# Automatic Verification of Correspondences for Security Protocols*

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

July 31, 2008

**Abstract**

We present a new technique for verifying correspondences in security protocols. In particular, correspondences can be used to formalize authentication. Our technique is fully automatic, it can handle an unbounded number of sessions of the protocol, and it is efficient in practice. It significantly extends a previous technique for the verification of secrecy. The protocol is represented in an extension of the pi calculus with fairly arbitrary cryptographic primitives. This protocol representation includes the specification of the correspondence to be verified, but no other annotation. This representation is then translated into an abstract representation by Horn clauses, which is used to prove the desired correspondence. Our technique has been proved correct and implemented. We have tested it on various protocols from the literature. The experimental results show that these protocols can be verified by our technique in less than 1 s.

## 1 Introduction

The verification of security protocols has already been the subject of numerous research works. It is particularly important since the design of protocols is error-prone, and errors cannot be detected by testing, since they appear only in the presence of a malicious adversary. An important trend in this area aims to verify protocols in the so-called Dolev-Yao model [40], with an unbounded number of sessions, while relying as little as possible on human intervention. While protocol insecurity is NP-complete for a bounded number of sessions [66], it is undecidable for an unbounded number of sessions [42]. Hence, automatic verification for an unbounded number of sessions cannot be achieved for all protocols. It is typically achieved using language-based techniques such as typing or abstract interpretation, which can handle infinite-state systems thanks to safe approximations. These techniques are not complete (a correct protocol

*This paper is an updated and extended version of [13] and [14].

†This research has been done within the INRIA ABSTRACTION project-team (common with the CNRS and the ÉNS).

can fail to typecheck, or false attacks can be found by abstract interpretation tools), but they are sound (when they do not find attacks, the protocol is guaranteed to satisfy the considered property). This is important for the certification of protocols.

Our goal in this paper is to extend previous work in this line of research by providing a fully automatic technique for verifying correspondences in security protocols, without bounding the number of sessions of the protocol. Correspondences are properties of the form: if the protocol executes some event, then it must have executed some other events before[1]. We consider a rich language of correspondences, in which the events that must have been executed can be described by a logical formula containing conjunctions and disjunctions. Furthermore, we consider both non-injective correspondences (if the protocol executes some event, then it must have executed some other events at least once) and injective correspondences (if the protocol executes some event $n$ times, then it must have executed some other events at least $n$ times). Correspondences, initially named correspondence assertions [72], and the similar notion of agreement [55] were first introduced to model authentication. Intuitively, a protocol authenticates $A$ to $B$ if, when $B$ thinks he talks to $A$, then he actually talks to $A$. When $B$ thinks he has run the protocol with $A$, he executes an event $e(A, B)$. When $A$ thinks she runs the protocol with $B$, she executes another event $e'(A, B)$. Authentication is satisfied when, if $B$ executes his event $e(A, B)$, then $A$ has executed her event $e'(A, B)$. Several variants along this scheme appear in the literature and, as we show below, our technique can handle most of them. Our correspondences can also encode secrecy, as follows. A protocol preserves the secrecy of some value $M$ when the adversary cannot obtain $M$. We associate an "event" $\mathrm{attacker}(M)$ to the fact that the adversary obtains $M$, and represent the secrecy of $M$ as "$\mathrm{attacker}(M)$ cannot be executed", that is, "if $\mathrm{attacker}(M)$ has been executed, then false." More complex properties can also be specified by our correspondences, for example that all messages of the protocol have been sent in order; this feature was used in [3].

Our technique is based on a substantial extension of a previous verification technique for secrecy [1, 13, 70]. More precisely, the protocol is represented in the process calculus introduced in [1], which is an extension of the pi calculus with fairly arbitrary cryptographic primitives. This process calculus is extended with events, used in the statement of correspondences. These events are the only required annotation of the protocol; no annotation is needed to help the tool proving correspondences. The protocol is then automatically translated into a set of Horn clauses. This translation requires significant extensions with respect to the translation for secrecy given in [1], and can be seen as an implementation of a type system, as in [1]. Some of these extensions improve the precision of the analysis, in particular to avoid merging different nonces. Other extensions define the translation of events. Finally, this set of Horn clauses is passed to a resolution-based solver, similar to that of [13, 21, 70]. Some minor extensions of this solver are required to prove correspondences. This solver does not always terminate, but we show in Section 8.1 that it terminates for a large class of well-designed protocols, named *tagged protocols*. Our experiments also demonstrate that, in practice, it terminates on many examples of protocols.

The main advantages of our method can be summarized as follows. It is fully auto-

[1] In the CSP terminology, our events correspond to CSP signal events.

matic; the user only has to code the protocol and the correspondences to prove. It puts no bounds on the number of sessions of the protocol or the size of terms that the adversary can manipulate. It can handle fairly general cryptographic primitives, including shared-key encryption, public-key encryption, signatures, one-way hash functions, and Diffie-Hellman key agreements. It relies on a precise semantic foundation. One limitation of the technique is that, in rare cases, the solving algorithm does not terminate. The technique is also not complete: the translation into Horn clauses introduces an abstraction, which forgets the number of repetitions of each action [17]. This abstraction is key to the treatment of an unbounded number of sessions. Due to this abstraction, the tool provides sufficient conditions for proving correspondences, but can fail on correct protocols. Basically, it fails to prove protocols that first need to keep some value secret and later reveal it (see Section 5.2.2). In practice, the tool is still very precise and, in our experiments, it always succeeded in proving protocols that were correct.

Our technique is implemented in the protocol verifier ProVerif, available at `http://www.proverif.ens.fr/`.

**Comparison with Other Papers on ProVerif** As mentioned above, this paper extends previous work on the verification of secrecy [1] in order to prove correspondences. Secrecy (defined as the impossibility for the adversary to compute the secret) and correspondences are trace properties. Other papers deal with the proof of certain classes of observational equivalences, *i.e.*, that the adversary cannot distinguish certain processes: [15, 16] deal with the proof of strong secrecy, *i.e.*, that the adversary cannot see when the value of a secret changes; [19] deals with the proof of equivalences between processes that differ only by the terms that they contain. Moreover, [19] also explains how to handle cryptographic primitives defined by equational theories (instead of rewrite rules) and how to deal with guessing attacks against weak secrets.

As shown in [21], the resolution algorithm terminates for tagged protocols. The present paper extends this result in Section 8.1, by providing a characterization of tagged protocols at the level of processes instead of at the level of Horn clauses.

ProVerif can also reconstruct an attack using a derivation from the Horn clauses, when the proof of a secrecy property fails [6]. Although the present paper does not detail this point, this work has also been extended to the reconstruction of attacks against non-injective correspondences.

Finally, [2], [3], and [20] present three case studies done at least partly using ProVerif: [2] studies a certified email protocol, [3] studies the Just Fast Keying protocol, and [20] studies the Plutus secure file system. These case studies rely partly on the results presented in this paper.

**Related Work** We mainly focus on the works that automatically verify correspondences and authentication for security protocols, without bounding the number of sessions.

The NRL protocol analyzer [43, 58], based on narrowing in rewriting systems, can verify correspondences defined in a rich language of logical formulae [69]. It is sound and complete, but does not always terminate. Our Horn clause representation is more abstract than the representation of NRL, which should enable us to terminate more

often and be more efficient, while remaining precise enough to prove most desired properties.

Gordon and Jeffrey designed a system named Cryptic for verifying authentication by typing in security protocols [46–48]. They handle shared-key and public-key cryptography. Our system allows more general cryptographic primitives (including hash functions and Diffie-Hellman key agreements). Moreover, in our system, no annotation is needed, whereas, in Cryptic, explicit type casts and checks have to be manually added. However, Cryptic has the advantage that type checking always terminates, whereas, in some rare cases, our analyzer does not.

Bugliesi et al. [26] define another type system for proving authentication in security protocols. The main advantage of their system is that it is compositional: it allows one to prove independently the correctness of the code of each role of the protocol. However, the form of messages is restricted to certain tagged terms. This approach is compared with Cryptic in [25].

Backes et al. [10] prove secrecy and authentication for security protocols, using an abstract-interpretation-based analysis. This analysis builds a causal graph, which captures the causality among program events; the security properties are proved by traversing this graph. This analysis can handle an unbounded number of sessions of the protocol; it always terminates, at the cost of additional abstractions, which may cause false attacks. It handles shared-key and public-key cryptography, but not Diffie-Hellman key agreements. It assumes that the messages are typed, so that names can be distinguished from other terms.

Bodei et al. [22] show message authentication via a control flow analysis on a process calculus named Lysa. Like [10], they handle shared-key and public-key cryptography, and their analysis always terminates, at the cost of additional abstractions. The notion of authentication they prove is different from ours: they show message authentication rather than entity authentication.

Debbabi et al. [37] also verify authentication thanks to a representation of protocols by inference rules, very similar to our Horn clauses. However, they verify a weaker notion of authentication (corresponding to aliveness: if $B$ terminates the protocol, then $A$ must have been alive at some point before), and handle only shared-key encryption.

A few other methods require little human effort, while supporting an unbounded number of runs: the verifier of [52], based on rank functions, can prove the correctness of or find attacks against protocols with atomic symmetric or asymmetric keys. Theorem proving [64] often requires manual intervention of the user. An exception to this is [33], but it deals only with secrecy. The theorem prover TAPS [31] often succeeds without or with little human intervention.

Model checking [54, 60] in general implies a limit on the number of sessions of the protocol. This problem has been tackled by [23, 24, 65]. They recycle nonces, to use only a finite number of them in an infinite number of runs. The technique was first used for sequential runs, then generalized to parallel runs in [24], but with the additional restriction that the agents must be "factorisable". (Basically, a single run of the agent has to be split into several runs such that each run contains only one fresh value.)

Strand spaces [45] are a formalism for reasoning about security protocols. They have been used for elegant manual proofs of authentication [50]. The automatic tool

Athena [67] combines model checking and theorem proving, and uses strand spaces to reduce the state space. Scyther [34] uses an extension of Athena's method with trace patterns to analyze simultaneously a group of traces. These tools still sometimes limit the number of sessions to guarantee termination.

Amadio and Prasad [7] note that authentication can be translated into secrecy, by using a judge process. The translation is limited in that only one message can be registered by the judge, so the verified authentication property is not exactly the same as ours.

**Outline** Section 2 introduces our process calculus. Section 3 defines the correspondences that we verify, including secrecy and various notions of authentication. Section 4 outlines the main ideas behind our technique for verifying correspondences. Section 5 explains the construction of Horn clauses and shows its correctness, Section 6 describes our solving algorithm and shows its correctness, and Section 7 applies these results to the proof of correspondences. Section 8 discusses the termination of our algorithm: it shows termination for tagged protocols and how to obtain termination more often in the general case. Section 9 presents some extensions to our framework. Section 10 gives our experimental results on a selection of security protocols of the literature, and Section 11 concludes. The detailed proofs of our results can be found in the companion technical report [18].

# 2 The Process Calculus

In this section, we present the process calculus that we use to represent security protocols: we give its syntax, semantics, and illustrate it on an example protocol.

## 2.1 Syntax and Informal Semantics

Figure 1 gives the syntax of terms (data) and processes (programs) of our calculus. The identifiers $a$, $b$, $c$, $k$, and similar ones range over names, and $x$, $y$, and $z$ range over variables. The syntax also assumes a set of symbols for constructors and destructors; we often use $f$ for a constructor and $g$ for a destructor.

Constructors are used to build terms. Therefore, the terms are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$; the terms are untyped. On the other hand, destructors do not appear in terms, but only manipulate terms in processes. They are partial functions on terms that processes can apply. The process $let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$ tries to evaluate $g(M_1, \ldots, M_n)$; if this succeeds, then $x$ is bound to the result and $P$ is executed, else $Q$ is executed. More precisely, the semantics of a destructor $g$ of arity $n$ is given by a set $\mathrm{def}(g)$ of rewrite rules of the form $g(M_1, \ldots, M_n) \to M$ where $M_1, \ldots, M_n, M$ are terms without names, and the variables of $M$ also occur in $M_1, \ldots, M_n$. We extend these rules by $g(M'_1, \ldots, M'_n) \to M'$ if and only if there exist a substitution $\sigma$ and a rewrite rule $g(M_1, \ldots, M_n) \to M$ in $\mathrm{def}(g)$ such that $M'_i = \sigma M_i$ for all $i \in \{1, \ldots, n\}$, and $M' = \sigma M$. We assume that the set $\mathrm{def}(g)$ is finite. (It usually contains one or two rules in examples.) We define destructors by rewrite rules instead of the equalities

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

Figure 1: Syntax of the process calculus

used in [1]. This definition allows destructors to yield several different results non-deterministically. (Non-deterministic rewrite rules are used in our modeling of Diffie-Hellman key agreements; see Section 9.1). Using constructors and destructors, we can represent data structures and cryptographic operations as summarized in Figure 2. (We present only probabilistic public-key encryption because, in the computational model, a secure public-key encryption algorithm must be probabilistic. We have chosen to present deterministic signatures; we could easily model probabilistic signatures by adding a third argument $r$ containing the random coins, as for encryption. The coins should be chosen using a restriction $(\nu a)$ which creates a fresh name $a$, representing a fresh random number.)

Constructors and destructors can be public or private. The public ones can be used by the adversary, which is the case when not stated otherwise. The private ones can be used only by honest participants. They are useful in practice to model tables of keys stored in a server, for instance. A public constructor $host$ computes a host name from a long-term secret key, and a private destructor $getkey$ returns the key from the host name, and simulates a lookup in a table of pairs (host name, key). Using a public constructor $host$ allows the adversary to create and register any number of host names and keys. However, since $getkey$ is private, the adversary cannot compute a key from the host name, which would break all protocols: host names are public while keys of honest participants are secret.

The process calculus provides additional instructions for executing events, which will be used for specifying correspondences. The process $\texttt{event}(M).P$ executes the event $\texttt{event}(M)$, then executes $P$.

The other constructs in the syntax of Figure 1 are standard; most of them come from the pi calculus. The input process $M(x).P$ inputs a message on channel $M$, and executes $P$ with $x$ bound to the input message. The output process $\overline{M}\langle N \rangle.P$ outputs

**Tuples:**
Constructor: tuple $ntuple(x_1, \ldots, x_n)$
Destructors: projections $ith_n(ntuple(x_1, \ldots, x_n)) \rightarrow x_i$
**Shared-key encryption:**
Constructor: encryption of $x$ under the key $y$, $sencrypt(x, y)$
Destructor: decryption $sdecrypt(sencrypt(x, y), y) \rightarrow x$
**Probabilistic shared-key encryption:**
Constructor: encryption of $x$ under the key $y$ with random coins $r$, $sencrypt_p(x, y, r)$
Destructor: decryption $sdecrypt_p(sencrypt_p(x, y, r), y) \rightarrow x$
**Probabilistic public-key encryption:**
Constructors: encryption of $x$ under the key $y$ with random coins $r$, $pencrypt_p(x, y, r)$
public key generation from a secret key $y$, $pk(y)$
Destructor: decryption $pdecrypt_p(pencrypt_p(x, pk(y), r), y) \rightarrow x$
**Signatures:**
Constructors: signature of $x$ with the secret key $y$, $sign(x, y)$
public key generation from a secret key $y$, $pk(y)$
Destructors: signature verification $checksignature(sign(x, y), pk(y)) \rightarrow x$
message without signature $getmessage(sign(x, y)) \rightarrow x$
**Non-message-revealing signatures:**
Constructors: signature of $x$ with the secret key $y$, $nmrsign(x, y)$
public key generation from a secret key $y$, $pk(y)$
constant $true$
Destructor: verification $nmrchecksign(nmrsign(x, y), pk(y), x) \rightarrow true$
**One-way hash functions:**
Constructor: hash function $h(x)$
**Table of host names and keys**
Constructor: host name from key $host(x)$
Private destructor: key from host name $getkey(host(x)) \rightarrow x$

Figure 2: Constructors and destructors

the message $N$ on the channel $M$ and then executes $P$. We allow communication on channels that can be arbitrary terms. (We could adapt our work to the case in which channels are only names.) Our calculus is monadic (in that the messages are terms rather than tuples of terms), but a polyadic calculus can be simulated since tuples are terms. It is also synchronous (in that a process $P$ is executed after the output of a message). The nil process $0$ does nothing. The process $P \mid Q$ is the parallel composition of $P$ and $Q$. The replication $!P$ represents an unbounded number of copies of $P$ in parallel. The restriction $(\nu a)P$ creates a new name $a$ and then executes $P$. The conditional $if\ M = N\ then\ P\ else\ Q$ executes $P$ if $M$ and $N$ reduce to the same term at runtime; otherwise, it executes $Q$. We define $let\ x = M\ in\ P$ as syntactic sugar for $P\{M/x\}$. As usual, we may omit an $else$ clause when it consists of $0$.

The name $a$ is bound in the process $(\nu a)P$. The variable $x$ is bound in $P$ in the processes $M(x).P$ and $let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in $P$, respectively. A process is closed if

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

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

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

$$E, \mathcal{P} \cup \{\, (\nu a)P \,\} \rightarrow E \cup \{a'\}, \mathcal{P} \cup \{\, P\{a'/a\} \,\} \qquad \text{(Red Res)}$$

where $a' \notin E$.

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

$$E, \mathcal{P} \cup \{\, \mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P\{M'/x\} \,\}$$

if $g(M_1, \ldots, M_n) \rightarrow M'$ (Red Destr 1)

$$E, \mathcal{P} \cup \{\, \mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} \qquad \text{(Red Destr 2)}$$

if there exists no $M'$ such that $g(M_1, \ldots, M_n) \rightarrow M'$

$$E, \mathcal{P} \cup \{\, \mathit{if}\ M = M\ \mathit{then}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\} \qquad \text{(Red Cond 1)}$$

$$E, \mathcal{P} \cup \{\, \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} \qquad \text{(Red Cond 2)}$$

if $M \neq N$

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

Figure 3: Operational semantics

it has no free variables; it may have free names. We identify processes up to renaming of bound names and variables. We write $\{M_1/x_1, \ldots, M_n/x_n\}$ for the substitution that replaces $x_1$, ..., $x_n$ with $M_1$, ..., $M_n$, respectively.

## 2.2 Operational Semantics

A semantic configuration is a pair $E, \mathcal{P}$ where the environment $E$ is a finite set of names and $\mathcal{P}$ is a finite multiset of closed processes. The environment $E$ must contain at least all free names of processes in $\mathcal{P}$. The configuration $\{a_1, \ldots, a_n\}, \{P_1, \ldots, P_n\}$ corresponds intuitively to the process $(\nu a_1) \ldots (\nu a_n)(P_1 \mid \ldots \mid P_n)$. The semantics of the calculus is defined by a reduction relation $\rightarrow$ on semantic configurations, shown in Figure 3. The rule (Red Res) is the only one that uses renaming. This is important so that the parameters of events are not renamed after the execution of the event, to be able to compare them with the parameters of events executed later. This semantics is superficially different from those of [1, 14], which were defined using a structural congruence relation and a reduction relation on processes. The new semantics (in particular the renaming point mentioned above) provides simplifications in the definitions of correspondences (Definitions 2, 3, 6, 7, and 9) and in the proofs that correspondences hold.

## 2.3 Example

As a running example, we consider a simplified version of the Needham-Schroeder public-key protocol [61], with the correction by Lowe [54], in which host names are replaced by public keys, which makes interaction with a server useless. (The version tested in the benchmarks is the full version. Obviously, our tool can verify much more complex protocols; we use this simple example for illustrative purposes.) The protocol contains the following messages:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B: & \{a, pk_A\}_{pk_B} \\ \text{Message 2.} & B \rightarrow A: & \{a, b, pk_B\}_{pk_A} \\ \text{Message 3.} & A \rightarrow B: & \{b\}_{pk_B} \end{array}$$

$A$ first sends to $B$ a nonce (fresh name) $a$ encrypted under the public key of $B$. $B$ decrypts this message using his secret key $sk_B$ and replies with the nonce $a$, a fresh nonce he chooses $b$, and its own public key $pk_B$, all encrypted under $pk_A$. When $A$ receives this message, she decrypts it. When $A$ sees the nonce $a$, she is convinced that $B$ answered since only $B$ can decrypt the first message and obtain $a$. Then $A$ replies with the nonce $b$ encrypted under $pk_B$. $B$ decrypts this message. When $B$ sees the nonce $b$, he is convinced that $A$ replied, since only $A$ could decrypt the second message and obtain $b$. The presence of $pk_A$ in the first message and $pk_B$ in the second message makes explicit that these messages are for sessions between $A$ and $B$, and so avoids man-in-the-middle attacks, such as the well-known attack found by Lowe [54]. This protocol can be represented in our calculus by the process $P$, explained below:

$$\begin{array}{l} P_A(sk_A, pk_A, pk_B) = !c(x\_pk_B).(\nu a)\mathtt{event}(e_1(pk_A, x\_pk_B, a)). \\ \quad (\nu r_1)\overline{c}\langle pencrypt_p((a, pk_A), x\_pk_B, r_1)\rangle. \\ \quad c(m).let\ (= a, x\_b, = x\_pk_B) = pdecrypt_p(m, sk_A)\ in \\ \quad \mathtt{event}(e_3(pk_A, x\_pk_B, a, x\_b)).(\nu r_3)\overline{c}\langle pencrypt_p(x\_b, x\_pk_B, r_3)\rangle \\ \quad if\ x\_pk_B = pk_B\ then \\ \quad \mathtt{event}(e_A(pk_A, x\_pk_B, a, x\_b)).\overline{c}\langle sencrypt(sAa, a)\rangle.\overline{c}\langle sencrypt(sAb, x\_b)\rangle \\ P_B(sk_B, pk_B, pk_A) = !c(m').let\ (x\_a, x\_pk_A) = pdecrypt_p(m', sk_B)\ in\ (\nu b) \\ \quad \mathtt{event}(e_2(x\_pk_A, pk_B, x\_a, b)).(\nu r_2)\overline{c}\langle pencrypt_p((x\_a, b, pk_B), x\_pk_A, r_2)\rangle. \\ \quad c(m'').let\ (= b) = pdecrypt_p(m'', sk_B)\ in \\ \quad if\ x\_pk_A = pk_A\ then \\ \quad \mathtt{event}(e_B(x\_pk_A, pk_B, x\_a, b)).\overline{c}\langle sencrypt(sBa, x\_a)\rangle.\overline{c}\langle sencrypt(sBb, b)\rangle \\ P = (\nu sk_A)(\nu sk_B)let\ pk_A = pk(sk_A)\ in\ let\ pk_B = pk(sk_B)\ in \\ \quad \overline{c}\langle pk_A\rangle\overline{c}\langle pk_B\rangle.(P_A(sk_A, pk_A, pk_B) \mid P_B(sk_B, pk_B, pk_A)) \end{array}$$

The channel $c$ is public: the adversary can send and listen on it. We use a single public channel and not two or more channels because the adversary could take a message from one channel and relay it on another channel, thus removing any difference between the channels. The process $P$ begins with the creation of the secret and public keys of $A$ and $B$. The public keys are output on channel $c$ to model that the adversary has them

in its initial knowledge. Then the protocol itself starts: $P_A$ represents $A$, $P_B$ represents $B$. Both principals can run an unbounded number of sessions, so $P_A$ and $P_B$ start with replications.

We consider that $A$ and $B$ are both willing to talk to any principal. So, to determine to whom $A$ will talk, we consider that $A$ first inputs a message containing the public key $x\_pk_B$ of its interlocutor. (This interlocutor is therefore chosen by the adversary.) Then $A$ starts a protocol run by choosing a nonce $a$, and executing the event $e_1(pk_A, x\_pk_B, a)$. Intuitively, this event records that $A$ sent Message 1 of the protocol, for a run with the participant of public key $x\_pk_B$, using the nonce $a$. Event $e_1$ is placed before the actual output of Message 1; this is necessary for the desired correspondences to hold: if event $e_1$ followed the output of Message 1, one would not be able to prove that event $e_1$ must have been executed, even though Message 1 must have been sent, because Message 1 could be sent without executing event $e_1$. The situation is similar for events $e_2$ and $e_3$ below. Then $A$ sends the first message of the protocol $pencrypt_p((a, pk_A), x\_pk_B, r_1)$, where $r_1$ are fresh coins, used to model that public-key encryption is probabilistic. $A$ waits for the second message and decrypts it using her secret key $sk_A$. If decryption succeeds, $A$ checks that the message has the right form using the pattern-matching construct $let\ (= a, x_b, = x\_pk_B) = pdecrypt_p(m, sk_A)\ in\ \ldots$ This construct is syntactic sugar for $let\ y = pdecrypt_p(m, sk_A)\ in\ let\ x_1 = 1th_3(y)\ in\ let\ x_b = 2th_3(y)\ in\ let\ x_3 = 3th_3(y)\ in\ if\ x_1 = a\ then\ if\ x_3 = x\_pk_B\ then\ \ldots$ Then $A$ executes the event $e_3(pk_A, x\_pk_B, a, x\_b)$, to record that she has received Message 2 and sent Message 3 of the protocol, in a session with the participant of public key $x\_pk_B$, and nonces $a$ and $x\_b$. Finally, she sends the last message of the protocol $pencrypt_p(x\_b, x\_pk_B, r_3)$. After sending this message, $A$ executes some actions needed only for specifying properties of the protocol. When $x\_pk_B = pk_B$, that is, when the session is between $A$ and $B$, $A$ executes the event $e_A(pk_A, x\_pk_B, a, x\_b)$, to record that $A$ ended a session of the protocol, with the participant of public key $x\_pk_B$ and nonces $a$ and $x\_b$. $A$ also outputs the secret name $sAa$ encrypted under the nonce $a$ and the secret name $sAb$ encrypted under the nonce $x\_b$. These outputs are helpful in order to formalize the secrecy of the nonces. Our tool can prove the secrecy of free names, but not the secrecy of bound names (such as $a$) or of variables (such as $x\_b$). In order to overcome this limitation, we publish the encryption of a free name $sAa$ under $a$; then $sAa$ is secret if and only if the nonce $a$ chosen by $A$ is secret. Similarly, $sAb$ is secret if and only if the nonce $x\_b$ received by $A$ is secret.

