ P_0 = (\nu sk_A) let\ pk_A = pk(sk_A)\ in\ \overline{c}\langle pk_A\rangle. \\ \quad (\nu sk_B) let\ pk_B = pk(sk_B)\ in\ \overline{c}\langle pk_B\rangle. \\ \quad (!^3 P_A(sk_A) \mid !^4 P_B(pk_A, sk_B)) \end{array}$$

P_0 first creates the secret key sk_A of A, builds its public key pk_A, and publishes it by sending it on the public channel c. Similarly for B, it creates sk_B, computes pk_B, and publishes it. Then it consists of an unbounded number of copies of P_A and P_B, which respectively represent the principals A and B. We consider that the attacker chooses the participant whom A will establish the shared key k with, by sending to A the public key of this participant on the channel c. P_A then inputs this message, binds x_{pk_B} to it, creates a new key k, and sends the key signed with sk_A and encrypted under x_{pk_B} on the channel c (message 1). P_B inputs this message, tries to get k by decrypting the message and checking the signature, binds x_k to the obtained key, and finally sends the secret s under x_k (message 2).

2.4. Secrecy

Let the protocol be represented by a closed process P_0. Let S_0 be the set of public names of the protocol, which corresponds to the initial knowledge of the adversary. This set contains for example public channel names. Secrecy is then defined as follows:

Definition 1 Let P_0 be a closed process and S_0 a finite set of names. A *trace $\mathcal{T}$ of P_0 from S_0* is a finite sequence of reductions $fn(P_0) \cup S_0, \{P_0\}, S_0 \rightarrow \ldots \rightarrow \mathcal{E}', \mathcal{P}', \mathcal{S}'$.

The closed term M *is learnt* in the trace $\mathcal{T}$ if and only if $\mathcal{T}$ contains a state $\mathcal{E}, \mathcal{P}, \mathcal{S}$ where $M \in \mathcal{S}$.

Let M_{secret} be a closed term such that $fn(M_{secret}) \subseteq fn(P_0)$. The process P_0 *preserves the secrecy of M_{secret} from S_0* if and only if there exists no trace of P_0 from S_0 in which M_{secret} is learnt.

This definition of secrecy resembles the one of [1, Section 6.2]. We have shown that this definition is equivalent to the notion of secrecy of [3], which is similar to [4, 2, 5] and in which the adversary is represented by any process instead of by its knowledge $\mathcal{S}$. [4] already mentioned that these two notions capture the "same concept", but as far as we know, the equivalence was not proved up to now.

In the following of the paper, P_0 will always be a closed process, M_{secret} a closed term such that $fn(M_{secret}) \subseteq fn(P_0)$, and S_0 a finite set of names. The description of the attack that we are going to produce will be a trace in which M_{secret} is learnt.

3. Generation of the clauses

The verification algorithm first translates the protocol and the actions of the adversary into Horn clauses. Then it uses a resolution-based solving algorithm in order to determine whether a fact is derivable from the clauses. If the fact "the attacker knows M_{secret}" is not derivable from the clauses, then the protocol preserves the secrecy of M_{secret}.

We present here the generation of the clauses, which is based on the one of [2, 10]. The main novelty with respect to [2, 10] is that we add labels to the clauses in order to remember where they come from. This information is necessary for reconstructing traces.

We assume that the protocol is represented by a closed process P_0, in which all restrictions use distinct names, and names in restrictions are distinct from free names of P_0. We also assume that the bound variables of P_0 are distinct. (These constraints can be enforced by renaming.)

The terms used in clauses are named "patterns". They are defined by the following grammar:

$p, q ::=$	patterns
x, y, z	variable
i	variable session identifier
λ	constant session identifier
$a[p_1, \ldots, p_n]$	name
$f(p_1, \ldots, p_n)$	constructor application

Session identifiers are used to distinguish copies of the same process created by a replication. Every time a replication is executed, the generated copy of the process is associated to a fresh session identifier λ. Each name a' created by a restriction (νa) is then mapped to pattern $a[p_1, \ldots, p_n]$, where a is considered as a function symbol. (We write $a[\ldots]$ rather than $a(\ldots)$ just to distinguish it from constructor and destructor applications.) The arguments $p_1, \ldots, p_n$ are used to distinguish different names created by the same restriction. They include both the messages received by inputs above the considered restriction and session identifiers of replications above that restriction. Since different names coming from the same restriction are created in different copies of the process, so have different session identifiers, they are mapped to different patterns. For example, in the Denning-Sacco protocol, names k are represented by patterns $k[i_A, x_{pk_B}]$ where i_A is the session identifier associated with replication $!^3$ just above P_A and x_{pk_B} is the message received in P_A by the input $c(x_{pk_B})^1$. Then we have a different pattern for names created in a different copy of P_A or after receiving different inputs x_{pk_B}.

The clauses use two predicates, att and mess:

$F ::=$	facts
$\mathsf{att}(p)$	the attacker may know p
$\mathsf{mess}(p, q)$	the message q may appear on channel p

3.1. Attacker clauses

The clauses below describe the actions of the adversary. The clause (Init) corresponds to its initial knowledge. Clause (Rn) expresses that it can create an unbounded number of new names $b[i]$. Clauses (Ra$_f$) and (Ra$_g$) mean that it can apply constructors and destructors respectively, clauses (Rl) and (Rs) that it can listen and send messages on channels it has. These clauses have a label $\mathcal{L}$ above the arrow, used to remember their origin: (Init) and (Rn) have empty label, (Ra$_f$) and (Ra$_g$) have label Ra, while (Rl) and (Rs) have labels Rl and Rs respectively.

For each $a \in S_0$, $\mathsf{att}(a[\,])$ (Init)

$\mathsf{att}(b[i])$ where b does not occur in P_0 (Rn)

For each constructor f of arity n,

$$\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \xRightarrow{Ra} \mathsf{att}(f(x_1, \ldots, x_n)) \quad (\mathrm{Ra}_f)$$

For each destructor g,

for each rewrite rule $g(N_1, \ldots, N_n) \rightarrow N$ in $\mathrm{def}(g)$,

$$\mathsf{att}(N_1) \wedge \ldots \wedge \mathsf{att}(N_n) \xRightarrow{Ra} \mathsf{att}(N) \quad (\mathrm{Ra}_g)$$

$$\mathsf{att}(x) \wedge \mathsf{mess}(x, y) \xRightarrow{Rl} \mathsf{att}(y) \quad (\mathrm{Rl})$$

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \xRightarrow{Rs} \mathsf{mess}(x, y) \quad (\mathrm{Rs})$$

3.2. Protocol clauses

We now define the translation of the protocol itself into Horn clauses. More precisely, we define the set of clauses $[\![P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E}$ by induction on P, where P is the process to translate, $\mathcal{K}$ is the sequence of occurrence labels of inputs and replications above P in P_0, $\mathcal{H}$ is the sequence of facts $\mathsf{mess}(p, p')$ corresponding to the inputs above P in P_0, $\mathcal{I}$ is the sequence of patterns corresponding to terms received by inputs and of session identifiers of replications above P in P_0, and $\mathcal{E}$ is an environment, that is, a mapping from names and variables to patterns.

The addition of a mapping $u \mapsto p$ to $\mathcal{E}$ is denoted by $\mathcal{E}\,[u \mapsto p]$. If M is a term, $\mathcal{E}(M)$ is the pattern defined by considering $\mathcal{E}$ as a substitution defined by $\mathcal{E}(u) = p$ if $u \mapsto p$ is in $\mathcal{E}$. The concatenation of a fact F to $\mathcal{H}$ is denoted by $\mathcal{H} \wedge F$. The concatenation of k to $\mathcal{K}$ is denoted by $\mathcal{K}, k$, and similarly for $\mathcal{I}, p$. The translation is defined as follows:

$[\![0, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E} = \emptyset$

$[\![P \mid Q, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E} = [\![P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E} \cup [\![Q, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E}$

$[\![!^k P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E} = [\![P, (\mathcal{K}, k), \mathcal{H}, (\mathcal{I}, i)]\!]\mathcal{E}$
where i is a new variable session identifier

$[\![(\nu a)P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E} = [\![P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!](\mathcal{E}\,[a \mapsto a\,[\mathcal{I}]])$

$[\![M(x)^k.P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E} = [\![P, (\mathcal{K}, k), (\mathcal{H} \wedge \mathsf{mess}(\mathcal{E}(M), x')), (\mathcal{I}, x')]\!](\mathcal{E}\,[x \mapsto x'])$ where x' is a new variable

$[\![\overline{M}\langle N\rangle.P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E} = [\![P, \mathcal{K}, \mathcal{H}, \mathcal{I}]\!]\mathcal{E}$
$\cup \{\mathcal{H} \xRightarrow{\mathcal{K}, \mathcal{I}} \mathsf{mess}(\mathcal{E}(M), \mathcal{E}(N))\}$

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

The set of clauses corresponding to the protocol is then $[\![P_0, \emptyset, \emptyset, \emptyset]\!]\mathcal{E}_0$ where $\mathcal{E}_0 = \{a \mapsto a[\,] \mid a \in fn(P_0)\}$ and $\emptyset$ is the empty sequence.