The process $P_B$ proceeds similarly: it executes the protocol, with the additional event $e_2(x\_pk_A, pk_B, x\_a, b)$ to record that Message 1 has been received and Message 2 has been sent by $B$, in a session with the participant of public key $x\_pk_A$ and nonces $x\_a$ and $b$. After finishing the protocol itself, when $x\_pk_A = pk_A$, that is, when the session is between $A$ and $B$, $P_B$ executes the event $e_B(x\_pk_A, pk_B, x\_a, b)$, to record that $B$ finished the protocol, and outputs $sBa$ encrypted under $x\_a$ and $sBb$ encrypted under $b$, to model the secrecy of $x\_a$ and $b$ respectively.

The events will be used in order to formalize authentication. For example, we formalize that, if $A$ ends a session of the protocol, then $B$ has started a session of the protocol with the same nonces by requiring that, if $e_A(x_1, x_2, x_3, x_4)$ has been

executed, then $e_2(x_1, x_2, x_3, x_4)$ has been executed.[2]

# 3 Definition of Correspondences

In this section, we formally define the correspondences that we verify. We prove correspondences of the form "if an event $e$ has been executed, then events $e_{11}$, ..., $e_{1l_1}$ have been executed, or ..., or $e_{m1}$, ..., $e_{ml_m}$ have been executed". These events may include arguments, which allows one to relate the values of variables at the various events. Furthermore, we can replace the event $e$ with the fact that the adversary knows some term (which allows us to prove secrecy properties), or that a certain message has been sent on a certain channel. We can prove that each execution of $e$ corresponds to a distinct execution of some events $e_{jk}$ (injective correspondences, defined in Section 3.2), and we can prove that the events $e_{jk}$ have been executed in a certain order (general correspondences, defined in Section 3.3).

We assume that the protocol is executed in the presence of an adversary that can listen to all messages, compute, and send all messages it has, following the so-called Dolev-Yao model [40]. Thus, an adversary can be represented by any process that has a set of public names $Init$ in its initial knowledge and that does not contain events. (Although the initial knowledge of the adversary contains only names in $Init$, one can give any terms to the adversary by sending them on a channel in $Init$.)

**Definition 1** Let $Init$ be a finite set of names. The closed process $Q$ is an *$Init$-adversary* if and only if $fn(Q) \subseteq Init$ and $Q$ does not contain events.

## 3.1 Non-injective Correspondences

Next, we define when a trace satisfies an atom $\alpha$, generated by the following grammar:

| $\alpha ::=$ | atom |
|---|---|
| $\mathrm{attacker}(M)$ | attacker knowledge |
| $\mathrm{message}(M, M')$ | message on a channel |
| $\mathrm{event}(M)$ | event |

Intuitively, a trace satisfies $\mathrm{attacker}(M)$ when the attacker has $M$, or equivalently, when $M$ has been sent on a public channel in $Init$. It satisfies $\mathrm{message}(M, M')$ when the message $M'$ has been sent on channel $M$. Finally, it satisfies $\mathrm{event}(M)$ when the event $\mathtt{event}(M)$ has been executed.

**Definition 2** We say that a trace $\mathcal{T} = E_0, \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ satisfies $\mathrm{attacker}(M)$ 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 $c \in Init$.

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

[2] For this purpose, the event $e_A$ must not be executed when $A$ thinks she talks to the adversary. Indeed, in this case, it is correct that no event has been executed by the interlocutor of $A$, since the adversary never executes events.

We say that a trace $\mathcal{T} = E_0, \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ satisfies $\mathrm{event}(M)$ if and only if $\mathcal{T}$ contains a reduction $E, \mathcal{P} \cup \{\, \mathtt{event}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\}$ for some $E$, $\mathcal{P}$, $P$.

The correspondence $\alpha \Rightarrow \bigvee_{j=1}^{m} \left(\alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk})\right)$, formally defined below, means intuitively that, if an instance of $\alpha$ is satisfied, then for some $j \in \{1, \ldots, m\}$, the considered instance of $\alpha$ is an instance of $\alpha_j$ and a corresponding instance of each of the events $\mathtt{event}(M_{j1})$, ..., $\mathtt{event}(M_{jl_j})$ has been executed.[3]

**Definition 3** The closed process $P_0$ *satisfies the correspondence*

$$\alpha \Rightarrow \bigvee_{j=1}^{m} \left( \alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk}) \right)$$

against $Init$-adversaries if and only if, for any $Init$-adversary $Q$, for any $E_0$ containing $fn(P_0) \cup Init \cup fn(\alpha) \cup \bigcup_j fn(\alpha_j) \cup \bigcup_{j,k} fn(M_{jk})$, for any substitution $\sigma$, for any trace $\mathcal{T} = E_0, \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, if $\mathcal{T}$ satisfies $\sigma\alpha$, then there exist $\sigma'$ and $j \in \{1, \ldots, m\}$ such that $\sigma'\alpha_j = \sigma\alpha$ and, for all $k \in \{1, \ldots, l_j\}$, $\mathcal{T}$ satisfies $\mathrm{event}(\sigma' M_{jk})$ as well.

This definition is very general; we detail some interesting particular cases below. When $m = 0$, the disjunction $\bigvee_{j=1}^{m} \ldots$ is denoted by $\mathrm{false}$. When $\alpha = \alpha_j$ for all $j$, we abbreviate the correspondence by $\alpha \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk})$. This correspondence means that, if an instance of $\alpha$ is satisfied, then for some $j \leq m$, a corresponding instance of $\mathtt{event}(M_{j1})$, ..., $\mathtt{event}(M_{jl_j})$ has been executed. The variables in $\alpha$ are universally quantified (because, in Definition 3, $\sigma$ is universally quantified). The variables in $M_{jk}$ that do not occur in $\alpha$ are existentially quantified (because $\sigma'$ is existentially quantified).

**Example 1** In the process of Section 2.3, the correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_1(x_1, x_2, x_3)) \wedge \mathrm{event}(e_2(x_1, x_2, x_3, x_4)) \wedge \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ means that, if the event $e_B(x_1, x_2, x_3, x_4)$ has been executed, then the events $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$ have been executed, with the same value of the arguments $x_1, x_2, x_3, x_4$.

The correspondence

$$\begin{aligned}
&\mathrm{event}(R_received(msg(x, z))) \Rightarrow \\
&\quad (\mathrm{event}(R_received(msg(x, (z', Auth)))) \rightsquigarrow \\
&\qquad \mathrm{event}(S_has(k, msg(x, (z', Auth))))\wedge \\
&\qquad \mathrm{event}(TTP_send(sign((sencrypt(msg(x, (z', Auth)), k), x), sk_{TTP})))) \\
&\vee (\mathrm{event}(R_received(msg(x, (z', NoAuth)))) \rightsquigarrow \\
&\qquad \mathrm{event}(S_has(k, msg(x, (z', NoAuth))))\wedge \\
&\qquad \mathrm{event}(TTP_send(sign(sencrypt(msg(x, (z', NoAuth)), k), sk_{TTP}))))
\end{aligned}$$

[3] The implementation in ProVerif uses a slightly different notation: $\alpha_j$ is omitted, but additionnally equality tests are allowed on the right-hand side of $\rightsquigarrow$, so that one can check that $\alpha$ is actually an instance of $\alpha_j$.

means that, if the event $R\_received(msg(x,z))$ has been executed, then two cases can happen: either $z = (z', Auth)$ or $z = (z', NoAuth)$ for some $z'$. In both cases, the events $TTP\_send(certificate)$ and $S\_has(k, msg(x,z))$ have been executed for some $k$, but with a different value of $certificate$: $certificate = sign((S2TTP, x), sk_{TTP})$ when $z = (z', Auth)$, and $certificate = sign(S2TTP, sk_{TTP})$ when $z = (z', NoAuth)$, with $S2TTP = sencrypt(msg(x,z), k)$. A similar correspondence was used in our study of a certified email protocol, in collaboration with Martín Abadi [2, Section 5, Proposition 4]. We refer to that paper for additional details.

The following definitions are particular cases of Definition 3.

**Definition 4** The closed process $P$ *preserves the secrecy of all instances of* $M$ from $Init$ if and only if it satisfies the correspondence $\mathrm{attacker}(M) \rightsquigarrow \mathrm{false}$ against $Init$-adversaries.

When $M$ is a free name, this definition is equivalent to that of [1].

**Example 2** The process $P$ of Section 2.3 preserves the secrecy of $sAa$ when the correspondence $\mathrm{attacker}(sAa) \rightsquigarrow \mathrm{false}$ is satisfied. In this case, intuitively, $P$ preserves the secrecy of the nonce $a$ that $A$ chooses. The situation is similar for $sAb$, $sBa$, and $sBb$.

**Definition 5** *Non-injective agreement* is a correspondence of the form $\mathrm{event}(e(x_1, \ldots, x_n)) \rightsquigarrow \mathrm{event}(e'(x_1, \ldots, x_n))$.

Intuitively, the correspondence $\mathrm{event}(e(x_1, \ldots, x_n)) \rightsquigarrow \mathrm{event}(e'(x_1, \ldots, x_n))$ means that, if an event $e(M_1, \ldots, M_n)$ is executed, then the event $e'(M_1, \ldots, M_n)$ has also been executed. This definition can be used to represent Lowe's notion of non-injective agreement [55].

**Example 3** In the example of Section 2.3, the correspondence $\mathrm{event}(e_A(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_2(x_1, x_2, x_3, x_4))$ means that, if $A$ executes an event $e_A(x_1, x_2, x_3, x_4)$, then $B$ has executed the event $e_2(x_1, x_2, x_3, x_4)$. So, if $A$ terminates the protocol thinking she talks to $B$, then $B$ is actually involved in the protocol. Moreover, the agreement on the parameter of the events, $pk_A = x\_pk_A$, $x\_pk_B = pk_B$, $a = x\_a$, and $x\_b = b$ implies that $B$ actually thinks he talks to $A$, and that $A$ and $B$ agree on the values of the nonces.

The correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ is similar, after swapping the roles of $A$ and $B$.

## 3.2 Injective Correspondences

**Definition 6** We say that the event $\mathtt{event}(M)$ is executed at step $\tau$ in a trace $\mathcal{T} = E_0, \mathcal{P}_0 \rightarrow^* E', \mathcal{P}'$ if and only if the $\tau$-th reduction of $\mathcal{T}$ is of the form $E, \mathcal{P} \cup \{\, \mathtt{event}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\}$ for some $E$, $\mathcal{P}$, $P$.

Intuitively, an injective correspondence $\mathrm{event}(M) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(M')$ requires that each event $\mathtt{event}(\sigma M)$ is enabled by distinct events $\mathtt{event}(\sigma M')$, while a non-injective correspondence $\mathrm{event}(M) \rightsquigarrow \mathrm{event}(M')$ allows several events $\mathtt{event}(\sigma M)$ to be enabled by the same event $\mathtt{event}(\sigma M')$. We denote by $[\mathrm{inj}]$ an optional inj marker: it can be either inj or nothing. When $[\mathrm{inj}] = \mathrm{inj}$, an injective correspondence is required. When $[\mathrm{inj}]$ is nothing, the correspondence does not need to be injective.

**Definition 7** The closed process $P_0$ *satisfies the correspondence*

$$\mathrm{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(N_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} \mathrm{event}(M_{jk}) \right)$$

against $Init$-adversaries if and only if, for any $Init$-adversary $Q$, for any $E_0$ containing $fn(P_0) \cup Init \cup fn(M) \cup \bigcup_j fn(N_j) \cup \bigcup_{j,k} fn(M_{jk})$, for any trace $\mathcal{T} = E_0, \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, there exist functions $\phi_{jk}$ from a subset of steps in $\mathcal{T}$ to steps in $\mathcal{T}$ such that

- For all $\tau$, if the event $\mathtt{event}(\sigma M)$ is executed at step $\tau$ in $\mathcal{T}$ for some $\sigma$, then there exist $\sigma'$ and $j$ such that $\sigma' N_j = \sigma M$ and, for all $k \in \{1, \ldots, l_j\}$, $\phi_{jk}(\tau)$ is defined and $\mathtt{event}(\sigma' M_{jk})$ is executed at step $\phi_{jk}(\tau)$ in $\mathcal{T}$.
- If $[\mathrm{inj}]_{jk} = \mathrm{inj}$, then $\phi_{jk}$ is injective.

The functions $\phi_{jk}$ map execution steps of events $\mathtt{event}(\sigma M)$ to the execution steps of the events $\mathtt{event}(\sigma' M_{jk})$ that enable $\mathtt{event}(\sigma M)$. When $[\mathrm{inj}]_{jk} = \mathrm{inj}$, the injectivity of $\phi_{jk}$ guarantees that distinct executions of $\mathtt{event}(\sigma M)$ correspond to distinct executions of $\mathtt{event}(\sigma' M_{jk})$. When $M = N_j$ for all $j$, we abbreviate the correspondence by $\mathrm{event}(M) \rightsquigarrow \bigvee_{j=1}^{m} \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} \mathrm{event}(M_{jk})$, as in the non-injective case.

Woo and Lam's correspondence assertions [72] are a particular case of this definition. Indeed, they consider properties of the form: if $\gamma_1$ or ... or $\gamma_k$ have been executed, then $\mu_1$ or ... or $\mu_m$ must have been executed, denoted by $\gamma_1 \mid \ldots \mid \gamma_k \hookrightarrow \mu_1 \mid \ldots \mid \mu_m$. Such a correspondence assertion is formalized in our setting by for all $i \in \{1, \ldots, k\}$, the process satisfies the correspondence $\mathrm{event}(\gamma_i) \rightsquigarrow \bigvee_{j=1}^{m} \mathrm{inj}\ \mathrm{event}(\mu_j)$.

**Remark 1** Correspondences $\alpha \Rightarrow \bigvee_{j=1}^{m} \left( \alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} \mathrm{event}(M_{jk}) \right)$ with $\alpha = \mathrm{attacker}(M)$ and at least one inj marker would always be wrong: the adversary can always repeat the output of $M$ on one of his channels any number of times. With $\alpha = \mathrm{message}(M, M')$ and at least one inj marker, the correspondence may be true only when the adversary cannot execute the corresponding output. For simplicity, we focus on the case $\alpha = \mathrm{event}(M)$ only.

**Definition 8** *Injective agreement* is a correspondence of the form $\mathrm{event}(e(x_1, \ldots, x_n)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e'(x_1, \ldots, x_n))$.

Injective agreement requires that the number of executions of $\mathtt{event}(e(M_1, \ldots, M_n))$ is smaller than the number of executions of $\mathtt{event}(e'(M_1, \ldots, M_n))$: each execution of $\mathtt{event}(e(M_1, \ldots, M_n))$ corresponds to a distinct execution of $\mathtt{event}(e'(M_1, \ldots, M_n))$. This corresponds to Lowe's agreement specification [55].

**Example 4** In the example of Section 2.3, the correspondence $\mathrm{event}(e_A(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_2(x_1, x_2, x_3, x_4))$ means that each execution of $\mathrm{event}(e_A(x_1, x_2, x_3, x_4))$ corresponds to a distinct execution of $\mathrm{event}(e_2(x_1, x_2, x_3, x_4))$. So each completed session of $A$ talking to $B$ corresponds to a distinct session of $B$ talking to $A$, and $A$ and $B$ agree on the values of the nonces.

The correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ is similar, after swapping the roles of $A$ and $B$.

## 3.3 General Correspondences

Correspondences also give information on the order in which events are executed. Indeed, if we have the correspondence

$$\mathrm{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(N_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} \mathrm{event}(M_{jk}) \right)$$

then the events $\mathrm{event}(M_{jk})$ for $k \leq l_j$ have been executed before $\mathrm{event}(N_j)$. Formally, in the definition of injective correspondences, we can define $\phi_{jk}$ such that $\phi_{jk}(\tau) \leq \tau$ when $\phi_{jk}$ is defined. (The inequality $\tau' \leq \tau$ means that $\tau'$ occurs before $\tau$ in the trace.) Indeed, otherwise, by considering the prefix of the trace that stops just after $\tau$, we would contradict the correspondence. In this section, we exploit this point to define more general properties involving the ordering of events.

Let us first consider some examples. Using the process of Section 2.3, we will denote by

$$\begin{aligned} \mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj}\ \mathrm{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \\ (\mathrm{inj}\ \mathrm{event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_1(x_1, x_2, x_3)))) \end{aligned} \quad (1)$$

the correspondence that means that each execution of the event $e_B(x_1, x_2, x_3, x_4)$ corresponds to distinct executions of the events $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$ in this order: each execution of $e_B(x_1, x_2, x_3, x_4)$ is preceded by a distinct execution of $e_3(x_1, x_2, x_3, x_4)$, which is itself preceded by a distinct execution of $e_2(x_1, x_2, x_3, x_4)$, which is itself preceded by a distinct execution of $e_1(x_1, x_2, x_3)$. This correspondence shows that, when $B$ terminates the protocol talking with $A$, $A$ and $B$ have exchanged all messages of the protocol in the expected order. This correspondence is not equivalent to the conjunction of the correspondences $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$, $\mathrm{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_2(x_1, x_2, x_3, x_4))$, and $\mathrm{event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_1(x_1, x_2, x_3))$, because (1) may be true even when, in order to prove that $e_2$ is executed, we need to know that $e_B$ has been executed, and not only that $e_3$ has been executed and, similarly, in order to prove that $e_1$ has been executed, we need to know that $e_B$ has been executed, and not only that $e_2$ has been executed. Using general correspondences such as (1) is therefore strictly more expressive than using injective correspondences. A correspondence similar to (1) has been used in our study of the Just Fast Keying protocol, one of the proposed replacements for IKE in IPSec, in collaboration with Martín Abadi and Cédric Fournet [3, Appendix B.5].

As a more generic example, the correspondence $\mathrm{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left(\mathrm{event}(M_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \left([\mathrm{inj}]_{jk}\mathrm{event}(M_{jk}) \rightsquigarrow \bigvee_{j'=1}^{m_{jk}} \bigwedge_{k'=1}^{l_{jkj'}} [\mathrm{inj}]_{jkj'k'}\mathrm{event}(M_{jkj'k'})\right)\right)$ means that, if an instance of $\mathtt{event}(M)$ has been executed, then there exists $j$ such that this instance of $\mathtt{event}(M)$ is an instance of $\mathtt{event}(M_j)$ and for all $k$, a corresponding instance of $\mathtt{event}(M_{jk})$ has been executed before $\mathtt{event}(M_j)$, and there exists $j'_k$ such that for all $k'$ a corresponding instance of $\mathtt{event}(M_{jkj'_kk'})$ has been executed before $\mathtt{event}(M_{jk})$.

Let us now consider the general definition. We denote by $\overline{k}$ a sequence of indices $k$. The empty sequence is denoted $\epsilon$. When $\overline{j} = j_1 \dots j_n$ and $\overline{k} = k_1 \dots k_n$ are sequences of the same length, we denote by $\overline{jk}$ the sequence obtained by taking alternatively one index in each sequence $\overline{j}$ and $\overline{k}$: $\overline{jk} = j_1k_1 \dots j_nk_n$. We sometimes use $\overline{jk}$ as an identifier that denotes a sequence obtained in this way; for instance, "for all $\overline{jk}$, $\phi_{\overline{jk}}$ is injective" abbreviates "for all $\overline{j}$ and $\overline{k}$ of the same length, $\phi_{\overline{jk}}$ is injective". We only consider sequences $\overline{jk}$ that occur in the correspondence. For instance, for the correspondence $\mathrm{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left(\mathrm{event}(M_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \left([\mathrm{inj}]_{jk}\mathrm{event}(M_{jk}) \rightsquigarrow \bigvee_{j'=1}^{m_{jk}} \bigwedge_{k'=1}^{l_{jkj'}} [\mathrm{inj}]_{jkj'k'}\mathrm{event}(M_{jkj'k'})\right)\right)$, we consider the sequences $\overline{jk} = \epsilon$, $\overline{jk} = jk$, and $\overline{jk} = jkj'k'$ where $1 \leq j \leq m$, $1 \leq k \leq l_j$, $1 \leq j' \leq m_{jk}$, and $1 \leq k' \leq l_{jkj'}$.

Given a family of indices $J = (j_{\overline{k}})_{\overline{k}}$ indexed by sequences of indices $\overline{k}$, we define $\mathrm{makejk}(\overline{k}, J)$ by $\mathrm{makejk}(\epsilon, J) = \epsilon$ and $\mathrm{makejk}(\overline{k}k, J) = \mathrm{makejk}(\overline{k}, J)j_{\overline{k}}k$. Less formally, if $\overline{k} = k_1k_2k_3 \dots$, we have $\mathrm{makejk}(\overline{k}, J) = j_\epsilon k_1 j_{k_1} k_2 j_{k_1k_2} k_3 \dots$ Intuitively, the correspondence contains disjunctions over indices $j$ and conjunctions over indices $k$, so we would like to express quantifications of the form $\exists j_\epsilon \forall k_1 \exists j_{k_1} \forall k_2 \exists j_{k_1k_2} \forall k_3 \dots$ on the sequence $j_\epsilon k_1 j_{k_1} k_2 j_{k_1k_2} k_3 \dots$. The notation $\mathrm{makejk}(\overline{k}, J)$ allows us to replace such a quantification with the quantification $\exists J \forall \overline{k}$ on the sequence $\mathrm{makejk}(\overline{k}, J)$.

**Definition 9** The closed process $P_0$ *satisfies the correspondence*

$$\mathrm{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(M_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} q_{jk} \right)$$

where

$$q_{\overline{jk}} = \mathrm{event}(M_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\mathrm{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

against $Init$-adversaries if and only if, for any $Init$-adversary $Q$, for any $E_0$ containing $fn(P_0) \cup Init \cup fn(M) \cup \bigcup_j fn(M_j) \cup \bigcup_{\overline{jk}} fn(M_{\overline{jk}})$, for any trace $\mathcal{T} = E_0, \{P_0, Q\} \rightarrow^* E', \mathcal{P}'$, there exists a function $\phi_{\overline{jk}}$ for each non-empty $\overline{jk}$, such that for all non-empty $\overline{jk}$, $\phi_{\overline{jk}}$ maps a subset of steps of $\mathcal{T}$ to steps of $\mathcal{T}$ and

- For all $\tau$, if the event $\mathtt{event}(\sigma M)$ is executed at step $\tau$ in $\mathcal{T}$ for some $\sigma$, then there exist $\sigma'$ and $J = (j_{\overline{k}})_{\overline{k}}$ such that $\sigma' M_{j_\epsilon} = \sigma M$ and, for all non-empty $\overline{k}$, $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$ is defined and $\mathtt{event}(\sigma' M_{\mathrm{makejk}(\overline{k},J)})$ is executed at step $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$ in $\mathcal{T}$.

- For all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, then $\phi_{\overline{jk}}$ is injective.
- For all non-empty $\overline{jk}$, for all $j$ and $k$, if $\phi_{\overline{jk}jk}(\tau)$ is defined, then $\phi_{\overline{jk}}(\tau)$ is defined and $\phi_{\overline{jk}jk}(\tau) \leq \phi_{\overline{jk}}(\tau)$. For all $j$ and $k$, if $\phi_{jk}(\tau)$ is defined, then $\phi_{jk}(\tau) \leq \tau$.

We abbreviate by $q_{\overline{jk}} = \mathrm{event}(M_{\overline{jk}})$ the correspondence $q_{\overline{jk}} = \mathrm{event}(M_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\mathrm{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$ when $m_{\overline{jk}} = 1$ and $l_{\overline{jk}1} = 0$, that is, the disjunction $\bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\mathrm{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$ is true. Injective correspondences are then a particular case of general correspondences.

The function $\phi_{\overline{jk}}$ maps the execution steps of instances of $\mathtt{event}(M)$ to the execution steps of the corresponding instances of $\mathtt{event}(M_{\overline{jk}})$. The first item of Definition 9 guarantees that the required events have been executed. The second item means that, when the $\mathrm{inj}$ marker is present, the correspondence is injective. Finally, the third item guarantees that the events have been executed in the expected order.