Although the presentation of the formulas is a bit different in order to fit the instrumented semantics defined in Section 4.1, the generated clauses are the same as in [10], except for the addition of labels on the arrow of the clauses.

The translation of a replication simply updates the various parameters: it adds the occurrence label k to $\mathcal{K}$ and a new session identifier to $\mathcal{I}$. The replication is otherwise ignored since clauses can be applied any number of times.

The restriction adds to $\mathcal{E}$ the mapping of the name a to the pattern $a[\mathcal{I}]$. So each name is represented by a pattern having as arguments the previous inputs and the session identifiers of the session in which the name is created.

Each input adds the fact $\mathsf{mess}(\mathcal{E}(M), x')$ corresponding to the received message to $\mathcal{H}$. So $\mathcal{H}$ keeps track of the messages that must be received in order to reach the current program point. The input also adds its occurrence label k to $\mathcal{K}$, and the pattern of the received message x' to $\mathcal{I}$.

Each output generates a new clause, which expresses that, when $\mathcal{H}$ is true (that is, when the output may be executed), the message N may be sent on channel M. In other words, if the process outputs message N on channel M after receiving $N_1, \ldots, N_n$ on channels $M_1, \ldots, M_n$ respectively (thanks to inputs located above the output in P_0), then we generate a clause $\mathsf{mess}(p_1, q_1) \wedge \ldots \wedge \mathsf{mess}(p_n, q_n) \xRightarrow{\mathcal{K},\mathcal{I}} \mathsf{mess}(p, q)$ where p, q, p_i, q_i are the patterns corresponding to M, N, M_i, N_i respectively. This clause is labeled with the pair $\mathcal{K}, \mathcal{I}$ in order to remember that the output is executed after the inputs and replications whose occurrence labels are in $\mathcal{K}$ and whose received messages and session identifiers are in $\mathcal{I}$.

Finally, the translation of a destructor application is the union of the cases in which the destructor succeeds and in which the destructor fails. In the first case, we execute P after instantiating all patterns by the substitution σ, in order to record the information that the destructor succeeds. In the second case, we execute Q.

Furthermore, in the generated clauses, the algorithm will replace every atom of the form $\mathsf{mess}(c[\,], p)$ where $c \in S_0$ by the atom $\mathsf{att}(p)$, in order to simplify the clauses. The new clauses are obviously equivalent to the previous ones thanks to the clauses (Rl) and (Rs). (For simplicity, we ignore this optimization in our theoretical study in this paper.)

For example, the Denning-Sacco protocol is translated into the following clauses:

$$\xRightarrow{\emptyset,\emptyset} \mathsf{att}(pk(sk_A[\,])) \tag{DS1}$$

$$\xRightarrow{\emptyset,\emptyset} \mathsf{att}(pk(sk_B[\,])) \tag{DS2}$$

$$\begin{aligned}&\mathsf{att}(x) \xRightarrow{(3,1),(i_A,x)} \\ &\quad \mathsf{att}(pencrypt(sign(k[i_A, x], sk_A[\,]), x))\end{aligned} \tag{DS3}$$

$$\begin{aligned}&\mathsf{att}(p) \xRightarrow{(4,2),(i_B,p)} \mathsf{att}(sencrypt(s[\,], x')) \\ &\text{with } p = pencrypt(sign(x', sk_A[\,]), pk(sk_B[\,]))\end{aligned} \tag{DS4}$$

The first two clauses correspond to the outputs of the public keys of A and B. The third one represents the behavior of a session of the process P_A: if the attacker has a public key x, it can send x to A, A replies with the fresh key k signed with sk_A and encrypted under x, which the attacker intercepts. The label $(3, 1), (i_A, x)$ of this clause means that the output is executed after making a copy of the replicated process of replication 3, with associated session identifier i_A, and receiving the message x on input 1. The last clause translates the fact that a session of P_B is able to output the secret s encrypted under a key when the message received by the input 2 is well-formed (*i.e.* all destructors in P_B succeed). Similarly, the label $(4, 2), (i_B, p)$ means that replication 4 has been reduced (with associated session identifier i_B) and that message p has been received in input 2 before sending the output.

This translation introduces some approximations. For instance, we assume that the "$else$" clause of the destructor application may always be executed. Moreover, actions are considered as implicitly replicated, since clauses can be applied any number of times. (The only exception concerns name creation: since different names are distinguished by different session identifiers, a restriction is not equivalent to a replicated restriction.) These approximations lead to producing false attacks, that is, situations in which the verifier can derive $\mathsf{att}(\mathcal{E}_0(M_{secret}))$ from the clauses but the process preserves the secrecy of M_{secret}. To illustrate such a case, let us consider the following process:

$$P_{false} = (\nu a)c(x)^1.\overline{c}\langle a\rangle.if\ a = x\ then\ \overline{c}\langle s\rangle\ else\ 0$$

where c is a public channel. It generates

$$\mathsf{att}(x) \xRightarrow{(1),(x)} \mathsf{att}(a[\,])$$
$$\mathsf{att}(a[\,]) \xRightarrow{(1),(a[\,])} \mathsf{att}(s[\,])$$

Then we derive $\mathsf{att}(s[\,])$ using for instance (Rn) to obtain $\mathsf{att}(x)$ for some x. Nevertheless, the reader will easily check that s is not learnt in any trace of P_{false} from $\{c\}$. This discrepancy comes from the fact that the Horn clause model allows repetitions of executions, as if the process was

$$P'_{false} = (\nu a)!^2 c(x)^1.\overline{c}\langle a\rangle.if\ a = x\ then\ \overline{c}\langle s\rangle\ else\ 0$$

and P'_{false} does not preserve the secrecy of s. (The attacker obtains a after sending anything on c, then in a second run sends a on c so that P'_{false} outputs s.) Our algorithm will obviously not find a trace of P_{false} corresponding to a derivation of $\mathsf{att}(s[\,])$.

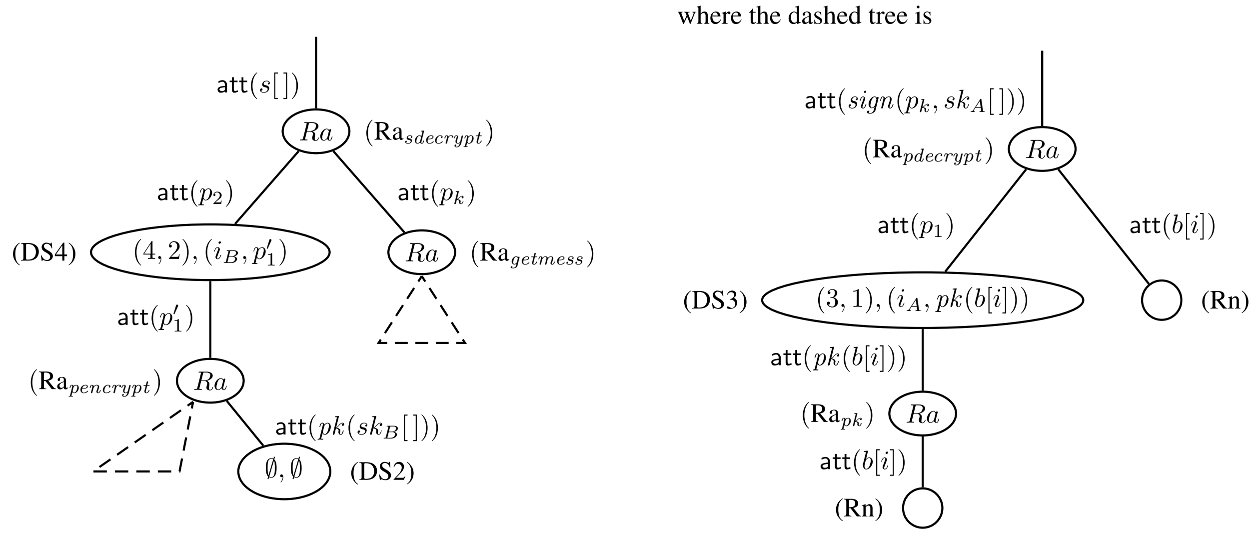


Figure 3. Derivation tree for the Denning-Sacco protocol

3.3. Secrecy result

We denote by $\mathcal{R}_{P_0,S_0}$ the set of clauses that represent the attacker abilities and the process P_0:

$$\mathcal{R}_{P_0,S_0} = [\![P_0, \emptyset, \emptyset, \emptyset]\!]\mathcal{E}_0 \cup \{(\text{Init}), (\text{Rn}), (\text{Ra}_f), (\text{Ra}_g), (\text{Rl}), (\text{Rs})\}$$

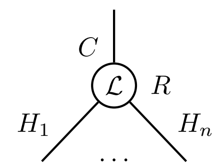
We recall the following result [5]:

Theorem 1 *If the fact* $\mathsf{att}(\mathcal{E}_0(M_{secret}))$ *is not derivable from the clauses* $\mathcal{R}_{P_0,S_0}$*, then* P_0 *preserves the secrecy of* M_{secret} *from* S_0.