**Example 5** Let us consider again the correspondence (1). Using the notations of Definition 9, this correspondence is written $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ q_{11}$ (or $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \Rightarrow \mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ q_{11}$), where $q_{11} = \mathrm{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ q_{1111}$, $q_{1111} = \mathrm{event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ q_{111111}$, and $q_{111111} = \mathrm{event}(e_1(x_1, x_2, x_3))$. By Definition 9, this correspondence means that there exist functions $\phi_{11}$, $\phi_{1111}$, and $\phi_{111111}$ such that:

- For all $\tau$, if the event $\mathtt{event}(\sigma e_B(x_1, x_2, x_3, x_4))$ is executed at step $\tau$ for some $\sigma$, then $\phi_{11}(\tau)$, $\phi_{1111}(\tau)$, and $\phi_{111111}(\tau)$ are defined, and $\mathtt{event}(\sigma e_3(x_1, x_2, x_3, x_4))$ is executed at step $\phi_{11}(\tau)$, $\mathtt{event}(\sigma e_2(x_1, x_2, x_3, x_4))$ is executed at step $\phi_{1111}(\tau)$, and $\mathtt{event}(\sigma e_1(x_1, x_2, x_3))$ is executed at step $\phi_{111111}(\tau)$. (Here, $\sigma' = \sigma$ since all variables of the correspondence occur in $\mathrm{event}(e_B(x_1, x_2, x_3, x_4))$. Moreover, $j_{\overline{k}} = 1$ for all $\overline{k}$ and the non-empty sequences $\overline{k}$ are 1, 11, and 111, since all conjunctions and disjunctions have a single element. The sequences $\mathrm{makejk}(\overline{k}, J)$ are then 11, 1111, and 111111.)
- The functions $\phi_{11}$, $\phi_{1111}$, and $\phi_{111111}$ are injective, so distinct executions of $e_B(x_1, x_2, x_3, x_4)$ correspond to distinct executions of $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$.
- When $\phi_{111111}(\tau)$ is defined, $\phi_{111111}(\tau) \leq \phi_{1111}(\tau) \leq \phi_{11}(\tau) \leq \tau$, so the events $e_1(x_1, x_2, x_3)$, $e_2(x_1, x_2, x_3, x_4)$, and $e_3(x_1, x_2, x_3, x_4)$ are executed in this order, before $e_B(x_1, x_2, x_3, x_4)$.

Similarly, general correspondences allow us to express that, if a protocol participant successfully terminates with honest interlocutors, then the expected messages of the protocol have been exchanged between the protocol participants, in the expected order. This notion is the formal counterpart of the notion of matching conversations initially introduced in the computational model by Bellare and Rogaway [11]. This notion of authentication is also used in [35].

We first focus on non-injective correspondences, and postpone the treatment of general correspondences to Section 7.2.

# 4 Automatic Verification: from Secrecy to Correspondences

Let us first summarize our analysis for secrecy. The clauses use two predicates: $\mathrm{attacker}$ and $\mathrm{message}$, where $\mathrm{attacker}(M)$ means that the attacker may have the message $M$ and $\mathrm{message}(M, M')$ means that the message $M'$ may be sent on channel $M$. The clauses relate atoms that use these predicates as follows. A clause $\mathrm{message}(M_1, M_1') \wedge \ldots \wedge \mathrm{message}(M_n, M_n') \Rightarrow \mathrm{message}(M, M')$ is generated when the process outputs $M'$ on channel $M$ after receiving $M_1'$, ..., $M_n'$ on channels $M_1$, ..., $M_n$ respectively. A clause $\mathrm{attacker}(M_1) \wedge \ldots \wedge \mathrm{attacker}(M_n) \Rightarrow \mathrm{attacker}(M)$ is generated when the attacker can compute $M$ from $M_1$, ..., $M_n$. The clause $\mathrm{message}(x, y) \wedge \mathrm{attacker}(x) \Rightarrow \mathrm{attacker}(y)$ means that the attacker can listen on channel $x$ when he has $x$, and the clause $\mathrm{attacker}(x) \wedge \mathrm{attacker}(y) \Rightarrow \mathrm{message}(x, y)$ means that the attacker can send any message $y$ he has on any channel $x$ he has. When $\mathrm{attacker}(M)$ is derivable from the clauses the attacker *may* have $M$, that is, when $\mathrm{attacker}(M)$ is not derivable from the clauses, we are sure that the attacker cannot have $M$, but the converse is not true, because the Horn clauses can be applied any number of times, which is not true in general for all actions of the process. Similarly, when $\mathrm{message}(M, M')$ is derivable from the clauses, the message $M'$ *may* be sent on channel $M$. Hence our analysis overapproximates the execution of actions.

Let us now consider that we want to prove a correspondence, for instance $\mathrm{event}(e_1(x)) \rightsquigarrow \mathrm{event}(e_2(x))$. In order to prove this correspondence, we can overapproximate the executions of event $e_1$: if we prove the correspondence with this overapproximation, it will also hold in the exact semantics. So we can easily extend our analysis for secrecy with an additional predicate $\mathrm{event}$, such that $\mathrm{event}(M)$ means that $\texttt{event}(M)$ may have been executed. We generate clauses $\mathrm{message}(M_1, M_1') \wedge \ldots \wedge \mathrm{message}(M_n, M_n') \Rightarrow \mathrm{event}(M)$ when the process executes $\texttt{event}(M)$ after receiving $M_1'$, ..., $M_n'$ on channels $M_1$, ..., $M_n$ respectively. However, such an overapproximation cannot be done for the event $e_2$: if we prove the correspondence after overapproximating the execution of $e_2$, we are not really sure that $e_2$ will be executed, so the correspondence may be wrong in the exact semantics. Therefore, we have to use a different method for treating $e_2$.

We use the following idea: we fix the exact set $\mathcal{E}$ of allowed events $e_2(M)$ and, in order to prove $\mathrm{event}(e_1(x)) \rightsquigarrow \mathrm{event}(e_2(x))$, we check that only events $e_1(M)$ for $M$ such that $e_2(M) \in \mathcal{E}$ can be executed. If we prove this property for any value of $\mathcal{E}$, we have proved the desired correspondence. So we introduce a predicate $\mathrm{m\text{-}event}$, such that $\mathrm{m\text{-}event}(e_2(M))$ is true if and only if $e_2(M) \in \mathcal{E}$. We generate clauses $\mathrm{message}(M_1, M_1') \wedge \ldots \wedge \mathrm{message}(M_n, M_n') \wedge \mathrm{m\text{-}event}(e_2(M_0)) \Rightarrow \mathrm{message}(M, M')$ when the process outputs $M'$ on channel $M$ after executing the event $e_2(M_0)$ and receiving $M_1'$, ..., $M_n'$ on channels $M_1$, ..., $M_n$ respectively. In other words, the output of $M'$ on channel $M$ can be executed only when $\mathrm{m\text{-}event}(e_2(M_0))$ is true, that is, $e_2(M_0) \in \mathcal{E}$. (When the output of $M'$ on channel $M$ is under several events, the clause contains several $\mathrm{m\text{-}event}$ atoms in its hypothesis. We also have similar clauses with $\mathrm{event}(e_1(M))$ instead of $\mathrm{message}(M, M')$ when the event $e_1$ is executed after executing $e_2$ and receiving $M_1'$, ..., $M_n'$ on channels $M_1$, ..., $M_n$ re-

spectively.)

For instance, if the events $e_2(M_1)$ and $e_2(M_2)$ are executed in a certain trace of the protocol, we define $\mathcal{E} = \{e_2(M_1), e_2(M_2)\}$, so that $\mathrm{m\text{-}event}(e_2(M_1))$ and $\mathrm{m\text{-}event}(e_2(M_2))$ are true and all other $\mathrm{m\text{-}event}$ facts are false. Then we show that the only events $e_1$ that may be executed are $e_1(M_1)$ and $e_1(M_2)$. We prove a similar result for all values of $\mathcal{E}$, which proves the desired correspondence.

In order to determine whether an atom is derivable from the clauses, we use a resolution-based algorithm. The resolution is performed for an unknown value of $\mathcal{E}$. So, basically, we keep $\mathrm{m\text{-}event}$ atoms without trying to evaluate them (which we cannot do since $\mathcal{E}$ is unknown). In the vocabulary of resolution, we never select $\mathrm{m\text{-}event}$ atoms. (We detail this point in Section 6.1.) Thus the obtained result holds for any value of $\mathcal{E}$, which allows us to prove correspondences. In order to prove the correspondence $\mathrm{event}(e_1(x)) \rightsquigarrow \mathrm{event}(e_2(x))$, we show that $\mathrm{event}(e_1(M))$ is derivable only when $\mathrm{m\text{-}event}(e_2(M))$ holds. We transform the initial set of clauses into a set of clauses that derives the same atoms. If, in the obtained set of clauses, all clauses that conclude $\mathrm{event}(e_1(M))$ contain $\mathrm{m\text{-}event}(e_2(M))$ in their hypotheses, then $\mathrm{event}(e_1(M))$ is derivable only when $\mathrm{m\text{-}event}(e_2(M))$ holds, so the desired correspondence holds.

We still have to solve one problem. For simplicity, we have considered that terms, which represent messages, are directly used in clauses. However, in order to represent nonces in our analysis for secrecy, we use a special encoding of names: a name $a$ created by a restriction $(\nu a)$ is represented by a function $a[M_1, \ldots, M_n]$ of the messages $M_1, \ldots, M_n$ received above the restriction, so that names created after receiving different messages are distinguished in the analysis (which is important for the precision of the analysis). However, this encoding still merges names created by the same restriction after receiving the same messages. For example, in the process $!c(x)(\nu a)$, the names created by $(\nu a)$ are represented by $a[x]$, so several names created for the same value of $x$ are merged. This merging is not acceptable for the verification of correspondences, because when we prove $\mathrm{event}(e_1(x)) \rightsquigarrow \mathrm{event}(e_2(x))$, we must make sure that $x$ contains exactly the same names in $e_1(x)$ and in $e_2(x)$. In order to solve this problem, we label each replication with a *session identifier* $i$, which is an integer that takes a different value for each copy of the process generated by the replication. We add session identifiers as arguments to our encoding of names, which becomes $a[M_1, \ldots, M_n, i_1, \ldots, i_{n'}]$ where $i_1, \ldots, i_{n'}$ are the session identifiers of the replications above the restriction $(\nu a)$. For example, in the process $!c(x)(\nu a)$, the names created by $(\nu a)$ are represented by $a[x, i]$. Each execution of the restriction is then associated with a distinct value of the session identifiers $i_1, \ldots, i_{n'}$, so each name has a distinct encoding. We detail and formalize this encoding in Section 5.1.

# 5 From Processes to Horn Clauses

In this section, we first explain the instrumentation of processes with session identifiers. Next, we explain the translation of processes into Horn clauses.

## 5.1 Instrumented Processes

We consider a closed process $P_0$ representing the protocol we wish to check. We assume that the bound names of $P_0$ have been renamed so that they are pairwise distinct and distinct from names in $Init \cup fn(P_0)$ and in the correspondence to prove. We denote by $Q$ a particular adversary; below, we prove the correspondence properties for any $Q$. Furthermore, we assume that, in the initial configuration $E_0, \{P_0, Q\}$, the names of $E_0$ not in $Init \cup fn(P_0)$ or in the correspondence to prove have been renamed to fresh names, and the bound names of $Q$ have been renamed so that they are pairwise distinct and fresh. (These renamings do not change the satisfied correspondences, since $(\nu a)P$ and the renamed process $(\nu a')P\{a'/a\}$ reduce to the same configuration by (Red Res).) After encoding names, the terms are represented by *patterns* $p$ (or "terms", but we prefer the word "patterns" in order to avoid confusion), which are generated by the following grammar:

| $p ::=$ | patterns |
|---|---|
| $x, y, z, i$ | variable |
| $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$ | name |
| $f(p_1, \ldots, p_n)$ | constructor application |

For each name $a$ in $P_0$ we have a corresponding pattern construct $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$. We treat $a$ as a function symbol, and write $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$ rather than $a(p_1, \ldots, p_n, i_1, \ldots, i_{n'})$ only to distinguish names from constructors. The symbol $a$ in $a[\ldots]$ is called a *name function symbol*. If $a$ is a free name, then its encoding is simply $a[\,]$. If $a$ is bound by a restriction $(\nu a)P$ in $P_0$, then its encoding $a[\ldots]$ takes as argument session identifiers $i_1, \ldots, i_{n'}$, which can be constant session identifiers $\lambda$ or variables $i$ (taken in a set $V_s$ disjoint from the set $V_o$ of ordinary variables). There is one session identifier for each replication above the restriction $(\nu a)$. The pattern $a[\ldots]$ may also take as argument patterns $p_1, \ldots, p_n$ containing the messages received by inputs above the restriction $(\nu a)P$ in the abstract syntax tree of $P_0$ and the result of destructor applications above the restriction $(\nu a)P$. (The precise definition is given below.)

In order to define formally the patterns associated with a name, we use a notion of instrumented processes. The syntax of instrumented processes is defined as follows:

- The replication $!P$ is labeled with a variable $i$ in $V_s$: $!^i P$. The process $!^i P$ represents copies of $P$ for a countable number of values of $i$. The variable $i$ is a session identifier. It indicates which copy of $P$, that is, which session, is executed.

- The restriction $(\nu a)P$ is labeled with a restriction label $\ell$: $(\nu a : \ell)P$, where $\ell$ is either $a[M_1, \ldots, M_n, i_1, \ldots, i_{n'}]$ for restrictions in honest processes or $b_0[a[i_1, \ldots, i_{n'}]]$ for restrictions in the adversary. The symbol $b_0$ is a special name function symbol, distinct from all other such symbols. Using a specific instrumentation for the adversary is helpful so that all names generated by the adversary are encoded by instances of $b_0[x]$. They are therefore easy to generate. This labeling of restrictions is similar to a Church-style typing: $\ell$ can be considered as the type of $a$. (This type is polymorphic since it can contain variables.)

The instrumented processes are then generated by the following grammar:

| | |
|---|---|
| $P, Q ::=$ | instrumented processes |
| $\quad !^i P$ | replication |
| $\quad (\nu a : \ell) P$ | restriction |
| $\quad \ldots$ (as in the standard calculus) | |

For instrumented processes, a semantic configuration $S, E, \mathcal{P}$ consists of a set $S$ of session identifiers that have not yet been used by $\mathcal{P}$, an environment $E$ that is a mapping from names to closed patterns of the form $a[\ldots]$, and a finite multiset of instrumented processes $\mathcal{P}$. The first semantic configuration uses any countable set of session identifiers $S_0$. The domain of $E$ must always contain all free names of processes in $\mathcal{P}$, and the initial environment maps all names $a$ to the pattern $a[\,]$. The semantic rules (Red Repl) and (Red Res) become:

$$S, E, \mathcal{P} \cup \{\, !^i P \,\} \rightarrow S \setminus \{\lambda\}, E, \mathcal{P} \cup \{\, P\{\lambda/i\}, !^i P \,\} \text{ where } \lambda \in S \qquad \text{(Red Repl)}$$

$$S, E, \mathcal{P} \cup \{\, (\nu a : \ell) P \,\} \rightarrow S, E[a' \mapsto E(\ell)\,], \mathcal{P} \cup \{\, P\{a'/a\} \,\} \text{ if } a' \notin dom(E) \qquad \text{(Red Res)}$$

where the mapping $E$ is extended to all terms as a substitution by $E(f(M_1, \ldots, M_n)) = f(E(M_1), \ldots, E(M_n))$ and to restriction labels by $E(a[M_1, \ldots, M_n, i_1, \ldots, i_{n'}]) = a[E(M_1), \ldots, E(M_n), i_1, \ldots, i_{n'}]$ and $E(b_0[a[i_1, \ldots, i_{n'}]]) = b_0[a[i_1, \ldots, i_{n'}]]$, so that it maps terms and restriction labels to patterns. The rule (Red Repl) takes an unused constant session identifier $\lambda$ in $S$, and creates a copy of $P$ with session identifier $\lambda$. The rule (Red Res) creates a fresh name $a'$, substitutes it for $a$ in $P$, and adds to the environment $E$ the mapping of $a'$ to its encoding $E(\ell)$. Other semantic rules $E, \mathcal{P} \rightarrow E, \mathcal{P}'$ simply become $S, E, \mathcal{P} \rightarrow S, E, \mathcal{P}'$.

The instrumented process $P'_0 = \mathrm{instr}(P_0)$ associated with the process $P_0$ is built from $P_0$ as follows:

- We label each replication $!P$ of $P_0$ with a distinct, fresh session identifier $i$, so that it becomes $!^i P$.
- We label each restriction $(\nu a)$ of $P_0$ with $a[t, s]$, so that it becomes $(\nu a : a[t, s])$, where $s$ is the sequence of session identifiers that label replications above $(\nu a)$ in the abstract syntax tree of $P'_0$, in the order from top to bottom; $t$ is the sequence of variables $x$ that store received messages in inputs $M(x)$ above $(\nu a)$ in $P_0$ and results of non-deterministic destructor applications $\mathit{let}\ x = g(\ldots)\ \mathit{in}\ P\ \mathit{else}\ Q$ above $(\nu a)$ in $P_0$. (A destructor is said to be non-deterministic when it may return several different results for the same arguments. Adding the result of destructor applications to $t$ is useful to improve precision, only for non-deterministic destructors. For deterministic destructors, the result of the destructor can be uniquely determined from the other elements of $t$, so the addition is useless. If we add the result of non-deterministic destructors to $t$, we can show that the relative completeness result of [1] still holds in the presence of non-deterministic destructors. This result shows that, for secrecy, the Horn clause approach is at least as precise as a large class of type systems.)

Hence names are represented by functions $a[t, s]$ of the inputs and results of destructor applications in $t$ and the session identifiers in $s$. In each trace of the process, at most one name corresponds to a given $a[t, s]$, since different copies of the restriction have different values of session identifiers in $s$. Therefore, different names are not merged by the verifier.

For the adversary, we use a slightly different instrumentation. We build the instrumented process $Q' = \mathrm{instrAdv}(Q)$ as follows:

- We label each replication $!P$ of $Q$ with a distinct, fresh session identifier $i$, so that it becomes $!^i P$.
- We label each restriction $(\nu a)$ of $Q$ with $b_0[a[s]]$, so that it becomes $(\nu a{:}b_0[a[s]])$, where $s$ is the sequence of session identifiers that label replications above $(\nu a)$ in $Q'$. (Including the session identifiers as arguments of nonces is necessary for soundness, as discussed in Section 4. Including the messages previously received as arguments of nonces is important for precision in the case of honest processes, in order to relate the nonces to these messages. It is however useless for the adversary: since we consider any $Init$-adversary $Q$, we have no definite information on the relation between nonces generated by the adversary and messages previously received by the adversary.)

**Remark 2** By moving restrictions downwards in the syntax tree of the process (until the point at which the fresh name is used), one can add more arguments to the pattern that represents the fresh name, when the restriction is moved under an input, replication, or destructor application. Therefore, this transformation can make our analysis more precise. The tool can perform this transformation automatically.

**Example 6** The instrumentation of the process of Section 2.3 yields:

$$\begin{aligned}
&P'_A(sk_A, pk_A, pk_B) = !^{i_A} c(x_pk_B).(\nu a : a[x_pk_B, i_A]) \dots (\nu r_1 : r_1[x_pk_B, i_A]) \dots \\
&\qquad c(m) \dots (\nu r_3 : r_3[x_pk_B, m, i_A]]) \\
&P'_B(sk_B, pk_B, pk_A) = !^{i_B} c(m') \dots (\nu b : b[m', i_B]) \dots (\nu r_2 : r_2[m', i_B]) \dots \\
&P' = (\nu sk_A : sk_A[\,])(\nu sk_B : sk_B[\,]) \dots (P'_A(sk_A, pk_A, pk_B) \mid P'_B(sk_B, pk_B, pk_A))
\end{aligned}$$

The names created by the restriction $(\nu a)$ will be represented by the pattern $a[x\_pk_B, i_A]$, so we have a different pattern for each copy of the process, indexed by $i_A$, and the pattern also records the public key $x\_pk_B$ of the interlocutor of $A$. Similarly, the names created by the restriction $(\nu b)$ will be represented by the pattern $b[m', i_B]$.

The semantics of instrumented processes allows exactly the same communications and events as the one of standard processes. More precisely, let $\mathcal{P}$ be a multiset of instrumented processes. We define $\mathrm{unInstr}(\mathcal{P})$ as the multiset of processes of $\mathcal{P}$ without the instrumentation. Thus we have:

**Proposition 1** *If $E_0, \{P_0, Q\} \rightarrow^* E_1, \mathcal{P}_1$, then there exist $E'_1$ and $\mathcal{P}'_1$ such that for any $S$, countable set of session identifiers, there exists $S'$ such that $S, \{a \mapsto a[\,] \mid a \in E_0\}$,*

$\{\mathrm{instr}(P_0), \mathrm{instrAdv}(Q)\} \rightarrow^* S', E'_1, \mathcal{P}'_1$, $dom(E'_1) = E_1$, $\mathrm{unInstr}(\mathcal{P}'_1) = \mathcal{P}_1$, *and both traces execute the same events at the same steps and satisfy the same atoms.*

*Conversely, if* $S, \{a \mapsto a[\,] \mid a \in E_0\}, \{\mathrm{instr}(P_0), \mathrm{instrAdv}(Q)\} \rightarrow^* S', E'_1, \mathcal{P}'_1$, *then* $E_0, \{P_0, Q\} \rightarrow^* dom(E'_1), \mathrm{unInstr}(\mathcal{P}'_1)$, *and both traces execute the same events at the same steps and satisfy the same atoms.*

**Proof** This is an easy proof by induction on the length of the traces. The reduction rules applied in both traces are rules with the same name. □

We can define correspondences for instrumented processes. These correspondences and the clauses use *facts* defined by the following grammar:

| $F ::=$ | facts |
|---|---|
| $\mathrm{attacker}(p)$ | attacker knowledge |
| $\mathrm{message}(p, p')$ | message on a channel |
| $\mathrm{m\text{-}event}(p)$ | must-event |
| $\mathrm{event}(p)$ | may-event |

The fact $\mathrm{attacker}(p)$ means that the attacker may have $p$, and the fact $\mathrm{message}(p, p')$ means that the message $p'$ may appear on channel $p$. The fact $\mathrm{m\text{-}event}(p)$ means that $\mathtt{event}(M)$ must have been executed with $M$ corresponding to $p$, and $\mathrm{event}(p)$ that $\mathtt{event}(M)$ may have been executed with $M$ corresponding to $p$. We use the word "fact" to distinguish them from atoms $\mathrm{attacker}(M)$, $\mathrm{message}(M, M')$, and $\mathrm{event}(M)$. The correspondences do not use the fact $\mathrm{m\text{-}event}(p)$, but the clauses use it.

The mapping $E$ of a semantic configuration is extended to atoms by $E(\mathrm{attacker}(M)) = \mathrm{attacker}(E(M))$, $E(\mathrm{message}(M, M')) = \mathrm{message}(E(M), E(M'))$, and $E(\mathrm{event}(M)) = \mathrm{event}(E(M))$, so that it maps atoms to facts. We define that an instrumented trace $\mathcal{T}$ satisfies an atom $\alpha$ by naturally adapting Definition 2. When $F$ is not $\mathrm{m\text{-}event}(p)$, we say that an instrumented trace $\mathcal{T} = S_0, E_0, \mathcal{P}_0 \rightarrow^* S', E', \mathcal{P}'$ satisfies a fact $F$ when there exists an atom $\alpha$ such that $\mathcal{T}$ satisfies $\alpha$ and $E'(\alpha) = F$. We also define that $\mathtt{event}(M)$ is executed at step $\tau$ in the instrumented trace $\mathcal{T}$ by naturally adapting Definition 6. We say that $\mathtt{event}(p)$ is executed at step $\tau$ in the instrumented trace $\mathcal{T} = S_0, E_0, \mathcal{P}_0 \rightarrow^* S', E', \mathcal{P}'$ when there exists a term $M$ such that $\mathtt{event}(M)$ is executed at step $\tau$ in $\mathcal{T}$ and $E'(M) = p$.

**Definition 10** Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}(P_0)$. The instrumented process $P'_0$ satisfies the correspondence

$$F \Rightarrow \bigvee_{j=1}^{m} \left( F_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(p_{jk}) \right)$$

against $Init$-adversaries if and only if, for any $Init$-adversary $Q$, for any trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$, with $Q' = \mathrm{instrAdv}(Q)$, $E_0(a) = a[\,]$ for all $a \in dom(E_0)$, and $fn(P'_0) \cup Init \subseteq dom(E_0)$, if $\mathcal{T}$ satisfies $\sigma F$ for some substitution $\sigma$, then there exist $\sigma'$ and $j \in \{1, \ldots, m\}$ such that $\sigma' F_j = \sigma F$ and for all $k \in \{1, \ldots, l_j\}$, $\mathcal{T}$ satisfies $\mathrm{event}(\sigma' p_{jk})$.

A correspondence for instrumented processes implies a correspondence for standard processes, as shown by the following lemma, proved in [18, Appendix A].

**Lemma 1** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let $M_{jk}$ ($j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, l_j\}$) be terms; let $\alpha$ and $\alpha_j$ ($j \in \{1, \ldots, m\}$) be atoms. Let $p_{jk}, F, F_j$ be the patterns and facts obtained by replacing names $a$ with patterns $a[\,]$ in the terms and atoms $M_{jk}, \alpha, \alpha_j$ respectively. If $P'_0$ satisfies the correspondence*

$$F \Rightarrow \bigvee_{j=1}^{m} \left( F_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(p_{jk}) \right)$$

*against $Init$-adversaries then $P_0$ satisfies the correspondence*

$$\alpha \Rightarrow \bigvee_{j=1}^{m} \left( \alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk}) \right)$$

*against $Init$-adversaries.*

For instrumented processes, we can specify properties referring to bound names of the process, which are represented by patterns. Such a specification is impossible in standard processes, because bound names can be renamed, so they cannot be referenced in terms in correspondences.

## 5.2 Generation of Horn Clauses

Given a closed process $P_0$ and a set of names $Init$, the protocol verifier first instruments $P_0$ to obtain $P'_0 = \mathrm{instr}(P_0)$, then it builds a set of Horn clauses, representing the protocol in parallel with any $Init$-adversary. The clauses are of the form $F_1 \wedge \ldots \wedge F_n \Rightarrow F$, where $F_1, \ldots, F_n, F$ are facts. They comprise clauses for the attacker and clauses for the protocol, defined below. These clauses form the set $\mathcal{R}_{P'_0, Init}$. The predicate $\mathrm{m\text{-}event}$ is defined by a set of closed facts $\mathcal{F}_{\mathrm{me}}$, such that $\mathrm{m\text{-}event}(p)$ is true if and only if $\mathrm{m\text{-}event}(p) \in \mathcal{F}_{\mathrm{me}}$. The facts in $\mathcal{F}_{\mathrm{me}}$ do not belong to $\mathcal{R}_{P'_0, Init}$. The set $\mathcal{F}_{\mathrm{me}}$ is the set of facts that corresponds to the set of allowed events $\mathcal{E}$, mentioned in Section 4.

### 5.2.1 Clauses for the Attacker

The clauses describing the attacker are almost the same as for the verification of secrecy in [1]. The only difference is that, here, the attacker is given an infinite set of fresh names $b_0[x]$, instead of only one fresh name $b_0[\,]$. Indeed, we cannot merge all fresh names created by the attacker, since we have to make sure that different terms are represented by different patterns for the verification of correspondences to be correctly implemented, as seen in Section 4. The abilities of the attacker are then represented by the following clauses:

$$\text{For each } a \in Init, \ \mathrm{attacker}(a[\,]) \qquad \text{(Init)}$$

$$\mathrm{attacker}(b_0[x]) \quad \text{(Rn)}$$

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

For each public destructor $g$,

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

$$\mathrm{message}(x, y) \wedge \mathrm{attacker}(x) \Rightarrow \mathrm{attacker}(y) \quad \text{(Rl)}$$

$$\mathrm{attacker}(x) \wedge \mathrm{attacker}(y) \Rightarrow \mathrm{message}(x, y) \quad \text{(Rs)}$$

The clause (Init) represents the initial knowledge of the attacker. The clause (Rn) means that the attacker can generate an unbounded number of new names. The clauses (Rf) and (Rg) mean that the attacker can apply all operations to all terms it has, (Rf) for constructors, (Rg) for destructors. For (Rg), notice that the rewrite rules in $\mathrm{def}(g)$ do not contain names and that terms without names are also patterns, so the clauses have the required format. Clause (Rl) means that the attacker can listen on all channels it has, and (Rs) that it can send all messages it has on all channels it has.

If $c \in \mathit{Init}$, we can replace all occurrences of $\mathrm{message}(c[\,], M)$ with $\mathrm{attacker}(M)$ in the clauses. Indeed, these facts are equivalent by the clauses (Rl) and (Rs).

### 5.2.2 Clauses for the Protocol

When a function $\rho$ associates a pattern with each name and variable, and $f$ is a constructor, we extend $\rho$ 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 $\rho$ is a function that associates a pattern with each name and variable, and $H$ is a sequence of facts of the form $\mathrm{message}(p, p')$ or $\text{m-event}(p)$. The environment $\rho$ maps each variable and name to its associated pattern representation. The sequence $H$ keeps track of events that have been executed and of messages received by the process, since these may trigger other messages. The empty sequence is denoted by $\emptyset$; the concatenation of a fact $F$ to the sequence $H$ is denoted by $H \wedge F$. The pattern $\rho i$ is always a session identifier variable of $V_s$.

$$[\![0]\!]\rho H = \emptyset$$
$$[\![P \mid Q]\!]\rho H = [\![P]\!]\rho H \cup [\![Q]\!]\rho H$$
$$[\![!^i P]\!]\rho H = [\![P]\!](\rho[i \mapsto i])H$$
$$[\![(\nu a : a[M_1, \ldots, M_n, i_1, \ldots, i_{n'}])P]\!]\rho H =$$
$$\quad [\![P]\!](\rho[a \mapsto a[\rho(M_1), \ldots, \rho(M_n), \rho(i_1), \ldots, \rho(i_{n'})]\,])H$$
$$[\![M(x).P]\!]\rho H = [\![P]\!](\rho[x \mapsto x])(H \wedge \mathrm{message}(\rho(M), x))$$
$$[\![\overline{M}\langle N\rangle.P]\!]\rho H = [\![P]\!]\rho H \cup \{H \Rightarrow \mathrm{message}(\rho(M), \rho(N))\}$$
$$[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho H = \bigcup \{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma H)$$
$$\quad \mid g(p'_1, \ldots, p'_n) \rightarrow p' \text{ is in } \mathrm{def}(g) \text{ and } (\sigma, \sigma') \text{ is a most general pair of}$$
$$\quad \text{substitutions such that } \sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho H$$

$[\![\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q]\!]\rho H = [\![P]\!](\sigma\rho)(\sigma H) \cup [\![Q]\!]\rho H$
where $\sigma$ is the most general unifier of $\rho(M)$ and $\rho(N)$
$[\![\mathtt{event}(M).P]\!]\rho H = [\![P]\!]\rho(H \wedge \text{m-event}(\rho(M))) \cup \{H \Rightarrow \text{event}(\rho(M))\}$

The translation of a process is a set of Horn clauses that express that it may send certain messages or execute certain events. The clauses are similar to those of [1], except in the cases of replication, restriction, and the addition of events.

- The nil process does nothing, so its translation is empty.
- The clauses for the parallel composition of processes $P$ and $Q$ are the union of clauses for $P$ and $Q$.
- The replication only inserts the new session identifier $i$ in the environment $\rho$. It is otherwise ignored, because all Horn clauses are applicable arbitrarily many times.
- For the restriction, we replace the restricted name $a$ in question with the pattern $a[\rho(M_1), \ldots, \rho(M_n), \rho(i_1), \ldots, \rho(i_{n'})]$. By definition of the instrumentation, this pattern contains the previous inputs, results of non-deterministic destructor applications, and session identifiers.
- The sequence $H$ is extended in the translation of an input, with the input in question.
- The translation of an output adds a clause, meaning that the output is triggered when all conditions in $H$ are true.
- The translation of a destructor application is the union of the clauses for the cases where the destructor succeeds (with an appropriate substitution) and where the destructor fails. For simplicity, we assume that the $\mathit{else}$ branch of destructors may always be executed; this is sufficient in most cases, since the $\mathit{else}$ branch is often empty or just sends an error message. We outline a more precise treatment in Section 9.2.
- The conditional $\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$ is in fact equivalent to $\mathit{let}\ x = \mathit{equal}(M, N)\ \mathit{in}\ P\ \mathit{else}\ Q$, where the destructor $\mathit{equal}$ is defined by $\mathit{equal}(x, x) \to x$, so the translation of the conditional is a particular case of the destructor application. We give it explicitly since it is particularly simple.
- The translation of an event adds the hypothesis $\text{m-event}(\rho(M))$ to $H$, meaning that $P$ can be executed only if the event has been executed first. Furthermore, it adds a clause, meaning that the event is triggered when all conditions in $H$ are true.

**Remark 3** Depending on the form of the correspondences we want to prove, we can sometimes simplify the clauses generated for events. Suppose that all arguments of events in the process and in correspondences are of the form $f(M_1, \ldots, M_n)$ for some function symbol $f$.

If, for a certain function symbol $f$, events $\mathrm{event}(f(\ldots))$ occur only before $\rightsquigarrow$ in the desired correspondences, then it is easy to see in the following theorems that hypotheses of the form $\mathrm{m\text{-}event}(f(\ldots))$ in clauses can be removed without changing the result, so the clauses generated by the event $\mathtt{event}(M)$ when $M$ is of the form $f(\ldots)$ can be simplified into:

$$[\![\mathtt{event}(M).P]\!]\rho H = [\![P]\!]\rho H \cup \{H \Rightarrow \mathrm{event}(\rho(M))\}$$

(Intuitively, since the events $\mathrm{event}(f(\ldots))$ occur only before $\rightsquigarrow$ in the desired correspondences, we never prove that an event $\mathtt{event}(f(\ldots))$ has been executed, so the facts $\mathrm{m\text{-}event}(f(\ldots))$ are useless.)

Similarly, if $\mathrm{event}(f(\ldots))$ occurs only after $\rightsquigarrow$ in the desired correspondences, then clauses that conclude a fact of the form $\mathrm{event}(f(\ldots))$ can be removed without changing the result, so the clauses generated by the event $\mathtt{event}(M)$ when $M$ is of the form $f(\ldots)$ can be simplified into:

$$[\![\mathtt{event}(M).P]\!]\rho H = [\![P]\!]\rho(H \wedge \mathrm{m\text{-}event}(\rho(M)))$$

(Intuitively, since the events $\mathrm{event}(f(\ldots))$ occur only after $\rightsquigarrow$ in the desired correspondences, we never prove properties of the form "if $\mathtt{event}(f(\ldots))$ has been executed, then ...", so clauses that conclude $\mathrm{event}(f(\ldots))$ are useless.)

This translation of the protocol into Horn clauses introduces approximations. The actions are considered as implicitly replicated, since the clauses can be applied any number of times. This approximation implies that the tool fails to prove protocols that first need to keep some value secret and later reveal it. For instance, consider the process $(\nu d)(\overline{d}\langle s\rangle.\overline{c}\langle d\rangle \mid d(x))$. This process preserves the secrecy of $s$, because $s$ is output on the private channel $d$ and received by the input on $d$, before the adversary gets to know $d$ by the output of $d$ on the public channel $c$. However, the Horn clause method cannot prove this property, because it treats this process like a variant with additional replications $(\nu d)(!\overline{d}\langle s\rangle.\overline{c}\langle d\rangle \mid !d(x))$, which does not preserve the secrecy of $s$. Similarly, the process $(\nu d)(\overline{d}\langle M\rangle \mid d(x).d(x).\mathtt{event}(e_1))$ never executes the event $e_1$, but the Horn clause method cannot prove this property because it treats this process like $(\nu d)(!\overline{d}\langle M\rangle \mid d(x).d(x).\mathtt{event}(e_1))$, which may execute $e_1$. The only exception to this implicit replication of processes is the creation of new names: since session identifiers appear in patterns, the created name is precisely related to the session that creates it, so name creation cannot be unduly repeated inside the same session. Due to these approximations, our tool is not complete (it may produce false attacks) but, as we show below, it is sound (the security properties that it proves are always true).

### 5.2.3 Summary and Correctness

Let $\rho = \{a \mapsto a[\,] \mid a \in fn(P'_0)\}$. We define the clauses corresponding to the instrumented process $P'_0$ as:

$$\mathcal{R}_{P'_0,Init} = [\![P'_0]\!]\rho\emptyset \cup \{\mathrm{attacker}(a[\,]) \mid a \in Init\} \cup \{(\mathrm{Rn}), (\mathrm{Rf}), (\mathrm{Rg}), (\mathrm{Rl}), (\mathrm{Rs})\}$$

**Example 7** The clauses for the process $P$ of Section 2.3 are the clauses for the adversary, plus:

$$\text{attacker}(pk(sk_A[\,])) \tag{2}$$

$$\text{attacker}(pk(sk_B[\,])) \tag{3}$$

$$H_1 \Rightarrow \text{attacker}(pencrypt_p((a[x\_pk_B, i_A], pk(sk_A[\,])), x\_pk_B, r_1[x\_pk_B, i_A])) \tag{4}$$

$$H_2 \Rightarrow \text{attacker}(pencrypt_p(x\_b, x\_pk_B, r_3[x\_pk_B, p_2, i_A])) \tag{5}$$

$$H_3 \Rightarrow \text{event}(e_A(pk(sk_A[\,]), pk(sk_B[\,]), a[pk(sk_B[\,]), i_A], x\_b)) \tag{6}$$

$$H_3 \Rightarrow \text{attacker}(sencrypt(sAa[\,], a[pk(sk_B[\,]), i_A])) \tag{7}$$

$$H_3 \Rightarrow \text{attacker}(sencrypt(sAb[\,], x\_b)) \tag{8}$$

where $p_2 = pencrypt_p((a[x\_pk_B, i_A], x\_b, x\_pk_B), pk(sk_A[\,]), x\_r_2)$

$H_1 = \text{attacker}(x\_pk_B) \wedge \text{m-event}(e_1(pk(sk_A[\,]), x\_pk_B, a[x\_pk_B, i_A]))$

$H_2 = H_1 \wedge \text{attacker}(p_2) \wedge \text{m-event}(e_3(pk(sk_A[\,]), x\_pk_B, a[x\_pk_B, i_A], x\_b))$

$H_3 = H_2\{pk(sk_B[\,])/x\_pk_B\}$

$$\begin{array}{l}\text{attacker}(p_1) \wedge \text{m-event}(e_2(x\_pk_A, pk(sk_B[\,]), x\_a, b[p_1, i_B])) \\ \quad \Rightarrow \text{attacker}(pencrypt_p((x_a, b[p_1, i_B], pk(sk_B[\,])), x\_pk_A, r_2[p_1, i_B]))\end{array} \tag{9}$$

where $p_1 = pencrypt_p((x\_a, x\_pk_A), pk(sk_B[\,]), x\_r_1)$

$$H_4 \Rightarrow \text{event}(e_B(pk(sk_A[\,]), pk(sk_B[\,]), x\_a, b[p'_1, i_B])) \tag{10}$$

$$H_4 \Rightarrow \text{attacker}(sencrypt(sBa[\,], x\_a)) \tag{11}$$

$$H_4 \Rightarrow \text{attacker}(sencrypt(sBb[\,], b[p'_1, i_B])) \tag{12}$$

where $p'_1 = pencrypt_p((x\_a, pk(sk_A[\,])), pk(sk_B[\,]), x\_r_1)$

$H_4 = \text{attacker}(p'_1) \wedge \text{m-event}(e_2(pk(sk_A[\,]), pk(sk_B[\,]), x\_a, b[p'_1, i_B])) \wedge$
$\text{attacker}(pencrypt_p(b[p'_1, i_B], pk(sk_B[\,]), x\_r_3))$

Clauses (2) and (3) correspond to the outputs in $P$; they mean that the adversary has the public keys of the participants. Clauses (4) and (5) correspond to the first two outputs in $P_A$. For example, (5) means that, if the attacker has $x\_pk_B$ and the second message of the protocol $p_2$ and the events $e_1(pk(sk_A[\,]), x\_pk_B, a[x\_pk_B, i_A])$ and $e_3(pk(sk_A[\,]), x\_pk_B, a[x\_pk_B, i_A], x\_b)$ are allowed, then the attacker can get $pencrypt_p(x\_b, x\_pk_B, r_3[x\_pk_B, p_2, i_A])$, because $P_A$ sends this message after receiving $x\_pk_B$ and $p_2$ and executing the events $e_1$ and $e_3$. When furthermore $x\_pk_B = pk(sk_B[\,])$, $P_A$ executes event $e_A$ and outputs the encryption of $sAa[\,]$ under $a[x\_pk_B, i_A]$ and the encryption of $sBb[\,]$ under $x\_b$. These event and outputs are taken into account by Clauses (6), (7), and (8) respectively. Similarly, Clauses (9), (11), and (12) correspond to the outputs in $P_B$ and (10) to the event $e_B$. These clauses have been simplified using Remark 3, taking into account that $e_1$, $e_2$, and $e_3$ appear only on the right-hand side of $\rightsquigarrow$, and $e_A$ and $e_B$ only on the left-hand side of $\rightsquigarrow$ in the queries of Examples 1, 2, and 3.

**Theorem 1 (Correctness of the clauses)** *Let $P_0$ be a closed process and $Q$ be an Init-adversary. Let $P'_0 = \text{instr}(P_0)$ and $Q' = \text{instrAdv}(Q)$. Consider a trace $\mathcal{T} =$*

*$S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$, with $fn(P'_0) \cup Init \subseteq dom(E_0)$ and $E_0(a) = a[]$ for all $a \in dom(E_0)$. Assume that, if $\mathcal{T}$ satisfies $\mathrm{event}(p)$, then $\mathrm{m\text{-}event}(p) \in \mathcal{F}_{\mathrm{me}}$. Finally, assume that $\mathcal{T}$ satisfies $F$. Then $F$ is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\mathrm{me}}$.*

This result shows that, if the only executed events are those allowed in $\mathcal{F}_{\mathrm{me}}$ and a fact $F$ is satisfied, then $F$ is derivable from the clauses. It is proved in [18, Appendix B]. Using a technique similar to that of [1], its proof relies on a type system to express the soundness of the clauses on $P'_0$, and on the subject reduction of this type system to show that soundness of the clauses is preserved during all executions of the process.

# 6 Solving Algorithm

We first describe a basic solving algorithm without optimizations. Next, we list the optimizations that we use in our implementation, and we prove the correctness of the algorithm. The termination of the algorithm is discussed in Section 8.

## 6.1 The Basic Algorithm

To apply the previous results, we have to determine whether a fact is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\mathrm{me}}$. This may be undecidable, but in practice there exist algorithms that terminate on numerous examples of protocols. In particular, we can use variants of resolution algorithms, such as the algorithms described in [13, 14, 21, 70]. The algorithm that we describe here is the one of [14], extended with a second phase to determine derivability of any query. It also corresponds to the extension to $\mathrm{m\text{-}event}$ facts of the algorithm of [21].

We first define resolution: when the conclusion of a clause $R$ unifies with an hypothesis $F_0$ of a clause $R'$, we can infer a new clause $R \circ_{F_0} R'$, that corresponds to applying $R$ and $R'$ one after the other. Formally, this is defined as follows:

**Definition 11** Let $R = H \Rightarrow C$ and $R' = H' \Rightarrow C'$ be two clauses. Assume that there exists $F_0 \in H'$ such that $C$ and $F_0$ are unifiable, and $\sigma$ is the most general unifier of $C$ and $F_0$. In this case, we define $R \circ_{F_0} R' = \sigma(H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma C'$.

An important idea to obtain an efficient solving algorithm is to specify conditions that limit the application of resolution, while keeping completeness. The conditions that we use correspond to resolution with free selection [9, 36, 56]: a selection function chooses selected facts in each clause, and resolution is performed only on selected facts, that is, the clause $R \circ_{F_0} R'$ is generated only when the conclusion is selected in $R$ and $F_0$ is selected in $R'$.

**Definition 12** We denote by $\mathsf{sel}$ a selection function, that is, a function from clauses to sets of facts, such that $\mathsf{sel}(H \Rightarrow C) \subseteq H$. If $F \in \mathsf{sel}(R)$, we say that $F$ is selected in $R$. If $\mathsf{sel}(R) = \emptyset$, we say that no hypothesis is selected in $R$, or that the conclusion of the clause is selected.

The choice of the selection function can change dramatically the speed of the algorithm. Since the algorithm combines clauses by resolution only when the facts unified in the resolution are selected, we will choose the selection function to reduce the number of possible unifications between selected facts. Having several selected facts slows down the algorithm, because it has more choices of resolutions to perform, therefore we will select at most one fact in each clause. In the case of protocols, facts of the form $\mathrm{attacker}(x)$, with $x$ variable, can be unified will all facts of the form $\mathrm{attacker}(p)$. Therefore we should avoid selecting them. The $\mathrm{m\text{-}event}$ facts must never be selected since they are not defined by known clauses.

**Definition 13** We say that a fact $F$ is *unselectable* when $F = \mathrm{attacker}(x)$ for some variable $x$ or $F = \mathrm{m\text{-}event}(p)$ for some pattern $p$. Otherwise, we say that $F$ is *selectable*.

We require that the selection function never selects unselectable hypotheses and that $\mathsf{sel}(H \Rightarrow \mathrm{attacker}(x)) \neq \emptyset$ when $H$ contains a selectable fact.

A basic selection function for security protocols is then

$$\mathsf{sel}_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if } \forall F \in H,\ F \text{ is unselectable} \\ \{F_0\} & \text{where } F_0 \in H \text{ and } F_0 \text{ is selectable, otherwise} \end{cases}$$

In the implementation, the hypotheses are represented by a list, and the selected fact is the first selectable element of the list of hypotheses.

The solving algorithm works in two phases, summarized in Figure 4. The first phase, saturate, transforms the set of clauses into an equivalent but simpler one. The second phase, derivable, uses a depth-first search to determine whether a fact can be inferred or not from the clauses.

The first phase contains 3 steps.

- The first step inserts in $\mathcal{R}$ the initial clauses representing the protocol and the attacker (clauses that are in $\mathcal{R}_0$), after simplification by $simplify$ (defined below in Section 6.2) and elimination of subsumed clauses by $elim$. We say that $H_1 \Rightarrow C_1$ subsumes $H_2 \Rightarrow C_2$, and we write $(H_1 \Rightarrow C_1) \sqsupseteq (H_2 \Rightarrow C_2)$, when there exists a substitution $\sigma$ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$. ($H_1$ and $H_2$ are multisets, and we use here multiset inclusion.) If $R'$ subsumes $R$, and $R$ and $R'$ are in $\mathcal{R}$, then $R$ is removed by $elim(\mathcal{R})$.

- The second step is a fixpoint iteration that adds clauses created by resolution. The composition of clauses $R$ and $R'$ is added only if no hypothesis is selected in $R$, and the hypothesis $F_0$ of $R'$ that we unify is selected. When a clause is created by resolution, it is added to the set of clauses $\mathcal{R}$ after simplification. Subsumed clauses are eliminated from $\mathcal{R}$.

- At last, the third step returns the set of clauses of $\mathcal{R}$ with no selected hypothesis.

Basically, saturate preserves derivability: $F$ is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\mathrm{me}}$ if and only if it is derivable from $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$. A formal statement of this result is given in Lemma 2 below.

**First phase: saturation**

$\mathsf{saturate}(\mathcal{R}_0) =$

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

**Second phase: backwards depth-first search**

$$\mathsf{deriv}(R, \mathcal{R}, \mathcal{R}_1) = \begin{cases} \emptyset & \text{if } \exists R' \in \mathcal{R}, R' \sqsupseteq R \\ \{R\} & \text{otherwise, if } \mathsf{sel}(R) = \emptyset \\ \bigcup\{\mathsf{deriv}(simplify'(R' \circ_{F_0} R), \{R\} \cup \mathcal{R}, \mathcal{R}_1) \mid R' \in \mathcal{R}_1, \\ \qquad F_0 \in \mathsf{sel}(R) \text{ such that } R' \circ_{F_0} R \text{ is defined}\} & \text{otherwise} \end{cases}$$

$$\mathsf{derivable}(F, \mathcal{R}_1) = \mathsf{deriv}(F \Rightarrow F, \emptyset, \mathcal{R}_1)$$

Figure 4: Solving algorithm

The second phase searches the facts that can be inferred from $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$. This is simply a backward depth-first search. The call $\mathsf{derivable}(F, \mathcal{R}_1)$ returns a set of clauses $R = H \Rightarrow C$ with empty selection, such that $R$ can be obtained by resolution from $\mathcal{R}_1$, $C$ is an instance of $F$, and all instances of $F$ derivable from $\mathcal{R}_1$ can be derived by using as last clause a clause of $\mathsf{derivable}(F, \mathcal{R}_1)$. (Formally, if $F'$ is an instance of $F$ derivable from $\mathcal{R}_1$, then there are a clause $H \Rightarrow C \in \mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution $\sigma$ such that $F' = \sigma C$ and $\sigma H$ is derivable from $\mathcal{R}_1$.)