We determine whether a fact is derivable from the clauses using a resolution-based algorithm detailed in [9, 12]. When the fact is derivable, the resolution algorithm reconstructs a derivation tree of $\mathsf{att}(\mathcal{E}_0(M_{secret}))$ from $\mathcal{R}_{P_0,S_0}$. Such a tree can be formally defined as follows:

Definition 2 (Derivation) Let F be a closed fact. Let $\mathcal{R}$ be a set of clauses. A derivation tree of F from $\mathcal{R}$ is a finite tree defined as follows:

1. Its nodes (except the root) are labeled by clause labels $\mathcal{L}$ (which may be empty, Rl, Rs, Ra, or the pair $\mathcal{K}, \mathcal{I}$) and by clauses $R \in \mathcal{R}$.
2. Its edges are labeled by facts.
3. If the tree contains a node labeled by $\mathcal{L}$ and R with one incoming edge labeled by C and n outgoing edges labeled by $H_1, \ldots, H_n$, then there exists a substitution σ such that $\sigma R = (H_1 \wedge \ldots \wedge H_n \overset{\mathcal{L}}{\Rightarrow} C)$ (so C can be derived from $H_1, \ldots, H_n$ using R).



4. The root has one outgoing edge, labeled by F.

In our running example, our resolution algorithm generates the derivation tree of $\mathsf{att}(s[])$ from the initial knowledge $S_0 = \{c\}$ given in Figure 3, where

$$\begin{aligned} p_k &= k[i_A, pk(b[i])] \\ p_1 &= pencrypt(sign(p_k, sk_A[]), pk(b[i])) \\ p'_1 &= pencrypt(sign(p_k, sk_A[]), pk(sk_B[])) \\ p_2 &= sencrypt(s[], p_k) \end{aligned}$$

Informally, this derivation corresponds to the well-known attack against the Denning-Sacco protocol [14]: at the bottom of the second column of Figure 3, the attacker creates a new secret key (of pattern $b[i]$) and the corresponding public key by (Ra$_{pk}$). Then it starts a run with A, by clause (DS3) to obtain the first message of the protocol (of pattern p_1), and decrypts it, obtaining the signature. Then, at the bottom of the first column of Figure 3, it encrypts the signature with pk_B, to obtain the first message of a run between A and B (pattern p'_1). Using (DS4), it obtains the second message $\{s\}_k$. On the other hand, it can also obtain k from the signature by (Ra$_{getmess}$), so it finally obtains s by decryption.

$$\mathcal{E}, \mathcal{P} \cup \{(0, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda \qquad \text{(Ins Nil)}$$

$$\mathcal{E}, \mathcal{P} \cup \{(P \mid Q, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(P, \mathcal{K}, \mathcal{H}, \mathcal{I}), (Q, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \qquad \text{(Ins Par)}$$

$$\frac{\lambda \notin \Lambda}{\mathcal{E}, \mathcal{P} \cup \{(!^k P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(P, (\mathcal{K}, k), \mathcal{H}, (\mathcal{I}, \lambda)), (!^k P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \cup \{\lambda\}} \qquad \text{(Ins Repl)}$$

$$\frac{a' \notin dom(\mathcal{E})}{\mathcal{E}, \mathcal{P} \cup \{((\nu a)P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}[a' \mapsto a[\mathcal{I}]], \mathcal{P} \cup \{(P\{a'/a\}, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda} \qquad \text{(Ins Restr)}$$

$$\frac{M \notin \mathcal{S}}{\begin{array}{c}\mathcal{E}, \mathcal{P} \cup \{(M(x)^k.P, \mathcal{K}, \mathcal{H}, \mathcal{I}), (\overline{M}\langle N\rangle.Q, \mathcal{K}', \mathcal{H}', \mathcal{I}')\}, \mathcal{S}, \Lambda \rightsquigarrow \\ \mathcal{E}, \mathcal{P} \cup \{(P\{N/x\}, (\mathcal{K}, k), \mathcal{H} \wedge \mathsf{mess}(\mathcal{E}(M), \mathcal{E}(N)), (\mathcal{I}, \mathcal{E}(N))), (Q, \mathcal{K}', \mathcal{H}', \mathcal{I}')\}, \mathcal{S}, \Lambda\end{array}} \qquad \text{(Ins I/O)}$$

$$\frac{g \text{ destructor of arity } n, g(M_1, \ldots, M_n) \to M}{\begin{array}{c}\mathcal{E}, \mathcal{P} \cup \{(let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \\ \mathcal{E}, \mathcal{P} \cup \{(P\{M/x\}, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda\end{array}} \qquad \text{(Ins Let1)}$$

$$\frac{g \text{ destructor of arity } n, g(M_1, \ldots, M_n) \not\to M \text{ for all terms } M}{\mathcal{E}, \mathcal{P} \cup \{(let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(Q, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda} \qquad \text{(Ins Let2)}$$

$$\frac{M \in \mathcal{S}}{\mathcal{E}, \mathcal{P} \cup \{(\overline{M}\langle N\rangle.P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S} \cup \{N\}, \Lambda} \qquad \text{(Ins Out)}$$

$$\frac{M, N \in \mathcal{S}}{\mathcal{E}, \mathcal{P} \cup \{(M(x)^k.P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(P\{N/x\}, (\mathcal{K}, k), \mathcal{H} \wedge \mathsf{mess}(\mathcal{E}(M), \mathcal{E}(N)), (\mathcal{I}, \mathcal{E}(N)))\}, \mathcal{S}, \Lambda} \qquad \text{(Ins In)}$$

$$\frac{f \text{ constructor of arity } n, M_1, \ldots, M_n \in \mathcal{S}}{\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P}, \mathcal{S} \cup \{f(M_1, \ldots, M_n)\}, \Lambda} \qquad \text{(Ins Constr)}$$

$$\frac{g \text{ destructor of arity } n, M_1, \ldots, M_n \in \mathcal{S}, g(M_1, \ldots, M_n) \to M}{\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P}, \mathcal{S} \cup \{M\}, \Lambda} \qquad \text{(Ins Destr)}$$

$$\frac{a' \notin dom(\mathcal{E}) \qquad b[\lambda] \notin im(\mathcal{E})}{\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}[a' \mapsto b[\lambda]], \mathcal{P}, \mathcal{S} \cup \{a'\}, \Lambda} \qquad \text{(Ins New)}$$

Figure 4. Instrumented semantics

4. Reconstruction of attacks

4.1. Instrumented semantics

We instrument the semantics, to add additional information in order to remember the mapping between names and their corresponding patterns, as well as to track where each process of the semantic configuration comes from. In order to achieve this goal, we remember the received messages, the session identifiers, and the occurrence labels of executed inputs and replications. More precisely, the state of the instrumented semantics is of the form $\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda$, where $\mathcal{E}$ is a mapping from names to their corresponding patterns, $\mathcal{S}$ is the attacker knowledge as in the initial semantics, $\mathcal{P}$ is a multiset of quadruples $(P, \mathcal{K}, \mathcal{H}, \mathcal{I})$, with the same meaning as in the generation of clauses (this helps in establishing a precise correspondence between the clauses and the trace):

- P is a process.
- $\mathcal{K}$ is the list of labels of already reduced inputs and replications that occur above P in P_0. That is, when $\mathcal{K} = (k_1, \ldots, k_n)$, P comes from reductions of a process P_0 of the form $\alpha_1.C_1[\alpha_2.C_2[\ldots \alpha_n.C_n[P'] \ldots]]$ where P is an instance of P', $C_1, \ldots, C_n$ are contexts not containing replications and inputs, and for all $i \in \{1, \ldots, n\}$, $\alpha_i = M_i(x_i)^{k_i}$ or $\alpha_i = !^{k_i}$.
- $\mathcal{H}$ is the list of facts $\mathsf{mess}(p, p')$ corresponding to already reduced inputs that occur above P in P_0.
- $\mathcal{I}$ is the list of patterns corresponding to terms received by the inputs and of session identifiers of replications that occur above P in P_0.

and Λ is the set of already used session identifiers.