The search itself is performed by $\mathsf{deriv}(R, \mathcal{R}, \mathcal{R}_1)$. The function deriv starts with $R = F \Rightarrow F$ and transforms the hypothesis of $R$ by using a clause $R'$ of $\mathcal{R}_1$ to derive an element $F_0$ of the hypothesis of $R$. So $R$ is replaced with $R' \circ_{F_0} R$ (third case of the definition of deriv). The fact $F_0$ is chosen using the selection function sel. The obtained clause $R' \circ_{F_0} R$ is then simplified by the function $simplify'$ defined in Section 6.2. (Hence deriv derives the hypothesis of $R$ using a backward depth-first search. At each step, the clause $R$ can be obtained by resolution from clauses of $\mathcal{R}_1$, and $R$ concludes an instance of $F$.) The set $\mathcal{R}$ is the set of clauses that we have already seen during the search. Initially, $\mathcal{R}$ is empty, and the clause $R$ is added to $\mathcal{R}$ in the third case of the definition of deriv.

The transformation of $R$ described above is repeated until one of the following two conditions is satisfied:

- $R$ is subsumed by a clause in $\mathcal{R}$: we are in a cycle; we are looking for instances of facts that we have already looked for (first case of the definition of deriv);
- $\mathsf{sel}(R)$ is empty: we have obtained a suitable clause $R$ and we return it (second case of the definition of deriv).

## 6.2 Simplification Steps

Before adding a clause to the clause base, it is first simplified using the following functions. Some of them are standard, such as the elimination of tautologies and of duplicate hypotheses; others are specific to protocols. The simplification functions take as input a clause or a set of clauses and return a set of clauses.

**Decomposition of Data Constructors** A data constructor is a constructor $f$ of arity $n$ that comes with associated destructors $g_i$ for $i \in \{1, \ldots, n\}$ 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:

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

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

Therefore, $\mathrm{attacker}(f(p_1, \ldots, p_n))$ is derivable if and only if $\forall i \in \{1, \ldots, n\}$, $\mathrm{attacker}(p_i)$ is derivable. So the function $decomp$ transforms clauses as follows. When a fact of the form $\mathrm{attacker}(f(p_1, \ldots, p_n))$ is met, it is replaced with $\mathrm{attacker}(p_1) \wedge \ldots \wedge \mathrm{attacker}(p_n)$. If this replacement is done in the conclusion of a clause $H \Rightarrow \mathrm{attacker}(f(p_1, \ldots, p_n))$, $n$ clauses are created: $H \Rightarrow \mathrm{attacker}(p_i)$ 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 function $decomphyp$ performs this decomposition only in the hypothesis of clauses. The functions $decomp$ and $decomphyp$ leave the clauses (Rf) and (Rg) for data constructors unchanged. (When $\mathrm{attacker}(x)$ cannot be selected, the clauses (Rf) and (Rg) for data constructors are in fact not necessary, because they generate only tautologies during resolution. However, when $\mathrm{attacker}(x)$ can be selected, which cannot be excluded in extensions such as the one presented in Section 9.3, these clauses may become necessary for soundness.)

**Elimination of Tautologies** The function $elimtaut$ removes clauses whose conclusion is already in the hypotheses, since such clauses do not generate new facts.

**Elimination of Duplicate Hypotheses** The function $elimdup$ eliminates duplicate hypotheses of clauses.

**Elimination of Useless $\mathrm{attacker}(x)$ Hypotheses** If a clause $H \Rightarrow C$ contains in its hypotheses $\mathrm{attacker}(x)$, where $x$ is a variable that does not appear elsewhere in the clause, the hypothesis $\mathrm{attacker}(x)$ is removed by the function $elimattx$. Indeed, the attacker always has at least one message, so $\mathrm{attacker}(x)$ is always satisfied.

**Secrecy Assumptions** When the user knows that a fact $F$ will not be derivable, he can tell it to the verifier. (When this fact is of the form $\mathrm{attacker}(p)$, the user tells that $p$ remains secret; that is why we use the name "secrecy assumptions".) Let $\mathcal{F}_{\mathrm{not}}$ be a set of facts, for which the user claims that no instance of these facts is derivable. The

$\mathsf{solve}_{P'_0,Init}(F) =$

1. Let $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_{P'_0,Init})$.
2. For each $F' \in \mathcal{F}_{\mathrm{not}}$, if $\mathsf{derivable}(F', \mathcal{R}_1) \neq \emptyset$, then terminate with error.
3. Return $\mathsf{derivable}(F, \mathcal{R}_1)$.

Figure 5: Summary of the solving algorithm

function $elimnot$ removes all clauses that have an instance of a fact in $\mathcal{F}_{\mathrm{not}}$ in their hypotheses. As shown in Figure 5, at the end of the saturation, the solving algorithm checks that the facts in $\mathcal{F}_{\mathrm{not}}$ are indeed underivable from the obtained clauses. If this condition is satisfied, $\mathsf{solve}_{P'_0,Init}(F)$ returns clauses that conclude instances of $F$. Otherwise, the user has given erroneous information, so an error message is displayed. Even when the user gives erroneous secrecy assumptions, the verifier never wrongly claims that a protocol is secure.

Mentioning such underivable facts prunes the search space, by removing useless clauses. This speeds up the search process. In most cases, the secret keys of the principals cannot be known by the attacker, so examples of underivable facts are $\mathrm{attacker}(sk_A[])$ and $\mathrm{attacker}(sk_B[])$.

**Elimination of Redundant Hypotheses** When a clause is of the form $H \wedge H' \Rightarrow C$, and there exists $\sigma$ such that $\sigma H \subseteq H'$ and $\sigma$ does not change the variables of $H'$ and $C$, then the clause is replaced with $H' \Rightarrow C$ by the function $elimredundanthyp$. These clauses are semantically equivalent: obviously, $H' \Rightarrow C$ subsumes $H \wedge H' \Rightarrow C$; conversely, if a fact can be derived by an instance $\sigma' H' \Rightarrow \sigma' C$ of $H' \Rightarrow C$, then it can also be derived by the instance $\sigma'\sigma H \wedge \sigma' H' \Rightarrow \sigma' C$ of $H \wedge H' \Rightarrow C$, since the elements of $\sigma'\sigma H$ can be derived because they are in $\sigma' H'$.

This replacement is especially useful when $H$ contains $\mathrm{m\text{-}event}$ facts. Otherwise, the elements of $H$ could be selected and transformed by resolution, until they are of the form $\mathrm{attacker}(x)$, in which case they are removed by $elimattx$ if $\sigma x \neq x$ (because $x$ does not occur in $H'$ and $C$ since $\sigma$ does not change the variables of $H'$ and $C$) or by $elimdup$ if $\sigma x = x$ (because $\mathrm{attacker}(x) = \sigma\mathrm{attacker}(x) \in \sigma H \subseteq H'$). In contrast, $\mathrm{m\text{-}event}$ facts remain forever, because they are unselectable. Depending on user settings, this replacement can be applied for all $H$, applied only when $H$ contains a $\mathrm{m\text{-}event}$ fact, or switched off, since testing this property takes time and slows down small examples. On the other hand, on big examples, such as some of those generated by TulaFale [12] for verifying Web services, this technique can yield important speedups.

**Putting All Simplifications Together** The function $simplify$ groups all these simplifications. We define $simplify = elimattx \circ elimtaut \circ elimnot \circ elimredundanthyp \circ elimdup \circ decomp$. In this definition, the simplifications are ordered in such a way that $simplify \circ simplify = simplify$, so it is not necessary to repeat the simplification.

Similarly, $simplify' = elimattx \circ elimnot \circ elimredundanthyp \circ elimdup \circ decomphyp$. In $simplify'$, we use $decomphyp$ instead of $decomp$, because the conclu-

sion of the considered clause is the fact we want to derive, so it must not be modified.

## 6.3 Soundness

The following lemmas show the correctness of saturate and derivable (Figure 4). Proofs can be found in [18, Appendix C]. Intuitively, the correctness of saturate expresses that saturation preserves derivability, provided the secrecy assumptions are satisfied.

**Lemma 2 (Correctness of** saturate**)** *Let $F$ be a closed fact. If, for all $F' \in \mathcal{F}_{\mathrm{not}}$, no instance of $F'$ is derivable from* $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$*, then $F$ is derivable from $\mathcal{R}_0 \cup \mathcal{F}_{\mathrm{me}}$ if and only if $F$ is derivable from* $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$.

This result is proved by transforming a derivation of $F$ from $\mathcal{R}_0 \cup \mathcal{F}_{\mathrm{me}}$ into a derivation of $F$ (or a fact in $\mathcal{F}_{\mathrm{not}}$) from $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$. Basically, when the derivation contains a clause $R'$ with $\mathsf{sel}(R') \neq \emptyset$, we replace in this derivation two clauses $R$, with $\mathsf{sel}(R) = \emptyset$, and $R'$ that have been combined by resolution during the execution of saturate with a single clause $R \circ_{F_0} R'$. This replacement decreases the number of clauses in the derivation, so it terminates, and, upon termination, all clauses of the obtained derivation satisfy $\mathsf{sel}(R') = \emptyset$ so they are in $\mathsf{saturate}(\mathcal{R}_0) \cup \mathcal{F}_{\mathrm{me}}$.

Intuitively, the correctness of derivable expresses that if $F'$, instance of $F$, is derivable, then $F'$ is derivable from $\mathcal{R}_1$ by a derivation in which the clause that concludes $F'$ is in $\mathsf{derivable}(F, \mathcal{R}_1)$, provided the secrecy assumptions are satisfied.

**Lemma 3 (Correctness of** derivable**)** *Let $F'$ be a closed instance of $F$. If, for all $F'' \in \mathcal{F}_{\mathrm{not}}$,* $\mathsf{derivable}(F'', \mathcal{R}_1) = \emptyset$*, then $F'$ is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$ if and only if there exist a clause $H \Rightarrow C$ in* $\mathsf{derivable}(F, \mathcal{R}_1)$ *and a substitution $\sigma$ such that $\sigma C = F'$ and all elements of $\sigma H$ are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$.*

Basically, this result is proved by transforming a derivation of $F'$ from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$ into a derivation of $F'$ (or a fact in $\mathcal{F}_{\mathrm{not}}$) whose last clause (the one that concludes $F'$) is $H \Rightarrow C$ and whose other clauses are still in $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. The transformation relies on the replacement of clauses combined by resolution during the execution of derivable.

It is important to apply saturate before derivable, so that all clauses in $\mathcal{R}_1$ have no selected hypothesis. Then the conclusion of these clauses is in general not $\mathrm{attacker}(x)$ (with the simplifications of Section 6.2 and the selection function $\mathsf{sel}_0$, it is never $\mathrm{attacker}(x)$), so that we avoid unifying with $\mathrm{attacker}(x)$.

Finally, the following theorem shows the correctness of $\mathsf{solve}_{P'_0, Init}$ (Figure 5). Below, when we require that $\mathsf{solve}_{P'_0, Init}(F)$ has a certain value, we also implicitly require that $\mathsf{solve}_{P'_0, Init}(F)$ does not terminate with error. Intuitively, if an instance $F'$ of $F$ is satisfied by a trace $\mathcal{T}$, then $F'$ is derivable from $\mathcal{R}_{P'_0, Init} \cup \mathcal{F}_{\mathrm{me}}$, so, by the soundness of the solving algorithm, it is derivable by a derivation whose last clause is in $\mathsf{solve}_{P'_0, Init}(F)$. Then there must exist a clause $H \Rightarrow C \in \mathsf{solve}_{P'_0, Init}(F)$ that can be used to derive $F'$, so $F' = \sigma C$ and the hypothesis $\sigma H$ is derivable from $\mathcal{R}_{P'_0, Init} \cup \mathcal{F}_{\mathrm{me}}$. In particular, the events in $\sigma H$ are satisfied, that is, are in $\mathcal{F}_{\mathrm{me}}$, so these events have been executed in the trace $\mathcal{T}$. Theorem 2 below states this result formally. It is proved by combining Lemmas 2 and 3, and Theorem 1.

**Theorem 2 (Main theorem)** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let $Q$ be an $Init$-adversary and $Q' = \mathrm{instrAdv}(Q)$.*

*Consider a trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', P'$, with $fn(P'_0) \cup Init \subseteq dom(E_0)$ and $E_0(a) = a[\,]$ for all $a \in dom(E_0)$.*

*If $\mathcal{T}$ satisfies an instance $F'$ of $F$, then there exist a clause $H \Rightarrow C \in \mathsf{solve}_{P'_0,Init}(F)$ and a substitution $\sigma$ such that $F' = \sigma C$ and, for all $\text{m-event}(p)$ in $\sigma H$, $\mathcal{T}$ satisfies $\mathrm{event}(p)$.*

**Proof** Since for all $F'' \in \mathcal{F}_{\mathrm{not}}$, $\mathsf{derivable}(F'', \mathcal{R}_1) = \emptyset$, by Lemma 3, no instance of $F''$ is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}} = \mathsf{saturate}(\mathcal{R}_{P'_0,Init}) \cup \mathcal{F}_{\mathrm{me}}$. This allows us to apply Lemma 2.

Let $\mathcal{F}_{\mathrm{me}} = \{\text{m-event}(p') \mid \mathcal{T} \text{ satisfies } \mathrm{event}(p')\}$. By Theorem 1, since $\mathcal{T}$ satisfies $F'$, $F'$ is derivable from $\mathcal{R}_{P'_0,Init} \cup \mathcal{F}_{\mathrm{me}}$. By Lemma 2, $F'$ is derivable from $\mathsf{saturate}(\mathcal{R}_{P'_0,Init}) \cup \mathcal{F}_{\mathrm{me}} = \mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. By Lemma 3, there exist a clause $R = H \Rightarrow C$ in $\mathsf{solve}_{P'_0,Init}(F) = \mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution $\sigma$ such that $\sigma C = F'$ and all elements of $\sigma H$ are derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. For all $\text{m-event}(p)$ in $\sigma H$, $\text{m-event}(p)$ is derivable from $\mathcal{R}_1 \cup \mathcal{F}_{\mathrm{me}}$. Since no clause in $\mathcal{R}_1$ has a conclusion of the form $\text{m-event}(p')$, $\text{m-event}(p) \in \mathcal{F}_{\mathrm{me}}$. Given the choice of $\mathcal{F}_{\mathrm{me}}$, this means that $\mathcal{T}$ satisfies $\mathrm{event}(p)$. □

Theorem 2 is our main correctness result: it allows one to show that some events must have been executed. The correctness of the analysis for correspondences follows from this theorem.

**Example 8** For the process $P$ of Section 2.3, $Init = \{c\}$, and $P' = \mathrm{instr}(P)$, our tool shows that

$$\begin{aligned}\mathsf{solve}_{P',Init}(\mathrm{event}(e_B(x_1,x_2,x_3,x_4))) = \{&\text{m-event}(e_1(pk_A, pk_B, p_a)) \wedge\\ &\text{m-event}(e_2(pk_A, pk_B, p_a, p_b)) \wedge\\ &\text{m-event}(e_3(pk_A, pk_B, p_a, p_b))\\ &\Rightarrow \mathrm{event}(e_B(pk_A, pk_B, p_a, p_b))\}\end{aligned}$$

where $pk_A = pk(sk_A[\,]),\ pk_B = pk(sk_B[\,]),\ p_a = a[pk_B, i_A]$
$p_b = b[pencrypt_p((p_a, pk_A), pk_B, r_1[pk_B, i_A]), i_B]$

By Theorem 2, if $\mathcal{T}$ satisfies $\mathrm{event}(e_B(p_1,p_2,p_3,p_4))$, this event is an instance of $\mathrm{event}(e_B(x_1,x_2,x_3,x_4))$, so, given the value of $\mathsf{solve}_{P',Init}(\mathrm{event}(e_B(x_1,x_2,x_3,x_4)))$, there exists $\sigma$ such that $\mathrm{event}(e_B(p_1,p_2,p_3,p_4)) = \sigma\mathrm{event}(e_B(pk_A, pk_B, p_a, p_b))$ and $\mathcal{T}$ satisfies

$$\begin{aligned}&\mathrm{event}(\sigma e_1(pk_A, pk_B, p_a)) = \mathrm{event}(e_1(p_1,p_2,p_3))\\ &\mathrm{event}(\sigma e_2(pk_A, pk_B, p_a, p_b)) = \mathrm{event}(e_2(p_1,p_2,p_3,p_4))\\ &\mathrm{event}(\sigma e_3(pk_A, pk_B, p_a, p_b)) = \mathrm{event}(e_3(p_1,p_2,p_3,p_4))\end{aligned}$$

Therefore, if $\texttt{event}(e_B(M_1,M_2,M_3,M_4))$ has been executed, then $\texttt{event}(e_1(M_1, M_2, M_3))$, $\texttt{event}(e_2(M_1,M_2,M_3,M_4))$, and $\texttt{event}(e_3(M_1,M_2,M_3,M_4))$ have been executed.

# 7 Application to Correspondences

## 7.1 Non-injective Correspondences

Correspondences for instrumented processes can be checked as shown by the following theorem:

**Theorem 3** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let $p_{jk}$ ($j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, l_j\}$) be patterns; let $F$ and $F_j$ ($j \in \{1, \ldots, m\}$) be facts. Assume that for all $R \in \mathsf{solve}_{P'_0, Init}(F)$, there exist $j \in \{1, \ldots, m\}$, $\sigma'$, and $H$ such that $R = H \wedge \mathrm{m\text{-}event}(\sigma' p_{j1}) \wedge \ldots \wedge \mathrm{m\text{-}event}(\sigma' p_{jl_j}) \Rightarrow \sigma' F_j$.*

*Then $P'_0$ satisfies the correspondence $F \Rightarrow \bigvee_{j=1}^{m} \left( F_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(p_{jk}) \right)$ against $Init$-adversaries.*

**Proof** Let $Q$ be an $Init$-adversary and $Q' = \mathrm{instrAdv}(Q)$. Consider a trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$, with $fn(P'_0) \cup Init \subseteq dom(E_0)$ and $E_0(a) = a[\,]$ for all $a \in dom(E_0)$. Assume that $\mathcal{T}$ satisfies $\sigma F$. By Theorem 2, there exist $R = H' \Rightarrow C' \in \mathsf{solve}_{P'_0, Init}(F)$ and $\sigma''$ such that $\sigma F = \sigma'' C'$ and for all $\mathrm{m\text{-}event}(p)$ in $\sigma'' H'$, $\mathcal{T}$ satisfies $\mathrm{event}(p)$. All clauses $R$ in $\mathsf{solve}_{P'_0, Init}(F)$ are of the form $H \wedge \mathrm{m\text{-}event}(\sigma' p_{j1}) \wedge \ldots \wedge \mathrm{m\text{-}event}(\sigma' p_{jl_j}) \Rightarrow \sigma' F_j$ for some $j$ and $\sigma'$. So, there exist $j$ and $\sigma'$ such that for all $k \in \{1, \ldots, l_j\}$, $\mathrm{m\text{-}event}(\sigma' p_{jk}) \in H'$ and $C' = \sigma' F_j$. Hence $\sigma F = \sigma'' C' = \sigma'' \sigma' F_j$ and for all $k \in \{1, \ldots, l_j\}$, $\mathrm{m\text{-}event}(\sigma'' \sigma' p_{jk}) \in \sigma'' H'$, so $\mathcal{T}$ satisfies $\mathrm{event}(\sigma'' \sigma' p_{jk})$, so we have the result. □

From this theorem and Lemma 1, we obtain correspondences for standard processes.

**Theorem 4** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let $M_{jk}$ ($j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, l_j\}$) be terms; let $\alpha$ and $\alpha_j$ ($j \in \{1, \ldots, m\}$) be atoms. Let $p_{jk}, F, F_j$ be the patterns and facts obtained by replacing names $a$ with patterns $a[\,]$ in the terms and atoms $M_{jk}, \alpha, \alpha_j$ respectively. Assume that, for all clauses $R$ in $\mathsf{solve}_{P'_0, Init}(F)$, there exist $j \in \{1, \ldots, m\}$, $\sigma'$, and $H$ such that $R = H \wedge \mathrm{m\text{-}event}(\sigma' p_{j1}) \wedge \ldots \wedge \mathrm{m\text{-}event}(\sigma' p_{jl_j}) \Rightarrow \sigma' F_j$.*

*Then $P_0$ satisfies the correspondence $\alpha \Rightarrow \bigvee_{j=1}^{m} \left( \alpha_j \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(M_{jk}) \right)$ against $Init$-adversaries.*

**Example 9** For the process $P$ of Section 2.3, $Init = \{c\}$, and $P' = \mathrm{instr}(P)$, the value of $\mathsf{solve}_{P', Init}(\mathrm{event}(e_B(x_1, x_2, x_3, x_4)))$ given in Example 8 shows that $P$ satisfies the correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_1(x_1, x_2, x_3)) \wedge \mathrm{event}(e_2(x_1, x_2, x_3, x_4)) \wedge \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ against $Init$-adversaries.

As particular cases of correspondences, we can show secrecy and non-injective agreement:

**Corollary 1 (Secrecy)** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}(P_0)$. Let $N$ be a term. Let $p$ be the pattern obtained by replacing names $a$ with patterns $a[\,]$ in the term*

$N$. *Assume that* $\mathsf{solve}_{P'_0,Init}(\mathrm{attacker}(p)) = \emptyset$. *Then* $P_0$ *preserves the secrecy of all instances of* $N$ *from* $Init$.

Intuitively, if no instance of $\mathrm{attacker}(p)$ is derivable from the clauses representing the protocol, then the adversary cannot have an instance of the term $N$ corresponding to $p$.

**Example 10** For the process $P$ of Section 2.3, $Init = \{c\}$, and $P' = \mathrm{instr}(P)$, our tool shows that $\mathsf{solve}_{P',Init}(\mathrm{attacker}(sAa[])) = \emptyset$. So $P$ preserves the secrecy of $sAa$ from $Init$. The situation is similar for $sAb$, $sBa$, and $sBb$.

**Corollary 2 (Non-injective agreement)** *Let* $P_0$ *be a closed process and* $P'_0 = \mathrm{instr}(P_0)$. *Assume that, for each* $R \in \mathsf{solve}_{P'_0,Init}(\mathrm{event}(e(x_1,\ldots,x_n)))$ *such that* $R = H \Rightarrow \mathrm{event}(e(p_1,\ldots,p_n))$, *we have* $\mathrm{m\text{-}event}(e'(p_1,\ldots,p_n)) \in H$. *Then* $P_0$ *satisfies the correspondence* $\mathrm{event}(e(x_1,\ldots,x_n)) \rightsquigarrow \mathrm{event}(e'(x_1,\ldots,x_n))$ *against* $Init$*-adversaries.*

Intuitively, the condition means that, if $\mathrm{event}(e(p_1,\ldots,p_n))$ can be derived, $\mathrm{m\text{-}event}(e'(p_1,\ldots,p_n))$ occurs in the hypotheses. Then the theorem says that, if $\mathtt{event}(e(M_1,\ldots,M_n))$ has been executed, then $\mathtt{event}(e'(M_1,\ldots,M_n))$ has been executed.

**Example 11** For the process $P$ of Section 2.3, $Init = \{c\}$, and $P' = \mathrm{instr}(P)$, the value of $\mathsf{solve}_{P',Init}(\mathrm{event}(e_B(x_1,x_2,x_3,x_4)))$ given in Example 8 also shows that $P$ satisfies the correspondence $\mathrm{event}(e_B(x_1,x_2,x_3,x_4)) \rightsquigarrow \mathrm{event}(e_3(x_1,x_2,x_3,x_4))$ against $Init$-adversaries. The tool shows in a similar way that $P$ satisfies the correspondence $\mathrm{event}(e_A(x_1,x_2,x_3,x_4)) \rightsquigarrow \mathrm{event}(e_2(x_1,x_2,x_3,x_4))$ against $Init$-adversaries.

## 7.2 General Correspondences

In this section, we explain how to prove general correspondences. Moreover, we also show that, when our verifier proves injectivity, it proves recentness as well. For example, when it proves a correspondence $\mathrm{event}(M) \rightsquigarrow \mathrm{inj\ event}(M')$, it shows that, when the event $\mathtt{event}(M)$ has been executed, not only the event $\mathtt{event}(M')$ has been executed, but also this event has been executed recently. As explained by Lowe [55], the precise meaning of "recent" depends on the circumstances: it can be that $\mathtt{event}(M)$ is executed within the duration of the part of the process after $\mathtt{event}(M')$, or it can be within a certain number of time units. Here, we define recentness as follows: the runtime of the session that executes $\mathtt{event}(M)$ overlaps with the runtime of the session that executes the corresponding $\mathtt{event}(M')$ event.

We can formally define recent correspondences for instrumented processes as follows. We assume that, in $P_0$, the events are under at least one replication. We define an instrumented process $P'_0 = \mathrm{instr}'(P_0)$, where $\mathrm{instr}'(P_0)$ is defined like $\mathrm{instr}(P_0)$, except that the events $\mathtt{event}(M)$ in $P_0$ are replaced with $\mathtt{event}(M,i)$, where $i$ is the session identifier that labels the down-most replication above $\mathtt{event}(M)$ in $P_0$. The session identifier $i$ indicates the session in which the considered event is executed.

When $\overline{k} = k_1 \dots k_n$ is a non-empty sequence of indices, we denote by $\overline{k}\lceil$ the sequence obtained by removing the last index from $\overline{k}$: $\overline{k}\lceil = k_1 \dots k_{n-1}$.