The semantics is defined in Figure 4. All rules just update the configuration described above. (Ins Repl) picks an unused session identifier λ and uses it for the new copy of P. (Ins Restr) adds a new name a' to $\mathcal{E}$, and maps it to its corresponding pattern $a[\mathcal{I}]$. (Ins In) updates the information for process P by adding the occurrence label k of the input to $\mathcal{K}$, the fact $\mathsf{mess}(\mathcal{E}(M), \mathcal{E}(N))$ to $\mathcal{H}$, and the message $\mathcal{E}(N)$ to $\mathcal{I}$. (Ins I/O) updates the information for the process performing the input in a similar way. In (Ins New), b is the same function symbol as in (Rn).

The following proposition expresses that the instrumented semantics just adds more information on existing traces, without really changing them.

Proposition 1 *If by a reduction (Ins R) of the instrumented semantics, we have*

$$\mathcal{E}, \{(P_1, \mathcal{K}_1, \mathcal{H}_1, \mathcal{I}_1), \ldots, (P_n, \mathcal{K}_n, \mathcal{H}_n, \mathcal{I}_n)\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}', \{(P'_1, \mathcal{K}'_1, \mathcal{H}'_1, \mathcal{I}'_1), \ldots, (P'_{n'}, \mathcal{K}'_{n'}, \mathcal{H}'_{n'}, \mathcal{I}'_{n'})\}, \mathcal{S}', \Lambda' \tag{1}$$

then the corresponding reduction (Red R) yields

$$\mathcal{E}_{\mathrm{dom}}, \{P_1, \ldots, P_n\}, \mathcal{S} \rightarrow \mathcal{E}'_{\mathrm{dom}}, \{P'_1, \ldots, P'_{n'}\}, \mathcal{S}' \tag{2}$$

with $\mathcal{E}_{\mathrm{dom}} = dom(\mathcal{E})$ and $\mathcal{E}'_{\mathrm{dom}} = dom(\mathcal{E}')$.

Conversely, if a reduction (Red R) yields (2) *then for any $\mathcal{E}$, Λ, $\mathcal{K}_i, \mathcal{H}_i, \mathcal{I}_i$ ($i \in \{1, \ldots, n\}$), there exist $\mathcal{E}'$, Λ', $\mathcal{K}'_i, \mathcal{H}'_i, \mathcal{I}'_i$ ($i \in \{1, \ldots, n'\}$), such that* (1) *with $\mathcal{E}_{\mathrm{dom}} = dom(\mathcal{E})$ and $\mathcal{E}'_{\mathrm{dom}} = dom(\mathcal{E}')$.*

4.2. Restricted semantics

In this section, we define the restricted semantics, in which the reductions are guided by a closed derivation tree. The reconstruction of the attack is done by exhaustive exploration of all traces of this semantics.

In the derivation tree built in Section 3.3, the session identifiers are variables. We build a closed derivation tree $\mathcal{D}$ from it, by substituting distinct constant session identifiers λ for variable session identifiers.[1]

We say that a derivation tree $\mathcal{D}$ contains a clause $\mathcal{H}' \xRightarrow{\mathcal{K}', \mathcal{I}'} C'$, and we write $\mathcal{H}' \xRightarrow{\mathcal{K}', \mathcal{I}'} C' \in \mathcal{D}$, if and only if $\mathcal{D}$ contains a node labeled $\mathcal{K}', \mathcal{I}'$ and R, whose incoming edge is labeled C' and outgoing edges are labeled $\mathcal{H}'$. We denote by $\mathcal{K} \sqsubseteq \mathcal{K}'$ the fact that $\mathcal{K}$ is a prefix of $\mathcal{K}'$, and similarly for $\mathcal{I} \sqsubseteq \mathcal{I}'$ and $\mathcal{H} \sqsubseteq \mathcal{H}'$. We say that $\mathcal{D}$ justifies $\mathcal{K}, \mathcal{H}, \mathcal{I}$ when there exists $\mathcal{H}' \xRightarrow{\mathcal{K}', \mathcal{I}'} C \in \mathcal{D}$ such that $\mathcal{K} \sqsubseteq \mathcal{K}'$, $\mathcal{I} \sqsubseteq \mathcal{I}'$, and $\mathcal{H} \sqsubseteq \mathcal{H}'$.

We now define the restricted semantics which restricts the instrumented semantics to traces that correspond to the derivation tree $\mathcal{D}$. The rules (Res Nil), (Res Par), (Res Restr), (Res Out), (Res Let1), and (Res Let2) are identical to the corresponding rules (Ins R) of the instrumented semantics. The other rules are given in Figure 5.

Intuitively, each clause corresponds to an action of the adversary or comes from an output of the protocol, as described in Section 3. In order to find a trace corresponding to a given derivation tree $\mathcal{D}$, we restrict the semantics so that a reduction rule is used only when there exists a corresponding clause in $\mathcal{D}$. More precisely:

- We first consider the cases of (Res In), (Res I/O) when $Q = 0$, and (Res Repl). These rules create a process P' as follows: (Res In) and (Res I/O) execute an input process $M(x)^k.P$ by replacing it with $P' = P\{N/x\}$; (Res Repl) executes $!^k P$ by creating a copy of P, so $P' = P$. We use these three reduction rules only if $\mathcal{D}$ justifies the parameters $\mathcal{K}_{P'}, \mathcal{H}_{P'}, \mathcal{I}_{P'}$ of the created process P', that is, if there exists a clause $\mathcal{H}' \xRightarrow{\mathcal{K}', \mathcal{I}'} C$ in $\mathcal{D}$ such that $\mathcal{K}_{P'} \sqsubseteq \mathcal{K}'$, $\mathcal{H}_{P'} \sqsubseteq \mathcal{H}'$, and $\mathcal{I}_{P'} \sqsubseteq \mathcal{I}'$.

  Indeed, the clause $\mathcal{H}' \xRightarrow{\mathcal{K}', \mathcal{I}'} C$ means that an output is executed after executing the sequence of inputs and replications described by $\mathcal{K}', \mathcal{H}', \mathcal{I}'$ and located above the output in the initial process. The process $(P', \mathcal{K}_{P'}, \mathcal{H}_{P'}, \mathcal{I}_{P'})$ is obtained after executing the sequence of inputs and replications described by $\mathcal{K}_{P'}, \mathcal{H}_{P'}, \mathcal{I}_{P'}$ and located above P' in the initial process. If the clause comes from an output located in the process P', then the sequence of inputs and replications above P' is a prefix of the one above the output, so $\mathcal{K}_{P'} \sqsubseteq \mathcal{K}'$, $\mathcal{H}_{P'} \sqsubseteq \mathcal{H}'$, and $\mathcal{I}_{P'} \sqsubseteq \mathcal{I}'$. In this case, we allow the creation of the process P' so that the out-

$$\frac{M, N \in \mathcal{S} \quad F = \mathsf{mess}(\mathcal{E}(M), \mathcal{E}(N)) \quad \mathcal{D} \text{ justifies } (\mathcal{K}, k), \mathcal{H} \wedge F, (\mathcal{I}, \mathcal{E}(N))}{\mathcal{E}, \mathcal{P} \cup \{(M(x)^k.P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(P\{N/x\}, (\mathcal{K}, k), \mathcal{H} \wedge F, (\mathcal{I}, \mathcal{E}(N)))\}, \mathcal{S}, \Lambda} \quad \text{(Res In)}$$

$$\frac{M \notin \mathcal{S} \quad F = \mathsf{mess}(\mathcal{E}(M), \mathcal{E}(N)) \quad Q \neq 0 \text{ or } \mathcal{D} \text{ justifies } (\mathcal{K}, k), \mathcal{H} \wedge F, (\mathcal{I}, \mathcal{E}(N))}{\mathcal{E}, \mathcal{P} \cup \{(M(x)^k.P, \mathcal{K}, \mathcal{H}, \mathcal{I}), (\overline{M}\langle N\rangle.Q, \mathcal{K}'', \mathcal{H}'', \mathcal{I}'')\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(P\{N/x\}, (\mathcal{K}, k), \mathcal{H} \wedge F, (\mathcal{I}, \mathcal{E}(N))), (Q, \mathcal{K}'', \mathcal{H}'', \mathcal{I}'')\}, \mathcal{S}, \Lambda} \quad \text{(Res I/O)}$$

$$\frac{\lambda \notin \Lambda \quad \mathcal{D} \text{ justifies } (\mathcal{K}, k), \mathcal{H}, (\mathcal{I}, \lambda)}{\mathcal{E}, \mathcal{P} \cup \{(!^k P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P} \cup \{(P, (\mathcal{K}, k), \mathcal{H}, (\mathcal{I}, \lambda)), (!^k P, \mathcal{K}, \mathcal{H}, \mathcal{I})\}, \mathcal{S}, \Lambda \cup \{\lambda\}} \quad \text{(Res Repl)}$$

$$\frac{\mathsf{att}(\mathcal{E}(M_1)) \wedge \ldots \wedge \mathsf{att}(\mathcal{E}(M_n)) \xRightarrow{Ra} \mathsf{att}(\mathcal{E}(M)) \in \mathcal{D} \quad M_1, \ldots, M_n \in \mathcal{S}}{\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}, \mathcal{P}, \mathcal{S} \cup \{M\}, \Lambda} \quad \text{(Res Constr) / (Res Destr)}$$

$$\frac{\mathsf{att}(b[\lambda]) \in \mathcal{D} \quad a' \notin dom(\mathcal{E}) \quad b[\lambda] \notin im(\mathcal{E})}{\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}[a' \mapsto b[\lambda]], \mathcal{P}, \mathcal{S} \cup \{a'\}, \Lambda} \quad \text{(Res New)}$$

Figure 5. Restricted semantics

1 According to the previous definitions, no other variable can occur, since we can show the invariant that all variables except session identifiers that occur in the conclusion or in the label of a clause also occur in its hypothesis. However, in practice, the solving algorithm sometimes leaves other variables uninstantiated. In particular, some hypotheses $\mathsf{att}(x)$ may occur. We substitute these variables x with constant patterns $b[\lambda]$ for distinct new λ, as if these facts were proved using clause (Rn).

put corresponding to the clause can be executed. On the other hand, the actions of processes P' that do not satisfy this condition have no counterpart in the derivation, so we do not generate such processes.

More formally, the clause $\mathcal{H}' \xrightarrow{\mathcal{K}',\mathcal{I}'} C$ comes from an output in $(P', \mathcal{K}_{P'}, \mathcal{H}_{P'}, \mathcal{I}_{P'})$ when $\mathcal{H}' \xrightarrow{\mathcal{K}',\mathcal{I}'} C \in [\![P', \mathcal{K}_{P'}, \mathcal{H}_{P'}, \mathcal{I}_{P'}]\!]\mathcal{E}$. Such a clause obviously verifies $\mathcal{K}_{P'} \sqsubseteq \mathcal{K}'$, $\mathcal{H}_{P'} \sqsubseteq \mathcal{H}'$, and $\mathcal{I}_{P'} \sqsubseteq \mathcal{I}'$.

We do not restrict (Res I/O) when $Q \neq 0$, because such a (Res I/O) reduction may just serve in allowing the process Q that follows the output to be executed, and in this case, there may not be any clause in $\mathcal{D}$ that justifies the process $P\{N/x\}$ executed after the input.

- A reduction (Res Constr) consists of the creation by the adversary of a new term M by applying a constructor f to $M_1, \dots, M_n$. The clause $\mathsf{att}(\mathcal{E}(M_1)) \wedge \dots \wedge \mathsf{att}(\mathcal{E}(M_n)) \xrightarrow{Ra} \mathsf{att}(\mathcal{E}(M))$ exactly corresponds to this action. Consequently, we restrict (Res Constr) to the case where $\mathcal{D}$ contains such a corresponding clause. The rules (Res Destr) and (Res New) are restricted similarly.

We focus on restricting the reduction rules mentioned above because, in contrast to the others, they introduce much non-determinism. For instance, (Ins Repl) can be applied an unbounded number of times on a process $!^k P$. Restricting these rules is then necessary in order to have a finite and small set of traces in the restricted semantics. Our algorithm simply consists in exploring all these traces, up to the optimizations described in Section 4.6.

Our algorithm succeeds in reconstructing the well-known attack against the Denning-Sacco protocol. The derivation tree of Figure 3 is first instantiated by substituting λ, λ_A, λ_B for i, i_A, i_B respectively, which yields the derivation tree $\mathcal{D}$. Starting from the initial configuration

$$\begin{aligned}&\mathcal{E}_0 = \{c \mapsto c[\,], s \mapsto s[\,]\}, \mathcal{P}_0 = \{(P_0, \emptyset, \emptyset, \emptyset)\},\\ &\mathcal{S}_0 = \{c\}, \Lambda_0 = \emptyset\end{aligned}$$

we apply (Res New) to create a new name a mapped to pattern $b[\lambda]$ (justified by clause (Rn) in $\mathcal{D}$). Then we start executing P_0 by applying (Res Restr), (Res Let1), and (Res Out) twice, followed by (Res Par), without any restriction. Then we apply (Res Repl) twice. For replication 3, with session identifier λ_A, (Res Repl) is justified by the instance of (DS3) in $\mathcal{D}$, whose label is $(3,1), (\lambda_A, pk(b[\lambda]))$: $(3) \sqsubseteq (3,1)$, $(\lambda_A) \sqsubseteq (\lambda_A, pk(b[\lambda]))$, and $\emptyset \sqsubseteq \mathcal{H}'$. Similarly, for replication 4 with session identifier λ_B, (Res Repl) is justified by the instance of (DS4) in $\mathcal{D}$. Then we obtain the configuration:

$$\begin{aligned}&\mathcal{E}_1 = \mathcal{E}_0[a \mapsto b[\lambda], sk'_A \mapsto sk_A[\,], sk'_B \mapsto sk_B[\,]],\\ &\mathcal{P}_1 = \{(P'_A, (3), \emptyset, (\lambda_A)), (P'_B, (1), \emptyset, (\lambda_B)),\\ &\qquad (!^3 P'_A, \emptyset, \emptyset, \emptyset), (!^4 P'_B, \emptyset, \emptyset, \emptyset)\},\\ &\mathcal{S}_1 = \{c, a, pk(sk'_A), pk(sk'_B)\}, \Lambda_1 = \{\lambda_A, \lambda_B\}\end{aligned}$$

where $P'_A = P_A(sk'_A)$ and $P'_B = P_B(pk(sk'_A), sk'_B)$. Replications cannot be further reduced, because $\mathcal{D}$ would not justify these reductions. We apply (Res Constr) to build $pk(a)$, justified by (Ra_{pk}) in $\mathcal{D}$. Then we execute (Res In) on input 1 with message $pk(a)$, justified by (DS3) in $\mathcal{D}$. Then we can apply (Res Restr), (Res Out), and (Res Nil) without restriction, to finish executing P'_A. We obtain:

$$\begin{aligned}&\mathcal{E}_2 = \mathcal{E}_1[k' \mapsto k[\lambda_A, pk(b[\lambda])]],\\ &\mathcal{P}_2 = \{(P'_B, (1), \emptyset, (\lambda_B)), (!^3 P'_A, \emptyset, \emptyset, \emptyset), (!^4 P'_B, \emptyset, \emptyset, \emptyset)\},\\ &\mathcal{S}_2 = \mathcal{S}_1 \cup \{pk(a), pencrypt(sign(k', sk'_A), pk(a))\}, \Lambda_1\end{aligned}$$

We apply (Res Destr) to obtain $sign(k', sk'_A)$, justified by ($\mathrm{Ra}_{pdecrypt}$) in $\mathcal{D}$. Then, by (Res Destr) again, we compute k', justified by ($\mathrm{Ra}_{getmess}$) in $\mathcal{D}$. (Res Constr) then gives $pencrypt(sign(k', sk'_A), pk(sk'_B))$, justified by ($\mathrm{Ra}_{pencrypt}$) in $\mathcal{D}$. Then we execute (Res In) on input 2 in P'_B with message $pencrypt(sign(k', sk'_A), pk(sk'_B))$, justified by (DS4) in $\mathcal{D}$. We can then apply (Res Let1) twice and (Res Out) which adds $sencrypt(s, k')$ to $\mathcal{S}_i$. Finally, (Res Destr) provides s, justified by ($\mathrm{Ra}_{sdecrypt}$) in $\mathcal{D}$.

4.3. Soundness

The following proposition expresses, as the name says, that the restricted semantics is a restriction of the instrumented semantics:

Proposition 2 *If, by a reduction rule (Res R) of the restricted semantics, we have $\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda \rightsquigarrow \mathcal{E}', \mathcal{P}', \mathcal{S}', \Lambda'$ then the reduction rule (Ins R) in the instrumented semantics yields the same reduction.*

This result is easy to prove, so we do not detail its proof here. Then, by combining Propositions 1 and 2, we obtain the following Theorem 2: learning a secret in the restricted semantics implies learning it in the initial semantics.