**Definition 14** Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}'(P_0)$. We say that $P'_0$ *satisfies the recent correspondence*

$$\mathrm{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} q_{jk} \right)$$

where

$$q_{\overline{jk}} = \mathrm{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\mathrm{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

against $Init$-adversaries if and only if for any $Init$-adversary $Q$, for any trace $\mathcal{T} = S_0, E_0, \{P'_0, Q'\} \rightarrow^* S', E', \mathcal{P}'$, with $Q' = \mathrm{instrAdv}(Q)$, $E_0(a) = a[\,]$ for all $a \in dom(E_0)$, and $fn(P'_0) \cup Init \subseteq dom(E_0)$, there exists a function $\phi_{\overline{jk}}$ for each non-empty $\overline{jk}$, such that for all non-empty $\overline{jk}$, $\phi_{\overline{jk}}$ maps a subset of steps of $\mathcal{T}$ to steps of $\mathcal{T}$ and

- For all $\tau$, if the event $\mathtt{event}(\sigma p, \lambda_\epsilon)$ is executed at step $\tau$ in $\mathcal{T}$ for some $\sigma$ and $\lambda_\epsilon$, then there exist $\sigma'$ and $J = (j_{\overline{k}})_{\overline{k}}$ such that $\sigma' p'_{j_\epsilon} = \sigma p$ and, for all non-empty $\overline{k}$, $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$ is defined, $\mathtt{event}(\sigma' p_{\mathrm{makejk}(\overline{k},J)}, \lambda_{\overline{k}})$ is executed at step $\phi_{\mathrm{makejk}(\overline{k},J)}(\tau)$ in $\mathcal{T}$, and if $[\mathrm{inj}]_{\mathrm{makejk}(\overline{k},J)} = \mathrm{inj}$, then the runtimes of $\mathrm{session}(\lambda_{\overline{k}\lceil})$ and $\mathrm{session}(\lambda_{\overline{k}})$ overlap (recentness).

  The runtime of $\mathrm{session}(\lambda)$ begins when the rule $S, E, \mathcal{P} \cup \{\, !^i P \,\} \rightarrow S \setminus \{\lambda\}$, $E, \mathcal{P} \cup \{\, P\{\lambda/i\}, !^i P \,\}$ is applied and ends when $P\{\lambda/i\}$ has disappeared.

- For all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, then $\phi_{\overline{jk}}$ is injective.

- For all non-empty $\overline{jk}$, for all $j$ and $k$, if $\phi_{\overline{jk}jk}(\tau)$ is defined, then $\phi_{\overline{jk}}(\tau)$ is defined and $\phi_{\overline{jk}jk}(\tau) \leq \phi_{\overline{jk}}(\tau)$. For all $j$ and $k$, if $\phi_{jk}(\tau)$ is defined, then $\phi_{jk}(\tau) \leq \tau$.

We do not define recentness for standard processes, since it is difficult to track formally the runtime of a session in these processes. Instrumented processes make that very easy thanks to session identifiers. It is easy to infer correspondences for standard processes from recent correspondences for instrumented processes, with a proof similar to that of Lemma 1.

**Lemma 4** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}'(P_0)$. Let $M_{\overline{jk}}$, $M$, and $M'_j$ be terms. Let $p_{\overline{jk}}, p, p'_j$ be the patterns obtained by replacing names $a$ with patterns $a[\,]$ in the terms $M_{\overline{jk}}, M, M'_j$ respectively. If $P'_0$ satisfies the recent correspondence*

$$\mathrm{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} q_{jk} \right)$$

*where*

$$q_{\overline{jk}} = \mathrm{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\mathrm{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

*against $Init$-adversaries then $P_0$ satisfies the correspondence*

$$\mathrm{event}(M) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(M'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} q'_{jk} \right)$$

*where*

$$q'_{\overline{jk}} = \mathrm{event}(M_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\mathrm{inj}]_{\overline{jk}jk} q'_{\overline{jk}jk}$$

*against $Init$-adversaries.*

Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}'(P_0)$. We adapt the generation of clauses as follows: the set of clauses $\mathcal{R}'_{P'_0,Init}$ is defined as $\mathcal{R}_{P'_0,Init}$ except that

$$[\![\overline{M}\langle N\rangle.P]\!]\rho H = [\![P]\!]\rho H \cup \{H\{\rho_{|V_o \cup V_s}/\Box\} \Rightarrow \mathrm{message}(\rho(M), \rho(N))\}$$
$$[\![!^i P]\!]\rho H = [\![P]\!](\rho[i \mapsto i])(H\{\rho_{|V_o \cup V_s}/\Box\})$$
$$[\![\mathtt{event}(M, i).P]\!]\rho H = [\![P]\!]\rho(H \wedge \text{m-event}(\rho(M), \Box)) \cup \{H \Rightarrow \mathrm{event}(\rho(M), i)\}$$

where $\Box$ is a special variable. The predicate $\mathrm{event}$ has as additional argument the session identifier in which the event is executed. The predicate $\text{m-event}$ has as additional argument an environment $\rho$ that gives values that variables will contain at the first output or replication that follows the event; $\Box$ is a placeholder for this environment. (Recall that $V_o$ is the set of ordinary variables and $V_s$ the set of session identifier variables, so $\rho_{|V_o \cup V_s}$ is the environment restricted to variables, names being excluded.) We define $\mathsf{solve}'_{P'_0,Init}$ as $\mathsf{solve}_{P'_0,Init}$ except that it applies to $\mathcal{R}'_{P'_0,Init}$ instead of $\mathcal{R}_{P'_0,Init}$.

Let us first consider the particular case of injective correspondences. We consider general correspondences in Theorem 5 below.

**Proposition 2 (Injective correspondences)** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}'(P_0)$. We assume that, in $P_0$, all events are of the form $\mathtt{event}(f(M_1, \ldots, M_n))$ and that different occurrences of $\mathtt{event}$ have different root function symbols.*

*We also assume that the patterns $p, p'_j, p_{jk}$ satisfy the following conditions: $p$ and $p'_j$ for $j \in \{1, \ldots, m\}$ are of the form $f(\ldots)$ for some function symbol $f$ and for all $j$, $k$ such that $[\mathrm{inj}]_{jk} = \mathrm{inj}$, $p_{jk} = f_{jk}(\ldots)$ for some function symbol $f_{jk}$.*

*Let $\mathsf{solve}'_{P'_0,Init}(\mathrm{event}(p, i)) = \{R_{jr} \mid j \in \{1, \ldots, m\}, r \in \{1, \ldots, n_j\}\}$. Assume that there exist $x_{jk}$, $i_{jr}$, and $\rho_{jrk}$ ($j \in \{1, \ldots, m\}$, $r \in \{1, \ldots, n_j\}$, $k \in \{1, \ldots, l_j\}$) such that*

- *For all $j \in \{1, \ldots, m\}$, for all $r \in \{1, \ldots, n_j\}$, there exist $H$ and $\sigma$ such that $R_{jr} = H \wedge \text{m-event}(\sigma p_{j1}, \rho_{jr1}) \wedge \ldots \wedge \text{m-event}(\sigma p_{jl_j}, \rho_{jrl_j}) \Rightarrow \mathrm{event}(\sigma p'_j, i_{jr})$.*

- *For all $j \in \{1, \ldots, m\}$, for all $r$ and $r'$ in $\{1, \ldots, n_j\}$, for all $k \in \{1, \ldots, l_j\}$ such that $[\mathrm{inj}]_{jk} = \mathrm{inj}$, $\rho_{jrk}(x_{jk})\{\lambda/i_{jr}\}$ does not unify with $\rho_{jr'k}(x_{jk})\{\lambda'/i_{jr'}\}$ when $\lambda \neq \lambda'$.*

*Then $P'_0$ satisfies the recent correspondence*

$$\mathrm{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} \mathrm{event}(p_{jk}) \right)$$

*against $Init$-adversaries.*

This proposition is a particular case of Theorem 5 below. It is proved in [18, Appendix E]. By Theorem 3, after deleting session identifiers and environments, the first item shows that $P'_0$ satisfies the correspondence

$$\mathrm{event}(p) \Rightarrow \bigvee_{j=1..m,r} \left( \mathrm{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(p_{jk}) \right) \tag{13}$$

The environments and session identifiers as well as the second item serve in proving injectivity. Suppose that $[\mathrm{inj}]_{jk} = \mathrm{inj}$, and denote by $\_$ an unknown term. If two instances of $\mathtt{event}(p, i)$ are executed in $P'_0$ for the branch $j$ of the correspondence, by the first item, they are instances of $\mathtt{event}(\sigma_{jr} p'_j, i_{jr})$ for some $r$, so they are $\mathtt{event}(\sigma'_1 \sigma_{jr_1} p'_j, \sigma'_1 i_{jr_1})$ and $\mathtt{event}(\sigma'_2 \sigma_{jr_2} p'_j, \sigma'_2 i_{jr_2})$ for some $\sigma'_1$ and $\sigma'_2$. Furthermore, there is only one occurrence of $\mathtt{event}(f(\ldots), i)$ in $P'_0$, so the event $\mathtt{event}(f(\ldots), i)$ can be executed at most once for each value of the session identifier $i$, so $\sigma'_1 i_{jr_1} \neq \sigma'_2 i_{jr_2}$. Then, by the first item, corresponding events $\mathtt{event}(\sigma'_1 \sigma_{jr_1} p_{jk}, \_)$ and $\mathtt{event}(\sigma'_2 \sigma_{jr_2} p_{jk}, \_)$ have been executed, with associated environments $\sigma'_1 \rho_{jr_1 k}$ and $\sigma'_2 \rho_{jr_2 k}$. By the second item, $\rho_{jr_1 k}(x_{jk})\{\lambda_1/i_{jr_1}\}$ does not unify with $\rho_{jr_2 k}(x_{jk})\{\lambda_2/i_{jr_2}\}$ for different values $\lambda_1 = \sigma'_1 i_{jr_1}$ and $\lambda_2 = \sigma'_2 i_{jr_2}$ of the session identifier. (In this condition, $r_1$ can be equal to $r_2$, and when $r_1 = r_2 = r$, the condition simply means that $i_{jr}$ occurs in $\rho_{jrk}$.) So $\sigma'_1 \rho_{jr_1 k}(x_{jk}) \neq \sigma'_2 \rho_{jr_2 k}(x_{jk})$, so the events $\mathtt{event}(\sigma'_1 \sigma_{jr_1} p_{jk}), \_)$ and $\mathtt{event}(\sigma'_2 \sigma_{jr_2} p_{jk}), \_)$ are distinct, which shows injectivity. This point is very similar to the fact that injective agreement is implied by non-injective agreement when the parameters of events contain nonces generated by the agent to whom authentication is being made, because the event can be executed at most once for each value of the nonce. (The session identifier $i_{jr}$ in our theorem plays the role of the nonce.) [Andrew Gordon, personal communication].

**Corollary 3 (Recent injective agreement)** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}'(P_0)$. We assume that, in $P_0$, all events are of the form $\mathtt{event}(f(M_1, \ldots, M_k))$ and that different occurrences of $\mathtt{event}$ have different root function symbols. Let $\{R_1, \ldots, R_n\} = \mathsf{solve}'_{P'_0, Init}(\mathrm{event}(e(x_1, \ldots, x_m), i))$. Assume that there exist $x$, $i_r$, and $\rho_r$ ($r \in \{1, \ldots, n\}$) such that*

- *For all $r \in \{1, \ldots, n\}$, $R_r = H \wedge \mathrm{m\text{-}event}(e'(p_1, \ldots, p_m), \rho_r) \Rightarrow \mathrm{event}(e(p_1, \ldots, p_m), i_r)$ for some $p_1, \ldots, p_m$, and $H$.*

- *For all $r$ and $r'$ in $\{1, \ldots, n\}$, $\rho_r(x)\{\lambda/i_r\}$ does not unify with $\rho_{r'}(x)\{\lambda'/i_{r'}\}$ when $\lambda \neq \lambda'$.*

*Then $P'_0$ satisfies the recent correspondence* $\mathrm{event}(e(x_1, \ldots, x_m)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e'(x_1, \ldots, x_m))$ *against $Init$-adversaries.*

**Proof** This result is an immediate consequence of Proposition 2. □

**Example 12** For the process $P$ of Section 2.3, $P' = \mathrm{instr}'(P)$, and $Init = \{c\}$, we have

$$\begin{array}{l}
\mathsf{solve}'_{P',Init}(\mathrm{event}(e_B(x_1, x_2, x_3, x_4), i)) = \\
\quad \{H \wedge \text{m-event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), \rho) \\
\qquad \Rightarrow \mathrm{event}(e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i_{B0})\} \\
\text{where } pk_A = pk(sk_A[\,]),\ pk_B = pk(sk_B[\,]) \\
\qquad p_1 = pencrypt_p((a[pk_B, i_{A0}], pk_A), pk_B, r_1[pk_B, i_{A0}]) \\
\qquad p_2 = pencrypt_p((a[pk_B, i_{A0}], b[p_1, i_{B0}], pk_B), pk_A, r_2[p_1, i_{B0}]) \\
\qquad \rho = \{i_A \mapsto i_{A0}, x_pk_B \mapsto pk_B, m \mapsto p_2\}
\end{array}$$

Intuitively, this result shows that each event $e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$, executed in the session of index $i_B = i_{B0}$ is preceded by an event $e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_A = i_{A0}$ with $x\_pk_B = pk_B$ and $m = p_2$. Since $i_{B0}$ occurs in this event (or in its environment[4]), different executions of $e_B$, which have different values of $i_{B0}$, cannot correspond to the same execution of $e_3$, so we have injectivity. More formally, the second hypothesis of Corollary 3 is satisfied because $\rho(m)\{\lambda/i_{B0}\}$ does not unify with $\rho(m)\{\lambda'/i_{B0}\}$ when $\lambda \neq \lambda'$, since $i_{B0}$ occurs in $\rho(m) = p_2$. Then, $P'$ satisfies the recent correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ against $Init$-adversaries.

The tool shows in a similar way that $P'$ satisfies the recent correspondence $\mathrm{event}(e_A(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_2(x_1, x_2, x_3, x_4))$ against $Init$-adversaries.

Let us now consider the case of general correspondences. The basic idea is to decompose the general correspondence to prove into several correspondences. For instance, the correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_2(x_1, x_2, x_3, x_4)))$ is implied by the conjunction of the correspondences $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ and $\mathrm{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{event}(e_2(x_1, x_2, x_3, x_4))$. However, as noted in Section 3.3, this proof technique would often fail because, in order to prove that $e_2(x_1, x_2, x_3, x_4)$ has been executed, we may need to know that $e_B(x_1, x_2, x_3, x_4)$ has been executed, and not only that $e_3(x_1, x_2, x_3, x_4)$ has been executed. To solve this problem, we use the following idea: when we know that $e_B(x_1, x_2, x_3, x_4)$ has been executed, we may be able to show that certain particular instances of $e_3(x_1, x_2, x_3, x_4)$ have been executed, and we can exploit this information in order to prove that $e_2(x_1, x_2, x_3, x_4)$ has been

[4] In general, the environment may contain more variables than the event itself, so looking for the session identifiers in the environment instead of the event is more powerful.

executed. In other words, we rather prove the correspondences $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \Rightarrow \bigvee_{r=1}^{m} \sigma_r \mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow \sigma_r \mathrm{event}(e_3(x_1, x_2, x_3, x_4))$ and for all $r \leq m$, $\sigma_r \mathrm{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow \sigma_r \mathrm{event}(e_2(x_1, x_2, x_3, x_4))$. When the considered general correspondence has several nesting levels, we perform such a decomposition recursively. The next theorem generalizes and formalizes these ideas.

Below, the notation $(Env_{\overline{jk}})_{\overline{jk}}$ represents a family $Env_{\overline{jk}}$ of sets of pairs $(\rho, i)$ where $\rho$ is an environment and $i$ is a session identifier, one for each non-empty $\overline{jk}$. The notation $(Env_{jk\overline{jk}})_{\overline{jk}}$ represents a subfamily of $(Env_{\overline{jk}})_{\overline{jk}}$ in which the first two indices are $jk$, and this family is reindexed by omitting the fixed indices $jk$.

**Theorem 5** *Let $P_0$ be a closed process and $P'_0 = \mathrm{instr}'(P_0)$. We assume that, in $P_0$, all events are of the form* `event`$(f(M_1, \ldots, M_n))$ *and that different occurrences of* `event` *have different root function symbols.*

*Let us define* $\mathrm{verify}(q', (Env_{\overline{jk}})_{\overline{jk}})$*, where $\overline{jk}$ is non-empty, by:*

*V1. If $q' = \mathrm{event}(p)$ for some $p$, then $\mathrm{verify}(q', (Env_{\overline{jk}})_{\overline{jk}})$ is true.*

*V2. If $q' = \mathrm{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} q'_{jk} \right)$ and $q'_{jk} = \mathrm{event}(p_{jk}) \rightsquigarrow \ldots$ for some $p$, $p'_j$, and $p_{jk}$, where $m \neq 1$, $l_j \neq 0$, or $p \neq p'_1$, then $\mathrm{verify}(q', (Env_{\overline{jk}})_{\overline{jk}})$ is true if and only if there exists $(\sigma_{jr})_{jr}$ such that the following three conditions hold:*

*V2.1. We have $\mathsf{solve}'_{P'_0, Init}(\mathrm{event}(p, i)) \subseteq \{H \wedge \bigwedge_{k=1}^{l_j} \mathrm{m\text{-}event}(\sigma_{jr} p_{jk}, \rho_{jrk}) \Rightarrow \mathrm{event}(\sigma_{jr} p'_j, i_{jr})$ for some $H$, $j \in \{1, \ldots, m\}$, $r$, and $(\rho_{jrk}, i_{jr}) \in Env_{jk}$ for all $k\}$.*

*V2.2. For all $j, r, k_0$, the common variables between $\sigma_{jr} q'_{jk_0}$ on the one hand and $\sigma_{jr} p'_j$ and $\sigma_{jr} q'_{jk}$ for all $k \neq k_0$ on the other hand occur in $\sigma_{jr} p_{jk_0}$.*

*V2.3. For all $j, r, k$, $\mathrm{verify}(\sigma_{jr} q'_{jk}, (Env_{jk\overline{jk}})_{\overline{jk}})$ is true.*

*Consider the following recent correspondence:*

$$q = \mathrm{event}(p) \Rightarrow \bigvee_{j=1}^{m} \left( \mathrm{event}(p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} [\mathrm{inj}]_{jk} q_{jk} \right)$$

*where*

$$q_{\overline{jk}} = \mathrm{event}(p_{\overline{jk}}) \rightsquigarrow \bigvee_{j=1}^{m_{\overline{jk}}} \bigwedge_{k=1}^{l_{\overline{jk}j}} [\mathrm{inj}]_{\overline{jk}jk} q_{\overline{jk}jk}$$

*We assume that the patterns in the correspondence satisfy the following conditions: $p$ and $p'_j$ for $j \in \{1, \ldots, m\}$ are of the form $f(\ldots)$ for some function symbol $f$ and, for all non-empty $\overline{jk}$ such that $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, $p_{\overline{jk}} = f_{\overline{jk}}(\ldots)$ for some function symbol $f_{\overline{jk}}$. We also assume that if $\mathrm{inj}$ occurs in $q_{\overline{jk}}$, then $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$.*

*Assume that there exist $(Env_{\overline{jk}})_{\overline{jk}}$ and $(x_{\overline{jk}})_{\overline{jk}}$, where $\overline{jk}$ is non-empty, such that*

*H1. $\mathrm{verify}(q, (Env_{\overline{jk}})_{\overline{jk}})$ is true.*

*H2. For all non-empty $\overline{jk}$, if $[\mathrm{inj}]_{\overline{jk}} = \mathrm{inj}$, then for all $(\rho, i), (\rho', i') \in Env_{\overline{jk}}$, $\rho(x_{\overline{jk}})\{\lambda/i\}$ does not unify with $\rho'(x_{\overline{jk}})\{\lambda'/i'\}$ when $\lambda \neq \lambda'$.*

*Then $P'_0$ satisfies the recent correspondence $q$ against $Init$-adversaries.*

This theorem is rather complex, so we give some intuition here. Its proof can be found in [18, Appendix E].

Point V2.1 allows us to infer correspondences by Theorem 3: after deleting session identifiers and environments, $P'_0$ satisfies the correspondences:

$$\mathrm{event}(p) \Rightarrow \bigvee_{j=1..m,r} \left( \mathrm{event}(\sigma_{jr} p'_j) \rightsquigarrow \bigwedge_{k=1}^{l_j} \mathrm{event}(\sigma_{jr} p_{jk}) \right) \tag{14}$$

and, using the recursive calls of Point V2.3,

$$\mathrm{event}(\sigma'_{\overline{jrk}\lceil} p_{\overline{jk}}) \Rightarrow \bigvee_{j=1..m_{\overline{jk}},r} \left( \mathrm{event}(\sigma'_{\overline{jrk}jr} p_{\overline{jk}}) \rightsquigarrow \bigwedge_{k=1}^{l_{\overline{jk}j}} \mathrm{event}(\sigma'_{\overline{jrk}jr} p_{\overline{jk}jk}) \right) \tag{15}$$

against $Init$-adversaries, where $\sigma'_{\overline{jrk}jr} = \sigma_{\overline{jrk}jr} \sigma_{\overline{jrk}\lceil} \dots \sigma_{jr}$ and we denote by $\sigma_{\overline{jrk}jr}$ the substitution $\sigma_{jr}$ obtained in recursive calls to verify indexed by $\overline{jrk}$. In order to infer the desired correspondence, we need to show injectivity properties and to combine the correspondences (14) and (15) into a single correspondence. Injectivity comes from Hypothesis H2: this hypothesis generalizes the second item of Proposition 2 to the case of general correspondences.

The correspondences (14) and (15) are combined into a single correspondence using Point V2.2. We illustrate this point on the simple example of the correspondence $\mathrm{event}(p) \Rightarrow (\mathrm{event}(p'_1) \rightsquigarrow (\mathrm{event}(p_{11}) \rightsquigarrow \mathrm{event}(p_{1111})))$. By V2.1 and the recursive call of V2.3, we have correspondences of the form:

$$\mathrm{event}(p) \Rightarrow \bigvee_{r} (\mathrm{event}(\sigma_{1r} p'_1) \rightsquigarrow \mathrm{event}(\sigma_{1r} p_{11})) \tag{16}$$

$$\mathrm{event}(\sigma_{1r} p_{11}) \Rightarrow \bigvee_{r'} (\mathrm{event}(\sigma_{1r11r'} \sigma_{1r} p_{11}) \rightsquigarrow \mathrm{event}(\sigma_{1r11r'} \sigma_{1r} p_{1111})) \tag{17}$$

for some $\sigma_{1r}$ and $\sigma_{1r11r'}$. The correspondence (17) implies the simpler correspondence

$$\mathrm{event}(\sigma_{1r} p_{11}) \rightsquigarrow \mathrm{event}(\sigma_{1r} p_{1111}). \tag{18}$$

Furthermore, if an instance of $\mathrm{event}(p)$ is executed, $e_1 = \mathrm{event}(\sigma p)$, then by (16), for some $r$ and $\sigma'_1$ such that $\sigma p = \sigma'_1 \sigma_{1r} p'_1$, the event $e_2 = \mathrm{event}(\sigma'_1 \sigma_{1r} p_{11})$ has been executed before $e_1$. By (18), for some $\sigma'_2$ such that $\sigma'_1 \sigma_{1r} p_{11} = \sigma'_2 \sigma_{1r} p_{11}$, the event $e_3 = \mathrm{event}(\sigma'_2 \sigma_{1r} p_{1111})$ has been executed before $e_2$. We now need to reconcile the substitutions $\sigma'_1$ and $\sigma'_2$; this can be done thanks to V2.2. Let us define $\sigma''$ such that $\sigma'' x = \sigma'_1 x$ for $x \in fv(\sigma_{1r} p_{11}) \cup fv(\sigma_{1r} p'_1)$ and $\sigma'' x = \sigma'_2 x$ for $x \in fv(\sigma_{1r} p_{1111}) \cup fv(\sigma_{1r} p_{11})$. Such a substitution $\sigma''$ exists because the common variables between $fv(\sigma_{1r} p_{11}) \cup fv(\sigma_{1r} p'_1)$ and $fv(\sigma_{1r} p_{1111}) \cup fv(\sigma_{1r} p_{11})$ occur in $\sigma_{1r} p_{11}$ by V2.2, and for the variables $x \in fv(\sigma_{1r} p_{11})$, $\sigma'_1 x = \sigma'_2 x$ since

$\sigma'_1\sigma_{1r}p_{11} = \sigma'_2\sigma_{1r}p_{11}$. So, for some $r$ and $\sigma''$ such that $\sigma p = \sigma''\sigma_{1r}p'_1$, the event $e_2 = \mathrm{event}(\sigma''\sigma_{1r}p_{11})$ has been executed before $e_1$ and $e_3 = \mathrm{event}(\sigma''\sigma_{1r}p_{1111})$ has been executed before $e_2$. This result proves the desired correspondence $\mathrm{event}(p) \Rightarrow (\mathrm{event}(p'_1) \rightsquigarrow (\mathrm{event}(p_{11}) \rightsquigarrow \mathrm{event}(p_{1111}))$. Point V2.2 generalizes this technique to any correspondence.