Definition 3 Let $\mathcal{D}$ be a closed derivation tree. A *$\mathcal{D}$-restricted trace $\mathcal{T}$ of P_0 from S_0* is a finite sequence $\mathcal{E}_0, \{(P_0, \emptyset, \emptyset, \emptyset)\}, S_0, \emptyset \rightsquigarrow \dots \rightsquigarrow \mathcal{E}', \mathcal{P}', \mathcal{S}', \Lambda'$ of reduction rules of the restricted semantics (restricted by $\mathcal{D}$) where $\mathcal{E}_0 = \{a \mapsto a[\,] \mid a \in fn(P_0) \cup S_0\}$.

Definition 4 The term M *is learnt in the restricted trace $\mathcal{T}$* if and only if $\mathcal{T}$ contains a state $\mathcal{E}, \mathcal{P}, \mathcal{S}, \Lambda$ where $M \in \mathcal{S}$.

Theorem 2 (Soundness) *Let $\mathcal{D}$ be a closed derivation tree. If M_{secret} is learnt in a $\mathcal{D}$-restricted trace of P_0 from S_0, then M_{secret} is learnt in a trace of P_0 from S_0 in the initial semantics.*

This result shows the soundness of our attack reconstruction algorithm. Indeed, the algorithm explores all traces in the restricted semantics. If it finds an attack, then there is also an attack in the initial semantics of the calculus. However, any restriction of the semantics would have this property as well. To justify our choices, we need to prove completeness results, as we do in Section 4.5.

4.4. Termination

The next theorem shows that a closed process has a finite number of restricted traces. This implies the termination of our trace reconstruction algorithm.

Theorem 3 (Termination) *Let $\mathcal{D}$ be a closed derivation tree. There exists a finite number of $\mathcal{D}$-restricted traces of P_0 from S_0, up to renaming of new names and excluding reductions that do not change the state.*

Proof sketch Let us consider the search tree associated to the restricted semantics, starting from the initial configuration of P_0. This tree has finite degree and its depth is bounded by

$$|P_0| \times (\text{number of distinct } \lambda\text{s in } \mathcal{D} + 1) + (\text{number of clauses } (\text{Ra}_f), (\text{Ra}_g), \text{and } (\text{Rn}) \text{ in } \mathcal{D})$$

where $|P_0|$ is the size of P_0. Intuitively, processes can be copied by replications at most as many times as there are distinct λs in $\mathcal{D}$, and each copy generates at most $|P_0|$ execution steps. The second component of the sum bounds the number of reductions (Res Constr), (Res Destr), and (Res New) executed by the adversary. So by König's lemma, this tree is finite. □

4.5. Partial completeness

The following result is a restatement of Theorem 1, which was already proved in previous papers. However, the major novelty of our proof is that it is *constructive*: from an attack, we give an explicit construction of a derivation tree of the fact $\mathsf{att}(\mathcal{E}_0(M_{secret}))$. This construction is important, because it formalizes a correspondence between attacks and derivation trees, which we use in order to state the completeness result. The proofs of these results are omitted because of space constraints.

Theorem 4 *For each trace of P_0 from S_0 in the instrumented semantics in which M_{secret} is learnt, one can build a corresponding closed derivation tree of $\mathsf{att}(\mathcal{E}_0(M_{secret}))$ from the clauses of $\mathcal{R}_{P_0,S_0}$.*

Next, we assume that all outputs are on a public channel or asynchronous (that is, of the form $\overline{M}\langle N\rangle.0$). This restriction is true in most cases for cryptographic protocols: very often, all messages are sent on a public channel, which models an insecure network such as Internet. Asynchronous outputs appear for instance when modeling a mutable value by having the current value available on a private channel. We show that, in this case, if our reconstruction algorithm is given a derivation tree that corresponds to an attack by the above construction, then it finds back an attack.

Theorem 5 (Partial completeness) *Assume that all outputs of P_0 are of the form $\overline{M}\langle N\rangle.P$ with M a name in S_0 not bound in P_0 or $P = 0$. If $\mathcal{D}$ is a closed derivation of $\mathsf{att}(\mathcal{E}_0(M_{secret}))$ from $\mathcal{R}_{P_0,S_0}$ which corresponds to some trace of P_0 from S_0 by Theorem 4, then there exists a $\mathcal{D}$-restricted trace of P_0 from S_0 in which M_{secret} is learnt.*

This result does not hold when we have a synchronous output on a channel not in S_0, because in this case, the reduction rule (Ins I/O) may introduce synchronizations with parts of the process whose execution is not visible on the derivation. For example, for the process

$$P_0 = (\nu c_{priv})(c_{pub}(x)^k.c_{priv}(y)^{k'} \mid \overline{c_{priv}}\langle s\rangle.\overline{c_{pub}}\langle s\rangle)$$

with $S_0 = \{c_{pub}\}$, $\mathsf{att}(s[\,])$ is derivable, by the derivation containing $\mathsf{att}(s[\,])$ as only clause. This derivation corresponds to the execution trace of P_0 by Theorem 4, but P_0 has no restricted trace, because no clause in the derivation corresponds to receiving a message on c_{pub} as needed to execute $c_{pub}(x)^k$ before executing the communication on c_{priv}. (If c_{priv} was public, we could execute $\overline{c_{priv}}\langle s\rangle$ without requiring any prior operation.) Moreover, the part of the process that needs to be executed in order to enable $c_{priv}(y)^{k'}$ may be arbitrarily large, so we cannot hope to succeed in general in this case.

The completeness result does not mean that ProVerif always finds attacks for insecure protocols, even when the constraint on the outputs required by Theorem 5 is satisfied. Indeed, the solving algorithm may not return a derivation tree that corresponds to an attack by Theorem 4. In practice, our algorithm finds the desired trace in the vast majority of cases, as detailed in Section 4.7.

4.6. Optimizations and complexity

We optimize our algorithm by systematically applying all rules except (Res I/O) and (Res In) as soon as they are allowed, even if other reductions are also allowed. This is justified by the following proposition:

Proposition 3 *If there is a $\mathcal{D}$-restricted trace $\mathcal{C}_0 \rightsquigarrow^* \mathcal{C}_i \rightsquigarrow^* \ldots$ in which M_{secret} is learnt, and a reduction rule R different from (Res I/O) and (Res In) is applicable in $\mathcal{C}_i$, then there is a $\mathcal{D}$-restricted trace $\mathcal{C}_0 \rightsquigarrow^* \mathcal{C}_i \overset{R}{\rightsquigarrow} \mathcal{C}'_{i+1} \rightsquigarrow^* \ldots$ in which M_{secret} is learnt and R is applied in $\mathcal{C}_i$.*

Proof sketch This result follows from fairly standard commutations of reduction rules. □

Thanks to this result, the only remaining choices concern rules (Res In) and (Res I/O): we have to choose the received message for (Res In) and the interacting processes for (Res I/O). With the hypotheses of Theorem 5, we can show that we have in fact no choice:

Proposition 4 *With the hypotheses of Theorem 5, our algorithm finds the desired trace without backtracking.*

Proof sketch In this case, for both (Res In) and (Res I/O), we just have to choose the received message. Indeed, for (Res I/O), if several asynchronous outputs send the same message on the same channel, their processes are equal, so it does not matter with which one we reduce. By hypothesis, the derivation $\mathcal{D}$ corresponds to a trace $\mathcal{T}$. In this trace, the input of occurrence k is executed only once for a given value of the session identifiers in $\mathcal{I}$, receiving a certain message N. Then all processes $(P', \mathcal{K}', \mathcal{H}', \mathcal{I}')$ in $\mathcal{T}$ with $\mathcal{K}, k \sqsubseteq \mathcal{K}'$ and $\mathcal{I}, x \sqsubseteq \mathcal{I}'$ have $x = \mathcal{E}(N)$. Moreover, by construction of $\mathcal{D}$ from $\mathcal{T}$ (Theorem 4), all clauses of $\mathcal{D}$ labeled $\mathcal{K}', \mathcal{I}'$ come from such processes, so N is the only possible received message in (Res In) and (Res I/O) knowing $\mathcal{I}$ and k. □

In this case, our algorithm is extremely efficient. The absence of backtracking does not hold in general when the derivation $\mathcal{D}$ does not correspond to a trace: there may be several applications of the clause, each yielding a possible received message, even when the input can in fact be executed only once.

Even in the absence of backtracking, some operations can be fairly costly: reductions (Res In), (Res Constr), (Res Destr), (Res Repl) involve searching the derivation tree for a suitable clause. Furthermore, reductions (Res In), (Res Out), (Res I/O), (Res Constr), (Res Destr) can fail at one point and succeed later, for instance when the attacker has more knowledge, so we have to repeat the test until the reduction succeeds (or the goal of the trace is reached). We optimized this part by keeping as much as possible the information computed when the test failed, to reuse it in future tests for the same reduction:

- As soon as a term M is added to $\mathcal{S}$, we remove $\mathsf{att}(\mathcal{E}(M))$ from the hypotheses of attacker clauses of $\mathcal{D}$. Then the reductions (Res Constr) or (Res Destr) are triggered simply when an attacker clause without hypothesis appears. This avoids the repeated testing of whether the arguments of the constructor or destructor are in $\mathcal{S}$.
- We cache the set of possible messages found in the derivation $\mathcal{D}$ for each input, so that this set is computed only once for each newly created input process.
- For each input and output channel and each possible input message, when we test whether it is in $\mathcal{S}$, we remember that this test has been done, so that only the newly added elements of $\mathcal{S}$ need to be tested next time.

As an additional optimization, we begin with translating all patterns in the derivation tree into their corresponding terms, by replacing patterns $a[\mathcal{I}]$ with fresh names a_i. The association between names and patterns is stored so that the name a_i is reused when executing the restriction that creates the name of pattern $a[\mathcal{I}]$. The rest of the algorithm then never manipulates patterns.

Moreover, when a derivation tree contains several subtrees deriving the same att fact, we keep only one of these subtrees and ignore the others. (This must not be done for mess facts, because sending several times the same message on the same private channel may be useful in the trace.)

The following proposition evaluates the worst-case complexity of our algorithm when the hypothesis of Theorem 5 is satisfied. (The general case is most probably exponential, because of backtracking.) The proof of this result is given in the appendix.

Proposition 5 *Let $|\mathcal{D}|$ be the size of the derivation tree, L the length of returned trace plus the number of free names of P_0, T the maximum size of a term, $|\mathcal{S}|$ the maximum number of terms in $\mathcal{S}$ in any configuration of the trace, $|P|$ the maximum size of a process in any configuration of the trace, $|\mathcal{C}|$ the maximum number of parallel processes in any configuration of the trace.*

Assuming the hypothesis of Theorem 5, the worst-case complexity of our algorithm is $\mathcal{O}(L \times (T|\mathcal{C}|^2 + T|\mathcal{S}| + |\mathcal{D}| + |P|))$. When furthermore all channels are in S_0, the complexity decreases to $\mathcal{O}(L \times (|\mathcal{C}| + T|\mathcal{S}| + |\mathcal{D}| + |P|))$.

4.7. Experimental results

Our algorithm could reconstruct attacks against the following protocols of the literature: Needham-Schroeder public-key, Denning-Sacco, several versions of Woo-Lam, flawed versions of Yahalom and Otway-Rees. It also reconstructs traces that show that the end of the protocol is reachable for more complex, secure protocols such as Skeme [19] or the web services protocols that serve as example to the verifier TulaFale [8]. For all these examples, the total time for finding derivation trees is 1.4 s and the trace reconstruction time is 0.1 s on a Pentium M 1.8 GHz.

On the certified email protocol studied in [3], our tool reconstructs 2 out of 4 traces. The failure cases come from the presence of synchronous outputs on private channels. The total time for finding derivation trees is 0.8 s and the trace reconstruction time is 25 ms.

On the JFK protocol [7], our tool reconstructs 2 out of 8 traces. In the failure cases, the derivation tree found by ProVerif does not correspond to a trace (although the trace exists). For JFK and the certified email protocol mentioned above, our algorithm explores 1 to 6 traces depending on the desired goal; in all other examples, it explores only one trace, which is the desired attack. (This is coherent with Proposition 4.) For JFK, the total time for finding derivation trees is 2.5 s and the trace reconstruction time is 1.6 s (0.3 s when backtracking is disabled—the algorithm succeeds exactly in the same cases as with backtracking). A preliminary modification of the derivation tree allows us to find all traces: when the same input (same occurrence label and same session identifiers) receives several different messages in the derivation, we unify these messages. Indeed, when the derivation corresponds to a trace, these messages must be equal. With this transformation, traces are found in all 8 cases, in 0.3 s and without backtracking.

Another interesting example is the family of $\mathrm{f}^n\mathrm{g}^n$ protocols [21]: the n-th protocol of the family has an attack using n parallel sessions. ProVerif correctly detects and rebuilds this attack, without any prior information on n. This illustrates its ability to handle an unbounded number of sessions. (Tested up to $n = 200$; the time for finding a derivation was 47 s and the trace reconstruction time 67 s. For instance, for $n = 50$, these times were respectively 0.4 s and 0.5 s. This protocol is the only example we have on which trace reconstruction is longer than finding a derivation. The complexity of the trace reconstruction algorithm on the family of $\mathrm{f}^n\mathrm{g}^n$ protocols is $\mathcal{O}(n^4)$. This result is not a consequence of Proposition 5. One needs to use the specificities of the $\mathrm{f}^n\mathrm{g}^n$ protocols to obtain it. Experimentally, for $n = 200$, the most time consuming parts are the execution of n^2 restrictions and the translation of patterns into terms.)

4.8. On a variant of this algorithm

We could also have exploited the tree structure of the derivation in order to reconstruct a trace, executing reductions in the partial order given by the derivation tree, from the leaves to the root. This variant would use the same restricted semantics as the one given above, except that each reduction should be justified by a particular node of the derivation, the *current* node, instead of by any node, and the output and input processes of (Res I/O) should correspond respectively to the current node and its father. The current node would move in the derivation tree according to a postfix depth-first search. (Each node would be visited after visiting its sons from left to right, so the current node would start at the left-most leaf and end at the root.) One would change the current node when one executes the action corresponding to the node (either the output $\overline{M}\langle N\rangle$ is executed for a node labeled $H \xRightarrow{\mathcal{K},\mathcal{I}} \mathsf{mess}(\mathcal{E}(M), \mathcal{E}(N))$ or a constructor or destructor is applied to $M_1, \dots, M_n$, yielding M for a node labeled $\mathsf{att}(\mathcal{E}(M_1)) \wedge \dots \wedge \mathsf{att}(\mathcal{E}(M_n)) \xRightarrow{Ra} \mathsf{att}(\mathcal{E}(M)))$ or when this action has already been executed before in the trace. (The second case is necessary because the derivation tree may contain several subtrees that correspond to the same actions, as the dashed subtree of Figure 3, and because an output may be executed at some point just to be able to execute the process that follows it, while the output itself is useful at some other point in the derivation.)

The soundness and termination results obviously also hold for this algorithm, with the same proofs, as well as the absence of backtracking (there is in fact no choice). We believe that the partial completeness theorem would also hold, although its proof would be more complicated. (We would have to show that the updates of the current node are done correctly.)

One might think that this variant would be even more efficient than the one presented above. Our first tests with this variant indicate that it can indeed be up to 4 times faster, but we also have examples on which it is up to 2 times slower. On $\mathrm{f}^{200}\mathrm{g}^{200}$, it is 1.2 times faster. (Since this variant never backtracks, and the previous algorithm with backtracking disabled also finds the traces that this variant finds, we compare with the previous algorithm with backtracking disabled.) Anyway, in practice, the difference in runtime does not matter much, since in most cases the time for reconstructing a trace is largely dominated by the time for finding a derivation.

Moreover, our algorithm succeeds more often in the presence of synchronous outputs on private channels (outputs $\overline{M}\langle N\rangle.P$ with $M \notin \mathcal{S}$ and a non-empty P). Indeed, if at some point in the derivation the output should be executed only so that P can be executed, the variant of this section will be blocked, even if later in the search of the derivation, a node justifies the output. In contrast, our previous algorithm will succeed in this case (but it will fail if no node in the derivation justifies the output). This is the reason why the variant of this section fails in one more case than the previous algorithm for the certified email protocol [3]. (For our other examples, it succeeds in exactly the same cases.) We preferred increasing our chances of success, even if we obtain a slightly less efficient algorithm.

5. Conclusion

We have presented an efficient algorithm for reconstructing attacks against cryptographic protocols. We have proved its soundness and termination, as well as a partial completeness result. By lack of space, some extensions have been omitted in this paper. For instance, we also handle cryptographic primitives defined by equations as in the applied pi calculus [6]. We have considered only secrecy properties in this paper, but we have also extended our technique to the case of correspondence properties between events [27]. These properties, of the form "if some event has been executed, then some other events must have been executed", can be used to formalize authenticity properties. ProVerif can prove correspondence properties as shown in [10, 3]. Our trace reconstruction algorithm is able to reconstruct attacks against non-injective correspondence properties (those in which the number of executions of events does not matter). For future work, it would be interesting to extend it to injective correspondence properties as well as process equivalence properties [11]. This is however more difficult because our techniques for proving these properties are more approximate than for secrecy or non-injective correspondences, so the failure of the proof is more likely not to correspond to an attack.