In the implementation, the hypotheses of this theorem are checked as follows. In order to check $\mathrm{verify}(q', (Env_{\overline{jk}})_{\overline{jk}})$, we first compute $\mathsf{solve}'_{P'_0,Init}(\mathrm{event}(p,i))$. By matching, we check V2.1 and obtain the values of $\sigma_{jr}$, $\rho_{jrk}$, and $i_{jr}$ for all $j$, $r$, and $k$. We add $(\rho_{jrk}, i_{jr})$ to $Env_{jk}$. We compute $\sigma_{jr}p'_j$ and $\sigma_{jr}q'_{jk}$ for each $j$, $r$, and $k$, and check V2.2 and V2.3.

After checking $\mathrm{verify}(q', (Env_{\overline{jk}})_{\overline{jk}})$, we finally check Hypothesis H2 for each $\overline{jk}$. We start with a set that contains the whole domain of $\rho$ for some $(\rho, i) \in Env_{\overline{jk}}$. For each $(\rho, i)$ and $(\rho', i')$ in $Env_{\overline{jk}}$, we remove from this set the variables $x$ such that $\rho(x)\{\lambda/i\}$ unifies with $\rho'(x)\{\lambda'/i'\}$ for $\lambda \neq \lambda'$. When the obtained set is non-empty, Hypothesis H2 is satisfied by taking for $x_{\overline{jk}}$ any element of the obtained set. Otherwise, Hypothesis H2 is not satisfied.

**Example 13** For the example $P$ of Section 2.3, the previous theorem does not enable us to prove the correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj\ event}(e_1(x_1, x_2, x_3))))$ directly. Indeed, Theorem 5 would require that we show a correspondence of the form $\mathrm{event}(\sigma e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj\ event}(\sigma e_1(x_1, x_2, x_3))$. However, such a correspondence does not hold, because after executing a single event $e_1$, the adversary can replay the first message of the protocol, so that $B$ executes several events $e_2$.

It is still possible to prove this correspondence by combining the automatic proof of the slightly weaker correspondence $q = \mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_1(x_1, x_2, x_3)) \wedge \mathrm{inj\ event}(e_2(x_1, x_2, x_3, x_4))))$, which does not order the events $e_1$ and $e_2$, with a simple manual argument. (This technique applies to many other examples.) Let us first prove the latter correspondence.

Let $P' = \mathrm{instr}'(P)$ and $Init = \{c\}$. We have

$$\begin{aligned}
&\mathsf{solve}'_{P',Init}(\mathrm{event}(e_B(x_1, x_2, x_3, x_4), i)) = \\
&\quad \{H \wedge \text{m-event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), \rho_{111}) \\
&\qquad \Rightarrow \mathrm{event}(e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i_{B0})\} \\
&\mathsf{solve}'_{P',Init}(\mathrm{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i)) = \\
&\quad \{\text{m-event}(e_1(pk_A, pk_B, a[pk_B, i_{A0}]), \rho_{111111}) \\
&\quad \wedge \text{m-event}(e_2(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), \rho_{111112}) \\
&\qquad \Rightarrow \mathrm{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i_{A0})\}
\end{aligned}$$

where $pk_A = pk(sk_A[]),\ pk_B = pk(sk_B[])$

$p_1 = pencrypt_p((a[pk_B, i_{A0}], pk_A), pk_B, r_1[pk_B, i_{A0}])$

$p_2 = pencrypt_p((a[pk_B, i_{A0}], b[p_1, i_{B0}], pk_B), pk_A, r_2[p_1, i_{B0}])$

$\rho_{111} = \rho_{111111} = \{i_A \mapsto i_{A0}, x\_pk_B \mapsto pk_B, m \mapsto p_2\}$

$\rho_{111112} = \{i_B \mapsto i_{B0}, m' \mapsto p_1\}$

Intuitively, as in Example 12, the value of $\mathsf{solve}'_{P',Init}(\mathrm{event}(e_B(x_1, x_2, x_3, x_4), i))$ guarantees that each event $e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$, executed in the session of index $i_B = i_{B0}$ is preceded by an event $e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_A = i_{A0}$ with $x\_pk_B = pk_B$ and $m = p_2$. Since $i_{B0}$ occurs in this event (or in its environment), we have injectivity. The value of $\mathsf{solve}'_{P',Init}(\mathrm{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}]), i))$ guarantees that each event $e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_A = i_{A0}$ is preceded by events $e_1(pk_A, pk_B, a[pk_B, i_{A0}])$ executed in the session of index $i_A = i_{A0}$ with $x\_pk_B = pk_B$ and $m = p_2$, and $e_2(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ executed in the session of index $i_B = i_{B0}$ with $m' = p_1$. Since $i_{A0}$ occurs in these events (or in their environments), we have injectivity. So we obtain the desired correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_1(x_1, x_2, x_3)) \wedge \mathrm{inj\ event}(e_2(x_1, x_2, x_3, x_4))))$.

More formally, let us show that we can apply Theorem 5. We have $p = p'_1 = e_B(x_1, x_2, x_3, x_4)$, $p_{11} = e_3(x_1, x_2, x_3, x_4)$, $p_{1111} = e_1(x_1, x_2, x_3)$, $p_{1112} = e_2(x_1, x_2, x_3, x_4)$. We show $\mathrm{verify}(q, (Env_{\overline{jk}})_{\overline{jk}})$. Given the first value of $\mathsf{solve}'_{P',Init}$ shown above, we satisfy V2.1 by letting $\sigma_{11} = \{x_1 \mapsto pk_A, x_2 \mapsto pk_B, x_3 \mapsto a[pk_B, i_{A0}], x_4 \mapsto b[p_1, i_{B0}]\}$ and $i_{11} = i_{B0}$, with $(\rho_{111}, i_{11}) \in Env_{11}$. The common variables between $\sigma_{11}q_{11} = \mathrm{event}(e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])) \rightsquigarrow (\mathrm{inj\ event}(e_1(pk_A, pk_B, a[pk_B, i_{A0}])) \wedge \mathrm{inj\ event}(e_2(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])))$ and $\sigma_{11}p'_1 = e_B(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$ are $i_{A0}$ and $i_{B0}$, and they occur in $\sigma_{11}p_{11} = e_3(pk_A, pk_B, a[pk_B, i_{A0}], b[p_1, i_{B0}])$. So we have V2.2. Recursively, in order to obtain V2.3, we have to show $\mathrm{verify}(\sigma_{11}q_{11}, (Env_{11\overline{jk}})_{\overline{jk}})$. Given the second value of $\mathsf{solve}'_{P',Init}$ shown above, we satisfy V2.1 by letting $\sigma_{11111} = \mathrm{Id}$ and $i_{11111} = i_{A0}$, with $(\rho_{111111}, i_{11111}) \in Env_{1111}$ and $(\rho_{111112}, i_{11111}) \in Env_{1112}$. (We prefix the indices with 111 in order to represent that these values concern the recursive call with $j = 1$, $r = 1$, and $k = 1$.) V2.2 holds trivially, because $\sigma_{11111}\sigma_{11}q_{111k_0} = \sigma_{11111}\sigma_{11}\mathrm{event}(p_{111k_0})$, since the considered correspondence has one nesting level only. V2.3 holds because $q_{1111}$ reduces to $\mathrm{event}(p_{1111})$, so $\mathrm{verify}(\sigma_{11111}\sigma_{11}q_{1111}, (Env_{1111\overline{jk}})_{\overline{jk}})$ holds by V1, and the situation is similar for $q_{1112}$. Therefore, we obtain H1. In order to show H2, we have to find $x_{11}$ such that $\rho_{111}(x_{11})\{\lambda/i_{11}\}$ does not unify with $\rho_{111}(x_{11})\{\lambda'/i_{11}\}$ when $\lambda \neq \lambda'$. This property holds with $x_{11} = m$, because $i_{11} = i_{B0}$ occurs in $\rho_{111}(m) = p_2$. Similarly, $\rho_{111111}(x_{1111})\{\lambda/i_{11111}\}$ does not unify with $\rho_{111111}(x_{1111})\{\lambda'/i_{11111}\}$ when $\lambda \neq \lambda'$, for $x_{1111} = i_A$, since $i_{11111} = i_{A0}$ occurs in $\rho_{111111}(i_A)$. Finally, $\rho_{111112}(x_{1112})\{\lambda/i_{11111}\}$ does not unify with $\rho_{111112}(x_{1112})\{\lambda'/i_{11111}\}$ when $\lambda \neq \lambda'$ for $x_{1112} = m'$, since $i_{11111} = i_{A0}$ occurs in $\rho_{111112}(m') = p_1$. So, by Theorem 5, the process $P'$ satisfies the recent correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj\ event}(e_1(x_1, x_2, x_3)) \wedge \mathrm{inj\ event}(e_2(x_1, x_2, x_3,$

$x_4))))$ against $Init$-adversaries.

We can then show that $P'$ satisfies the recent correspondence $\mathrm{event}(e_B(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj}\ \mathrm{event}(e_3(x_1, x_2, x_3, x_4)) \rightsquigarrow (\mathrm{inj}\ \mathrm{event}(e_2(x_1, x_2, x_3, x_4)) \rightsquigarrow \mathrm{inj}\ \mathrm{event}(e_1(x_1, x_2, x_3))))$. We just have to show that the event $e_2(x_1, x_2, x_3, x_4)$ is executed after $e_1(x_1, x_2, x_3)$. The nonce $a$ is created just before executing $e_1(x_1, x_2, x_3) = e_1(pk_A, x\_pk_B, a)$, and the event $e_2(x_1, x_2, x_3, x_4) = e_2(x\_pk_A, pk_B, x\_a, b)$ contains $a$ in the variable $x_3 = x\_a$. So $e_2$ has been executed after receiving a message that contains $a$, so after $a$ has been sent in some message, so after executing event $e_1$.

# 8 Termination

In this section, we study termination properties of our algorithm. We first show that it terminates on a restricted class of protocols, named *tagged protocols*. Then, we study how to improve the choice of the selection function in order to obtain termination in other cases.

## 8.1 Termination for Tagged Protocols

Intuitively, a tagged protocol is a protocol in which each application of a constructor can be immediately distinguished from others in the protocol, for example by a tag: for instance, when we want to encrypt $m$ under $k$, we add the constant tag $ct_0$ to $m$, so that the encryption becomes $sencrypt((ct_0, m), k)$ where the tag $ct_0$ is a different constant for each encryption in the protocol. The tags are checked when destructors are applied. This condition is easy to realize by adding tags, and it is also a good protocol design: the participants use the tags to identify the messages unambiguously, thus avoiding type flaw attacks [51].

In [21], in collaboration with Andreas Podelski, we have given conditions on the clauses that intuitively correspond to tagged protocols, and we have shown that, for tagged protocols using only public channels, public-key cryptography with atomic keys, shared-key cryptography and hash functions, and for secrecy properties, the solving algorithm using the selection function $\mathsf{sel}_0$ terminates.

Here, we extend this result by giving a definition of tagged protocols for processes and showing that the clause generation algorithm yields clauses that satisfy the conditions of [21], so that the solving algorithm terminates. (A similar result has been proved for strong secrecy in the technical report [16].)

**Definition 15 (Tagged protocol)** A tagged protocol is a process $P_0$ together with a signature of constructors and destructors such that:

C1. The only constructors and destructors are those of Figure 2, plus $equal$.

C2. In every occurrence of $M(x)$ and $\overline{M}\langle N\rangle$ in $P_0$, $M$ is a name free in $P_0$.

C3. In every occurrence of $f(\ldots)$ with $f \in \{sencrypt, sencrypt_p, pencrypt_p, sign, nmrsign, h, mac\}$ in $P_0$, the first argument of $f$ is a tuple $(ct, M_1, \ldots, M_n)$, where the tag $ct$ is a constant. Different occurrences of $f$ have different values of the tag $ct$.

C4. In every occurrence of $let\ x = g(\ldots)\ in\ P\ else\ Q$, for $g \in \{sdecrypt, sdecrypt_p, pdecrypt_p, checksignature, getmessage\}$ in $P_0$, $P = let\ y = 1th_n(x)\ in\ if\ y = ct\ then\ P'$ for some $ct$ and $P'$.

In every occurrence of $nmrchecksign$ in $P_0$, its third argument is $(ct, M_1, \ldots, M_n)$ for some $ct, M_1, \ldots, M_n$.

C5. The destructor applications (including equality tests) have no $else$ branches. There exists a trace of $P_0$ (without adversary) in which all program points are executed exactly once.

C6. The second argument of $pencrypt_p$ in the trace of Condition C5 is of the form $pk(M)$ for some $M$.

C7. The arguments of $pk$ and $host$ in the trace of Condition C5 are atomic constants (free names or names created by restrictions not under inputs, non-deterministic destructor applications, or replications) and they are not tags.

Condition C1 limits the set of allowed constructors and destructors. We could give conditions on the form of allowed destructor rules, but these conditions are complex, so it is simpler and more intuitive to give an explicit list. Condition C2 states that all channels must be public. This condition avoids the need for the predicate $\mathrm{message}$. Condition C3 guarantees that tags are added in all messages, and Condition C4 guarantees that tags are always checked.

In most cases, the trace of Condition C5 is simply the intended execution of the protocol. All terms that occur in the trace of Condition C5 have pairwise distinct tags (since each program point is executed at most once, and tags at different program points are different by Condition C3). We can prove that it also guarantees that the terms of all clauses generated for the process $P_0$ have instances in the set of terms that occur in the trace of Condition C5 (using the fact that all program points are executed at least once). These properties are key in the termination proof. More concretely, Condition C5 means that, after removing replications of $P_0$, the resulting process has a trace that executes each program point (at least) once. In this trace, all destructor applications succeed and the process reduces to a configuration with an empty set of processes. Since, after removing replications, the number of traces of a process is always finite, Condition C5 is decidable.

Condition C6 means that, in its intended execution, the protocol uses public-key encryption only with public keys, and Condition C7 means that long-term secret (symmetric and asymmetric) keys are atomic constants.

**Example 14** A tagged protocol can easily be obtained by tagging the Needham-Schroeder-Lowe protocol. The tagged protocol consists of the following messages:

$$\begin{array}{lll} \text{Message 1.} & A \to B: & \{ct_0, a, pk_A\}_{pk_B} \\ \text{Message 2.} & B \to A: & \{ct_1, a, b, pk_B\}_{pk_A} \\ \text{Message 3.} & A \to B: & \{ct_2, b\}_{pk_B} \end{array}$$

Each encryption is tagged with a different tag $ct_0$, $ct_1$, and $ct_2$. This protocol can be represented in our calculus by the following process $P$:

$$
\begin{aligned}
P_A(sk_A, pk_A, pk_B) = {} & !c(x\_pk_B).(\nu a)\mathtt{event}(e_1(pk_A, x\_pk_B, a)).\\
& (\nu r_1)\overline{c}\langle pencrypt_p((ct_0, a, pk_A), x\_pk_B, r_1)\rangle.\\
& c(m).let\ (= ct_1, = a, x\_b, = x\_pk_B) = pdecrypt_p(m, sk_A)\ in\\
& \mathtt{event}(e_3(pk_A, x\_pk_B, a, x\_b)).(\nu r_3)\overline{c}\langle pencrypt_p((ct_2, x\_b), x\_pk_B, r_3)\rangle\\
& if\ x\_pk_B = pk_B\ then\ \mathtt{event}(e_A(pk_A, x\_pk_B, a, x\_b)).\\
& \overline{c}\langle sencrypt((ct_3, sAa), a)\rangle.\overline{c}\langle sencrypt((ct_4, sAb), x\_b)\rangle\\
P_B(sk_B, pk_B, pk_A) = {} & !c(m').let\ (= ct_1, x\_a, x\_pk_A) = pdecrypt_p(m, sk_B)\ in\\
& (\nu b)\mathtt{event}(e_2(x\_pk_A, pk_B, x\_a, b)).\\
& (\nu r_2)\overline{c}\langle pencrypt_p((ct_2, x\_a, b, pk_B), x\_pk_A, r_2)\rangle.\\
& c(m'').let\ (= ct_3, = b) = pdecrypt_p(m'', sk_B)\ in\\
& if\ x\_pk_A = pk_A\ then\ \mathtt{event}(e_B(x\_pk_A, pk_B, x\_a, b)).\\
& \overline{c}\langle sencrypt((ct_5, sBa), x\_a)\rangle.\overline{c}\langle sencrypt((ct_6, sBb), b)\rangle\\
P_T = {} & !c(x_1).c(x_2).\overline{c}\langle x_2\rangle.(c(x_3).c(x_4) \mid c(x_5).c(x_6))\\
P = {} & (\nu sk_A)(\nu sk_B)let\ pk_A = pk(sk_A)\ in\ let\ pk_B = pk(sk_B)\ in\\
& \overline{c}\langle pk_A\rangle\overline{c}\langle pk_B\rangle.(P_A(sk_A, pk_A, pk_B) \mid P_B(sk_B, pk_B, pk_A) \mid P_T)
\end{aligned}
$$

The encryptions that are used for testing the secrecy of nonces are also tagged, with tags $ct_3$ to $ct_6$. Furthermore, a process $P_T$ is added in order to satisfy Condition C5, because, without $P_T$, in the absence of adversary, the process would block when it tries to send the public keys $pk_A$ and $pk_B$. The execution of Condition C5 is the intended execution of the protocol. In this execution, the process $P_T$ receives the public keys $pk_A$ and $pk_B$; it forwards $pk_B$ on channel $c$ to $P_A$, so that a session between $A$ and $B$ starts. Then $A$ and $B$ run this session normally, and finally output the encryptions of $sAa$, $sAb$, $sBa$, and $sBb$; these encryptions are received by $P_T$. The other conditions of Definition 15 are easy to check, so $P$ is tagged.

Proposition 3 below applies to $P$, and also to the process without $P_T$, because the addition of $P_T$ in fact does not change the clauses. (The only clause generated from $P_T$ is a tautology, immediately removed by $elimtaut$.)

We prove the following termination result in [18, Appendix D]; we give a short proof sketch below.

**Proposition 3** *For* $\mathsf{sel} = \mathsf{sel}_0$, *the algorithm terminates on tagged protocols for queries of the form* $\alpha \rightsquigarrow \mathrm{false}$ *when* $\alpha$ *is closed and all facts in* $\mathcal{F}_{\mathrm{not}}$ *are closed.*

The proof first considers the particular case in which $pk$ and $host$ have a single argument in the execution of Condition C5, and then generalizes by mapping all arguments of $pk$ and $host$ (which are atomic constants by Condition C7) to a single constant. The proof of the particular case proceeds in two steps. The first step shows that the clauses generated from a tagged protocol satisfy the conditions of [21]. Basically, these conditions require that the clauses for the protocol satisfy the following properties:

T1. The patterns in the clauses are *tagged*, that is, the first argument of all occurrences of constructors except tuples, $pk$, and $host$ is of the form $(ct, M_1, \ldots, M_n)$. The proof of this property relies on Conditions C3 and C4.

T2. Let $S_1$ be the set of subterms of patterns that correspond to the terms that occur in the execution of Condition C5. Every clause has an instance in which all patterns are in $S_1$. The proof of this property relies on Condition C5.

T3. Each non-variable, non-data tagged pattern has at most one instance in $S_1$. (A pattern is said to be *non-data* when it is not of the form $f(\ldots)$ with $f$ a data constructor, that is, here, a tuple.) This property comes from Condition C3 which guarantees that the tags at distinct occurrences are distinct and, for $pk(p)$ and $host(p)$, from the hypothesis that $pk$ and $host$ have a single argument in the execution of Condition C5.

Note that the patterns in the clauses (Rf) and (Rg) that come from constructors and destructors are not tagged, so we need to handle them specially; Conditions C1 and C6 are useful for that.

The second step of the proof uses the result of [21] in order to conclude termination. Basically, this result shows that Properties T1 and T2 are preserved by resolution. The proof of this result relies on the fact that, if two non-variable non-data tagged patterns unify and have instances in $S_1$, then their instances in $S_1$ are equal (by T3). So, when unifying two such patterns, their unification still has an instance in $S_1$. Furthermore, we show that the size of the instance in $S_1$ of a clause obtained by resolution is not greater than the size of the instance in $S_1$ of one of the initial clauses. Hence, we can bound the size of the instance in $S_1$ of generated clauses, which shows that only finitely many clauses are generated.

The hypothesis that all facts in $\mathcal{F}_{\mathrm{not}}$ are closed is not really a restriction, since we can always remove facts from $\mathcal{F}_{\mathrm{not}}$ without changing the result. (It may just slow down the resolution.) The restriction to queries $\alpha \rightsquigarrow \mathrm{false}$ allows us to remove $\mathrm{m\text{-}event}$ facts from clauses (by Remark 3). For more general queries, $\mathrm{m\text{-}event}$ facts may occur in clauses, and one can find examples on which the algorithm does not terminate. Here is such an example:

$$
\begin{aligned}
P_S &= c'_1(y); \mathit{let}\ z = \mathit{sencrypt}((ct_0, y), k_{SB})\ \mathit{in} \\
&\quad \overline{c'_2}\langle \mathit{sencrypt}((ct_2, \mathit{sencrypt}((ct_1, z), k_{SA})), k_{SB})\rangle; \mathtt{event}(h((ct_3, y))); \overline{c'_3}\langle z\rangle \\
P_B &= c'_2(z'); c'_3(z); \mathit{let}\ (= ct_0, y) = \mathit{sdecrypt}(z, k_{SB})\ \mathit{in} \\
&\quad \mathit{let}\ (= ct_2, y') = \mathit{sdecrypt}(z', k_{SB})\ \mathit{in}\ \mathtt{event}(h((ct_4, y, y'))); \overline{c'_4}\langle y'\rangle \\
P_0 &= (\nu k_{SB}); (\overline{c'_1}\langle C_0\rangle \mid !P_S \mid !P_B \mid c'_4(y'))
\end{aligned}
$$

This example has been built on purpose for exhibiting non-termination, since we did not meet such non-termination cases in our experiments with real protocols. One can interpret this example as follows. The participant $A$ shares a key $k_{SA}$ with a server $S$. Similarly, $B$ shares a key $k_{SB}$ with $S$. The code of $S$ is represented by $P_S$, the code of $B$ by $P_B$, and $A$ is assumed to be dishonest, so it is represented by the adversary. The process $P_S$ builds two tickets $\mathit{sencrypt}((ct_0, y), k_{SB})$ and $\mathit{sencrypt}((ct_2,$

$sencrypt((ct_1, sencrypt((ct_0, y), k_{SB})), k_{SA})), k_{SB})$. The first ticket is for $B$, the second ticket should first be decrypted by $B$, then sent to $A$, which is going to decrypt it again and sent it back to $B$. In the example, $P_B$ just decrypts the two tickets and forwards the second one to $A$. It is easy to check that this process is a tagged protocol. This process generates the following clauses:

$$\begin{aligned}&\mathrm{attacker}(y) \Rightarrow \\ &\quad \mathrm{attacker}(sencrypt((ct_2, sencrypt((ct_1, sencrypt((ct_0, y), k_{SB})), k_{SA})), k_{SB}))\end{aligned} \tag{19}$$

$$\mathrm{attacker}(y) \wedge \text{m-event}(h((ct_3, y))) \Rightarrow \mathrm{attacker}(sencrypt((ct_0, y), k_{SB})) \tag{20}$$

$$\begin{aligned}&\mathrm{attacker}(sencrypt((ct_0, y), k_{SB})) \wedge \mathrm{attacker}(sencrypt((ct_2, y'), k_{SB})) \\ &\quad \wedge \text{m-event}(h((ct_4, y, y'))) \Rightarrow \mathrm{attacker}(y')\end{aligned} \tag{21}$$

$$\mathrm{attacker}(C_0) \tag{22}$$

The first two clauses come from $P_S$, the third one from $P_B$, and the last one from the output in $P_0$. Obviously, clauses (Init) (in particular $\mathrm{attacker}(k_{SA})$ since $k_{SA} \in fn(P_0)$), (Rf) for $sencrypt$ and $h$, and (Rg) for $sdecrypt$ are also generated. Assuming the first hypothesis is selected in (21), the solving algorithm performs a resolution step between (20) and (21), which yields:

$$\begin{aligned}&\mathrm{attacker}(y) \wedge \mathrm{attacker}(sencrypt((ct_2, y'), k_{SB})) \wedge \\ &\quad \text{m-event}(h((ct_3, y))) \wedge \text{m-event}(h((ct_4, y, y'))) \Rightarrow \mathrm{attacker}(y')\end{aligned}$$

The second hypothesis is selected in this clause. By resolving with (19), we obtain

$$\begin{aligned}&\mathrm{attacker}(y) \wedge \mathrm{attacker}(y') \wedge \text{m-event}(h((ct_3, y))) \wedge \\ &\quad \text{m-event}(h((ct_4, y, sencrypt((ct_1, sencrypt((ct_0, y'), k_{SB})), k_{SA})))) \\ &\quad\quad \Rightarrow \mathrm{attacker}(sencrypt((ct_1, sencrypt((ct_0, y'), k_{SB})), k_{SA}))\end{aligned}$$

By applying (Rg) for $sdecrypt$ and resolving with $\mathrm{attacker}(ct_1)$ and $\mathrm{attacker}(k_{SA})$, we obtain:

$$\begin{aligned}&\mathrm{attacker}(y) \wedge \mathrm{attacker}(y') \wedge \text{m-event}(h((ct_3, y))) \wedge \\ &\quad \text{m-event}(h((ct_4, y, sencrypt((ct_1, sencrypt((ct_0, y'), k_{SB})), k_{SA})))) \\ &\quad\quad \Rightarrow \mathrm{attacker}(sencrypt((ct_0, y'), k_{SB}))\end{aligned}$$

This clause is similar to (20), so we can repeat this resolution process, resolving with (21), (19), and decrypting the conclusion. Hence we obtain

$$\begin{aligned}&\bigwedge_{j=1}^{n} \mathrm{attacker}(y_j) \wedge \text{m-event}(h((ct_3, y_1))) \wedge \\ &\quad \bigwedge_{j=1}^{n-1} \text{m-event}(h((ct_4, y_j, sencrypt((ct_1, sencrypt((ct_0, y_{j+1}), k_{SB})), k_{SA})))) \\ &\quad\quad \Rightarrow \mathrm{attacker}(sencrypt((ct_0, y_n), k_{SB}))\end{aligned}$$

for all $n > 0$, so the algorithm does not terminate.