Acknowledgments We thank Cédric Fournet for motivating our work on the reconstruction of attacks and for suggesting the idea of guiding reductions by the derivation tree. We also thank Jérôme Feret and the anonymous reviewers for their helpful comments on this work.

References

[1] M. Abadi. Security Protocols and their Properties. In F. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, 2000. Volume for the 20th International Summer School on Foundations of Secure Computation, held in Marktoberdorf, Germany (1999).

[2] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 33–44, Portland, Oregon, Jan. 2002. ACM Press.

[3] M. Abadi and B. Blanchet. Computer-Assisted Verification of a Protocol for Certified Email. In R. Cousot, editor, *Static Analysis, 10th International Symposium (SAS'03)*, volume 2694 of *Lecture Notes on Computer Science*, pages 316–335, San Diego, California, June 2003. Springer.

[4] M. Abadi and B. Blanchet. Secrecy Types for Asymmetric Communication. *Theoretical Computer Science*, 298(3):387–415, Apr. 2003. Special issue FoSSaCS'01.

[5] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[6] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, London, United Kingdom, Jan. 2001. ACM Press.

[7] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, K. Keromytis, and O. Reingold. Just Fast Keying: Key Agreement in a Hostile Internet. *ACM Transactions on Information and System Security*, 7(2):242–273, May 2004.

[8] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A Security Tool for Web Services. In *Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *Lecture Notes on Computer Science*, pages 197–222, Leiden, The Netherlands, Nov. 2003. Springer. Paper and tool available at `http://securing.ws/`.

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

[10] B. Blanchet. From Secrecy to Authenticity in Security Protocols. In M. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, Sept. 2002. Springer.

[11] B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[12] B. Blanchet and A. Podelski. Verification of Cryptographic Protocols: Tagging Enforces Termination. In A. Gordon, editor, *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, volume 2620 of *Lecture Notes on Computer Science*, pages 136–152, Warsaw, Poland, Apr. 2003. Springer.

[13] P. J. Broadfoot and A. W. Roscoe. Capturing Parallel Attacks within the Data Independence Framework. In *15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 147–159, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.

[14] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

[15] D. E. Denning and G. M. Sacco. Timestamps in Key Distribution Protocols. *Commun. ACM*, 24(8):533–536, Aug. 1981.

[16] N. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. Multiset Rewriting and the Complexity of Bounded Security Protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[17] A. Gordon and A. Jeffrey. Types and Effects for Asymmetric Cryptographic Protocols. In *15th IEEE Computer Security Foundations Workshop (CSFW 2002)*, pages 77–91, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.

[18] J. Goubault-Larrecq. Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve ? In *Actes 15èmes journées francophones sur les langages applicatifs (JFLA'04)*, Sainte-Marie-de-Ré, France, Jan. 2004. INRIA.

[19] H. Krawczyk. SKEME: A Versatile Secure Key Exchange Mechanism for Internet. In *Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

[20] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes on Computer Science*, pages 147–166. Springer, 1996.

[21] J. Millen. A Necessarily Parallel Attack. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, Trento, Italy, July 1999.

[22] J. Millen and V. Shmatikov. Constraint Solving for Bounded-Process Cryptographic Protocol Analysis. In *Proc. 8th ACM Conference on Computer and Communications Security (CCS '01)*, pages 166–175, 2001.

[23] D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. *Science of Computer Programming*, 47(2–3):177–202, 2003.

[24] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[25] A. W. Roscoe and P. J. Broadfoot. Proving Security Protocols with Model Checkers by Data Independence Techniques. *Journal of Computer Security*, 7(2, 3):147–190, 1999.

[26] P. Selinger. Models for an Adversary-Centric Protocol Logic. In J. Goubault-Larrecq, editor, *Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification*, volume 55(1) of *Electronic Notes in Theoretical Computer Science*, pages 73–88, Paris, France, July 2001.

[27] T. Y. C. Woo and S. S. Lam. A Semantic Model for Authentication Protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 1993.

Appendix: Complexity

Proof sketch (of Proposition 5) We give the complexity of our implementation for the language presented in this paper. The complexity could be improved, for instance by using more clever data structures such as balanced binary search trees for $\mathcal{S}$. However, such structures would not be able to handle the extension to equational theories mentioned in the conclusion, because we need to compare terms modulo the equational theory instead of syntactically in this extension. That is why we do not use them, and simply represent $\mathcal{S}$ by a list.

We first note that each reduction creates at most two new processes, so at most $2L$ different processes are considered in the whole trace.

For each free name a of P_0, we remove hypotheses $\mathsf{att}(a[\,])$ from all attacker clauses in $\mathcal{D}$. The time for this step is dominated by $\mathcal{O}(L|\mathcal{D}|)$. Then we perform the following actions:

- Reductions (Res In) or (Res I/O): For each new input process (so at most $2L$ times in the whole run), we look for corresponding clauses in the derivation tree. This takes time $\mathcal{O}(|\mathcal{D}|)$. At most one possible message term is then found (by Proposition 4), and this term is stored in a cache for further tests. For each input process, we test whether the channel is in $\mathcal{S}$. Thanks to the cache, the same part of $\mathcal{S}$ is scanned at most once for each new input process (there are at most $2L$ such processes in the whole run), so this test takes $\mathcal{O}(LT|\mathcal{S}|)$. If the channel is in $\mathcal{S}$, we test whether the previously found message term is in $\mathcal{S}$, which also takes $\mathcal{O}(LT|\mathcal{S}|)$. Then we execute (Res In). For each executed (Res In) reduction (at most L times), we substitute a term in a process, in time $\mathcal{O}(|P|)$, and update of the configuration, in time $\mathcal{O}(|\mathcal{C}|)$. If the channel is not in $\mathcal{S}$, we look for a corresponding asynchronous output process, which takes time $\mathcal{O}(LT|\mathcal{C}|^2)$. If we find one, we execute (Res I/O), which also takes time $\mathcal{O}(|P| + |\mathcal{C}|)$ for each reduction (Res I/O) and there are at most L of them.
  Total: $\mathcal{O}(L \times (|\mathcal{D}| + T|\mathcal{S}| + T|\mathcal{C}|^2 + |P|))$.
- Reductions (Res Out): For each output process, we test whether the channel is in $\mathcal{S}$ which takes $\mathcal{O}(LT|\mathcal{S}|)$ as the input case. If the channel is in $\mathcal{S}$, we execute (Res Out). For each (Res Out) reduction (at most L times), if the message term M of this output is not in $\mathcal{S}$, we add it, and remove $\mathsf{att}(\mathcal{E}(M))$ from all hypotheses of attacker clauses of $\mathcal{D}$. This takes time $\mathcal{O}(|\mathcal{D}| + T|\mathcal{S}|)$. If an attacker clause without hypothesis now occurs in $\mathcal{D}$, a reduction (Red Constr) or (Red Destr) is triggered. Furthermore, updating the configuration takes $\mathcal{O}(|\mathcal{C}|)$.
  Total: $\mathcal{O}(L \times (T|\mathcal{S}| + |\mathcal{D}| + |\mathcal{C}|))$.
- Reductions (Res Constr) or (Res Destr): If the term M built by this reduction is not in $\mathcal{S}$, we add it, and remove $\mathsf{att}(\mathcal{E}(M))$ from all hypotheses of attacker clauses of $\mathcal{D}$. This takes time $\mathcal{O}(T|\mathcal{S}| + |\mathcal{D}|)$ for each such reduction, and there are at most L of them.
  Total: $\mathcal{O}(L \times (T|\mathcal{S}| + |\mathcal{D}|))$.
- Reductions (Res Repl): For each (Res Repl) reduction (at most L times), we look for a suitable clause in $\mathcal{D}$, in time $\mathcal{O}(|\mathcal{D}|)$, we copy the process, in time $\mathcal{O}(|P|)$, and we update of the configuration, in time $\mathcal{O}(|\mathcal{C}|)$.
  Total: $\mathcal{O}(L \times (|\mathcal{D}| + |P| + |\mathcal{C}|))$.
- Reductions (Res New): In fact, this reduction is only executed at the beginning of the run, before all other reductions. Each execution of (Res New) takes $\mathcal{O}(|\mathcal{D}|)$: we look for a suitable clause $\mathsf{att}(b[\lambda])$ in $\mathcal{D}$, and then remove $\mathsf{att}(b[\lambda])$ from all hypotheses of attacker clauses of $\mathcal{D}$.
- All other reductions can be executed at most in time $\mathcal{O}(|P| + |\mathcal{C}|)$, and there are at most L of them.

By adding the given times, we obtain the announced complexity. When all channels are in S_0, we never try I/O reductions with asynchronous outputs, so the term $T|\mathcal{C}|^2$ disappears. □