As noticed in [21], termination could be obtained in the presence of m-event facts with an additional simplification:

Elimination of useless $\mathrm{m\text{-}event}$ facts: *elim-m-event* eliminates $\mathrm{m\text{-}event}$ facts in which a variable $x$ occurs, and $x$ only occurs in $\mathrm{m\text{-}event}$ facts and in $\mathrm{attacker}(x)$ hypotheses.

This simplification is always sound, because it creates a stronger clause. It does not lead to a loss of precision when all variables of events after $\rightsquigarrow$ also occur in the event before $\rightsquigarrow$. (This happens in particular for non-injective agreement.) Indeed, assume that $\mathrm{m\text{-}event}(p)$ contains a variable which does not occur in the conclusion. This is preserved by resolution, so when we obtain a clause $\mathrm{m\text{-}event}(p') \wedge H \Rightarrow \mathrm{event}(p'')$, where $\mathrm{m\text{-}event}(p')$ comes from $\mathrm{m\text{-}event}(p)$, $p'$ contains a variable that does not occur in $p''$, so this occurrence of $\mathrm{m\text{-}event}(p')$ cannot be used to prove the desired correspondence. However, in the general case, this simplification leads to a loss of precision. (It may miss some $\mathrm{m\text{-}event}$ facts.) That is why this optimization was present in early implementations which verified only authentication, and was later abandoned. We could reintroduce it when all variables of events after $\rightsquigarrow$ also occur in the event before $\rightsquigarrow$, if we had termination problems coming from $\mathrm{m\text{-}event}$ facts for practical examples. No such problems have occurred up to now.

## 8.2 Choice of the Selection Function

Unfortunately, not all protocols are tagged. In particular, protocols using a Diffie-Hellman key agreement (see Section 9.1) are not tagged in the sense of Definition 15. The algorithm still terminates for some of them (Skeme [53] for secrecy, SSH) with the previous selection function $\mathsf{sel}_0$. However, it does not terminate with the selection function $\mathsf{sel}_0$ for some other examples (Skeme [53] for one authentication property, the Needham-Schroeder shared-key protocol [61], some versions of the Woo-Lam shared-key protocol [71] and [5, Example 6.2].) In this section, we present heuristics to improve the choice of the selection function, in order to avoid most simple non-termination cases. As reported in more detail in Section 10, these heuristics provide termination for Skeme [53] and the Needham-Schroeder shared-key protocol [61].

Let us determine which constraints the selection function should satisfy to avoid loops in the algorithm. First, assume that there is a clause $H \wedge F \Rightarrow \sigma F$, where $\sigma$ is a substitution such that all $\sigma^n F$ are distinct for $n \in \mathbb{N}$.

- Assume that $F$ is selected in this clause, and there is a clause $H' \Rightarrow F'$, where $F'$ unifies with $F$, and the conclusion is selected in $H' \Rightarrow F'$. Let $\sigma'$ be the most general unifier of $F$ and $F'$. So the algorithm generates:

$$\sigma' H' \wedge \sigma' H \Rightarrow \sigma' \sigma F \quad \ldots \quad \sigma' H' \wedge \bigwedge_{i=0}^{n-1} \sigma' \sigma^i H \Rightarrow \sigma' \sigma^n F$$

  assuming that the conclusion is selected in all these clauses, and that no clause is removed because it is subsumed by another clause. So the algorithm would not terminate. Therefore, in order to avoid this situation, we should avoid selecting $F$ in the clause $H \wedge F \Rightarrow \sigma F$.

- Assume that the conclusion is selected in the clause $H \wedge F \Rightarrow \sigma F$, and there is a clause $H' \wedge \sigma' F \Rightarrow C$ (up to renaming of variables), where $\sigma'$ commutes with $\sigma$ (in particular, when $\sigma$ and $\sigma'$ have disjoint supports), and that $\sigma' F$ is selected in this clause. So the algorithm generates:

$$\sigma' H \wedge \sigma H' \wedge \sigma' F \Rightarrow \sigma C \quad \ldots \quad \bigwedge_{i=0}^{n-1} \sigma' \sigma^i H \wedge \sigma^n H' \wedge \sigma' F \Rightarrow \sigma^n C$$

  assuming that $\sigma' F$ is selected in all these clauses, and that no clause is removed because it is subsumed by another clause. So the algorithm would not terminate. Therefore, in order to avoid this situation, if the conclusion is selected in the clause $H \wedge F \Rightarrow \sigma F$, we should avoid selecting facts of the form $\sigma' F$, where $\sigma'$ and $\sigma$ have disjoint supports, in other clauses.

In particular, since there are clauses of the form $\mathrm{attacker}(x_1) \wedge \ldots \wedge \mathrm{attacker}(x_n) \Rightarrow \mathrm{attacker}(f(x_1, \ldots, x_n))$, by the first remark, the facts $\mathrm{attacker}(x_i)$ should not be selected in this clause. So the conclusion will be selected in this clause and, by the second remark, facts of the form $\mathrm{attacker}(x)$ with $x$ variable should not be selected in other clauses. We find again the constraint used in the definition of $\mathsf{sel}_0$.

We also have the following similar remarks after swapping conclusion and hypothesis. Assume that there is a clause $H \wedge \sigma F \Rightarrow F$, where $\sigma$ is a substitution such that all $\sigma^n F$ are distinct for $n \in \mathbb{N}$. We should avoid selecting the conclusion in this clause and, if we select $\sigma F$ in this clause, we should avoid selecting conclusions of the form $\sigma' F$, where $\sigma'$ and $\sigma$ have disjoint supports, in other clauses.

We define a selection function that takes into account all these remarks. For a clause $H \Rightarrow C$, we define the weight $w_{\mathrm{hyp}}(F)$ of a fact $F \in H$ by:

$$w_{\mathrm{hyp}}(F) = \begin{cases} -\infty & \text{if } F \text{ is an unselectable fact} \\ -2 & \text{if } \exists \sigma, \sigma F = C \\ -1 & \text{otherwise, if } F \in S_{\mathrm{hyp}} \\ 0 & \text{otherwise.} \end{cases}$$

The set $S_{\mathrm{hyp}}$ is defined as follows: at the beginning, $S_{\mathrm{hyp}} = \emptyset$; if we generate a clause $H \wedge F \Rightarrow \sigma F$ where $\sigma$ is a substitution that maps variables of $F$ to terms that are not all variables and, in this clause, we select the conclusion, then we add to $S_{\mathrm{hyp}}$ all facts $\sigma' F$ with $\sigma$ and $\sigma'$ of disjoint support (and renamings of these facts). For simplicity, we have replaced the condition "all $\sigma^n F$ are distinct for $n \in \mathbb{N}$" with "$\sigma$ maps variables of $F$ to terms that are not all variables". (The former implies the latter but the converse is wrong.) Our aim is only to obtain good heuristics, since there exists no perfect selection function that would provide termination in all cases. The set $S_{\mathrm{hyp}}$ can easily be represented finitely: just store the facts $F$ with, for each variable, a flag indicating whether this variable can be substituted by any term by $\sigma'$, or only by a variable.

Similarly, we define the weight of the conclusion:

$$w_{\mathrm{concl}} = \begin{cases} -2 & \text{if } \exists \sigma, \exists F \in H, \sigma C = F \\ -1 & \text{otherwise, if } C \in S_{\mathrm{concl}} \\ 0 & \text{otherwise.} \end{cases}$$

The set $S_{\mathrm{concl}}$ is defined as follows: at the beginning, $S_{\mathrm{concl}} = \emptyset$; if we generate a clause $H \wedge \sigma F \Rightarrow F$ where $\sigma$ is a substitution that maps variables of $F$ to terms that are not all variables and, in this clause, we select $\sigma F$, then we add to $S_{\mathrm{concl}}$ all facts $\sigma' F$ with $\sigma$ and $\sigma'$ of disjoint support (and renamings of these facts).

Finally, we define

$$\mathsf{sel}_1(H \Rightarrow C) = \begin{cases} \emptyset & \text{if } \forall F \in H, w_{\mathrm{hyp}}(F) < w_{\mathrm{concl}}, \\ \{F_0\} & \text{where } F_0 \in H \text{ of maximum weight, otherwise.} \end{cases}$$

Therefore, we avoid unifying facts of smallest weight when that is possible. The selected fact $F_0$ can be any element of $H$ of maximum weight. In the implementation, the hypotheses are represented by a list, and the selected fact is the first element of the list of hypotheses of maximum weight.

We can also notice that the bigger the fact is, the stronger are constraints to unify it with another fact. So selecting a bigger fact should reduce the possible unifications. Therefore, we consider $\mathsf{sel}_2$, defined as $\mathsf{sel}_1$ except that $w_{\mathrm{hyp}}(F) = size(F)$ instead of $0$ in the last case.

When selecting a fact that has a negative weight, we are in one of the cases when termination will probably not be achieved. We therefore emit a warning in this case, so that the user can stop the program.

# 9 Extensions

In this section, we briefly sketch a few extensions to the framework presented previously. The extensions of Sections 9.1, 9.2, and 9.3 were presented in [19] for the proof of process equivalences. We sketch here how to adapt them to the proof of correspondences.

## 9.1 Equational Theories and Diffie-Hellman Key Agreements

Up to now, we have defined cryptographic primitives by associating rewrite rules to destructors. Another way of defining primitives is by equational theories, as in the applied pi calculus [4]. This allows us to model, for instance, variants of encryption for which the failure of decryption cannot be detected or more complex primitives such as Diffie-Hellman key agreements. The Diffie-Hellman key agreement [39] enables two principals to build a shared secret. It is used as an elementary step in more complex protocols, such as Skeme [53], SSH, SSL, and IPsec.

As shown in [19], our verifier can be extended to handle some equational theories. Basically, one shows that each trace in a model with an equational theory corresponds to a trace in a model in which function symbols are equipped with additional rewrite rules, and conversely. (We could adapt [19, Lemma 1] to show that this result also applies to correspondences.) Therefore, we can show that a correspondence proved in the model with rewrite rules implies the same correspondence in the model with an equational theory. Moreover, we have implemented algorithms that compute the rewrite rules from an equational theory.

In the experiments reported in this paper, we use equational theories only for the Diffie-Hellman key agreement, which can be modeled by using two functions $f$ and $f'$ that satisfy the equation

$$f(y, f'(x)) = f(x, f'(y)). \tag{23}$$

In practice, the functions are $f(x, y) = y^x \mod p$ and $f'(x) = b^x \mod p$, where $p$ is prime and $b$ is a generator of $\mathbb{Z}_p^*$. The equation $f(y, f'(x)) = (b^x)^y \mod p = (b^y)^x \mod p = f(x, f'(y))$ is satisfied. In our verifier, following the ideas used in the applied pi calculus [4], we do not consider the underlying number theory; we work abstractly with the equation (23). The Diffie-Hellman key agreement involves two principals $A$ and $B$. $A$ chooses a random name $x_0$, and sends $f'(x_0)$ to $B$. Similarly, $B$ chooses a random name $x_1$, and sends $f'(x_1)$ to $A$. Then $A$ computes $f(x_0, f'(x_1))$ and $B$ computes $f(x_1, f'(x_0))$. Both values are equal by (23), and they are secret: assuming that the attacker cannot have $x_0$ or $x_1$, it can compute neither $f(x_0, f'(x_1))$ nor $f(x_1, f'(x_0))$.

In our verifier, the equation (23) is translated into the rewrite rules

$$f(y, f'(x)) \to f(x, f'(y)) \quad f(x, y) \to f(x, y).$$

Notice that this definition of $f$ is non-deterministic: a term such as $f(a, f'(b))$ can be reduced to $f(b, f'(a))$ and $f(a, f'(b))$, so that $f(a, f'(b))$ reduces to its two forms modulo the equational theory. The fact that these rewrite rules model the equation (23) correctly follows from [19, Section 5].

When using this model, we have to adapt the verification of correspondences. Indeed, the conditions on the clauses must be checked *modulo the equational theory*. (Using the rewrite rules, we can implement unification modulo the equational theory, basically by rewriting the terms by the rewrite rules before performing syntactic unification.) For example, in the case of non-injective agreement, even if the process $P_0$ satisfies non-injective agreement against $Init$-adversaries, it may happen that a clause $\text{m-event}(e'(p_1, \ldots, p_n)\{f(p_2, f'(p_1))/z\}) \Rightarrow \text{event}(e(p_1, \ldots, p_n)\{f(p_1, f'(p_2))/z\})$ is in $\mathsf{solve}_{P_0', Init}(\text{event}(e(x_1, \ldots, x_n)))$. The specification is still satisfied in this case, because $(p_1, \ldots, p_n)\{f(p_1, f'(p_2))/z\} = (p_1, \ldots, p_n)\{f(p_2, f'(p_1))/z\}$ modulo the equational theory. So we have to test that, if $H \Rightarrow \text{event}(e(p_1, \ldots, p_n))$ is in $\mathsf{solve}_{P_0', Init}(\text{event}(e(x_1, \ldots, x_n)))$, then there exist $p_1', \ldots, p_n'$ equal to $p_1, \ldots, p_n$ modulo the equational theory such that $\text{m-event}(e'(p_1', \ldots, p_n')) \in H$. More generally, the equality $R = H \wedge \text{m-event}(\sigma' p_{j1}) \wedge \ldots \wedge \text{m-event}(\sigma' p_{jl_j}) \Rightarrow \text{event}(\sigma' p_j')$ in the hypothesis of Theorem 3 is checked modulo the equational theory (using matching modulo the equational theory to find $\sigma'$). Point V2.1 of the definition of verify and Hypothesis H2 of Theorem 5 are also checked modulo the equational theory. Furthermore, the following condition is added to Point V2.2 of the definition of verify:

> For all $j$, $r$, and $k$, we let $q_c = \sigma_{jr} q_{jk}$ and $p_c = \sigma_{jr} p_{jk}$, and we require that, for all substitutions $\sigma$ and $\sigma'$, if $\sigma p_c = \sigma' p_c$ and for all $x \in fv(q_c) \setminus fv(p_c)$, $\sigma x = \sigma' x$, then $\sigma q_c = \sigma' q_c$ (where equalities are considered modulo the equational theory).

This property is useful in the proof of Theorem 5 (see [18, Appendix E]). It always holds when the equational theory is empty, because $\sigma p_c = \sigma' p_c$ implies that for all

$x \in fv(p_c)$, $\sigma x = \sigma' x$, so for all $x \in fv(q_c)$, $\sigma x = \sigma' x$. However, it does not hold in general for any equational theory, so we need to check it explicitly when the equational theory is non-empty. In the implementation, this condition is checked as follows. Let $\theta$ be a renaming of variables of $p_c$ to fresh variables. We check that, for every $\sigma_u$ most general unifier of $p_c$ and $\theta p_c$ modulo the equational theory, $\sigma_u q_c = \sigma_u \theta q_c$ modulo the equational theory. When this check succeeds, we can prove the condition above as follows. Let $\sigma_0$ be defined by, for all $x \in fv(q_c)$, $\sigma_0 x = \sigma x$ and, for all $x \in fv(\theta p_c)$, $\sigma_0 x = \sigma' \theta^{-1} x$. If $\sigma p_c = \sigma' p_c$, then $\sigma_0 p_c = \sigma p_c = \sigma' p_c = \sigma_0 \theta p_c$, so $\sigma_0$ unifies $p_c$ and $\theta p_c$, hence there exist $\sigma_1$ and a most general unifier $\sigma_u$ of $p_c$ and $\theta p_c$ such that $\sigma_0 = \sigma_1 \sigma_u$. We have $\sigma_u q_c = \sigma_u \theta q_c$, so $\sigma q_c = \sigma_0 q_c = \sigma_1 \sigma_u q_c = \sigma_1 \sigma_u \theta q_c = \sigma_0 \theta q_c = \sigma' q_c$.

This treatment of equations has the advantage that resolution can still use syntactic unification, so it remains efficient. However, it also has limitations; for example, it cannot handle associative functions, such as XOR, because it would generate an infinite number of rewrite rules for the destructors. We refer to [29, 32] for treatments of XOR and to [28, 49, 57, 59] for treatments of Diffie-Hellman key agreements with more detailed algebraic relations. The NRL protocol analyzer handles a limited version of associativity for strings of bounded length [44], which we could handle.

## 9.2 Precise Treatment of $else$ Branches

In the generation of clauses described in Section 5.2, we consider that the $else$ branch of destructor applications may always be executed. Our implementation takes into account these $else$ branches more precisely. In order to do that, it uses a set of special variables $GVar$ and a predicate $\mathrm{nounif}$, also used in [19], such that, for all closed patterns $p$ and $p'$, $\mathrm{nounif}(p, p')$ holds if and only if there is no closed substitution $\sigma$ with domain $GVar$ such that $\sigma p = \sigma p'$. The fact $\mathrm{nounif}(p, p')$ means that $p \neq p'$ for all values of the special variables in $GVar$.

One can then check the failure of an equality test $M = M'$ by $\mathrm{nounif}(\rho(M), \rho(M'))$ and the failure of a destructor application $g(M_1, \ldots, M_n)$ by $\bigwedge_{g(p_1,\ldots,p_n)\to p \in \mathrm{def}(g)} \mathrm{nounif}((\rho(M_1), \ldots, \rho(M_n)), GVar(p_1, \ldots, p_n))$, where $GVar(p)$ is the pattern $p$ after renaming all its variables to elements of $GVar$ and $\rho$ is the environment that maps variables to their corresponding patterns. Intuitively, the rewrite rule $g(p_1, \ldots, p_n) \to p$ can be applied if and only if $(\rho(M_1), \ldots, \rho(M_n))$ is an instance of $(p_1, \ldots, p_n)$. So the rewrite rule $g(p_1, \ldots, p_n) \to p$ cannot be applied if and only if $\mathrm{nounif}((\rho(M_1), \ldots, \rho(M_n)), GVar(p_1, \ldots, p_n))$.

The predicate $\mathrm{nounif}$ is handled by specific simplification steps in the solver, described and proved correct in [19].

## 9.3 Scenarios with Several Stages

Some protocols can be broken into several parts, or stages, numbered 0, 1, ..., such that when the protocol starts, stage 0 is executed; at some point in time, stage 0 stops and stage 1 starts; later, stage 1 stops and stage 2 starts, and so on. Therefore, stages allow us to model a global clock. Our verifier can be extended to such scenarios with several stages, as summarized in [19]. We add a construct $t : P$ to the syntax of processes, which means that process $P$ runs only in stage $t$, where $t$ is an integer.

The generation of clauses can easily be extended to processes with stages. We use predicates $\mathrm{attacker}_t$ and $\mathrm{message}_t$ for each stage $t$, generate the clauses for the attacker for each stage, and the clauses for the protocol with predicates $\mathrm{attacker}_t$ and $\mathrm{message}_t$ for each process that runs in stage $t$. Furthermore, we add clauses

$$\mathrm{attacker}_t(x) \Rightarrow \mathrm{attacker}_{t+1}(x) \qquad \text{(Rt)}$$

in order to transmit attacker knowledge from each stage $t$ to the next stage $t+1$.

Scenarios with several stages allow us to model properties related to the compromise of keys. For example, we can model forward secrecy properties as follows. Consider a public-key protocol $P$ (without stage prefix) and the process $P' = 0 : P \mid 1 : \overline{c}\langle sk_A\rangle; \overline{c}\langle sk_B\rangle$, which runs $P$ in stage 0 and later outputs the secret keys of $A$ and $B$ on the public channel $c$ in stage 1. If we prove that $P'$ preserves the secrecy of the session keys of $P$, then the attacker cannot obtain these session keys even if it later compromises the private keys of $A$ and $B$, which is forward secrecy.

## 9.4 Compromise of Session Keys

We consider the situation in which the attacker compromises some session keys of the protocol. Our goal is then to show that the other session keys of the protocol are still safe. For example, this property does not hold for the Needham-Schroeder shared-key protocol [61]: in this protocol, when an attacker manages to get some session keys, then it can also get the secrets of other sessions.

If we assume that the compromised sessions are all run before the standard sessions (to model that the adversary needs time to break the session keys before being able to use the obtained information against standard sessions), then this can be modeled as a scenario with two stages: in stage 0, the process runs a modified version of the protocol that outputs its session keys; in stage 1, the standard sessions runs; we prove the security of the sessions of stage 1.

However, we can also consider a stronger model, in which the compromised sessions may run in parallel with the non-compromised ones. In this case, we have a single stage.

Let $P_0$ be the process representing the whole protocol. We consider that the part of $P_0$ not under replications corresponds to the creation of long-term secrets, and the part of $P_0$ under at least one replication corresponds to the sessions. We say that the names generated under at least one replication in $P_0$ are *session names*. We add one argument $i_c$ to the function symbols $a[\ldots]$ that encode session names in the instrumented process $P_0'$; this additional argument is named *compromise identifier* and can take two values, $s_0$ or $s_1$. We consider that, during the execution of the protocol, each replicated subprocess $!Q_X$ of $P_0$ generates two sets of copies of $Q_X$, one with compromise identifier $s_0$, one with $s_1$. The attacker compromises sessions that involve only copies of processes $Q_X$ with the compromise identifier $s_0$. It does not compromise sessions that involve at least one copy of some process $Q_X$ with compromise identifier $s_1$.

The clauses for the process $P_0$ are generated as in Section 5.2 (except for the addition of a variable compromise identifier as argument of session names). The following

clauses are added:

For each constructor $f$, $\mathrm{comp}(x_1) \wedge \ldots \wedge \mathrm{comp}(x_k) \Rightarrow \mathrm{comp}(f(x_1, \ldots, x_k))$

For each $(\nu a : a[\ldots])$ under $n$ replications and $k$ inputs and non-deterministic destructor applications in $P'_0$,

$\mathrm{comp}(x_1) \wedge \ldots \wedge \mathrm{comp}(x_k) \Rightarrow \mathrm{comp}(a[x_1, \ldots, x_k])$ if $n = 0$

$\mathrm{comp}(x_1) \wedge \ldots \wedge \mathrm{comp}(x_k) \Rightarrow \mathrm{comp}(a[x_1, \ldots, x_k, i_1, \ldots, i_n, s_0])$ if $n > 0$

$\mathrm{comp}(x_1) \wedge \ldots \wedge \mathrm{comp}(x_k) \Rightarrow \mathrm{attacker}(a[x_1, \ldots, x_k, i_1, \ldots, i_n, s_0])$ if $n > 0$

The predicate $\mathrm{comp}$ is such that $\mathrm{comp}(p)$ is true when all session names in $p$ have compromise identifier $s_0$. These clauses express that the attacker has the session names that contain only the compromise identifier $s_0$.

In order to prove the secrecy of a session name $s$, we query the fact $\mathrm{attacker}(s[x_1, \ldots, x_k, i_1, \ldots, i_n, s_1])$. If this fact is underivable, then the protocol does not have the weakness of the Needham-Schroeder shared-key protocol mentioned above: the attacker cannot have the secret $s$ of a session that it has not compromised. In contrast, $\mathrm{attacker}(s[x_1, \ldots, x_k, i_1, \ldots, i_n, s_0])$ is always derivable, since the attacker has compromised the sessions with identifier $s_0$.

We can also prove correspondences in the presence of key compromise. We want to prove that the non-compromised sessions are secure, so we prove that, if an event $\mathtt{event}(M)$ has been executed in a copy of some $Q_X$ with compromise identifier $s_1$, then the required events $\mathtt{event}(M_{\overline{jk}})$ have been executed in any process. (A copy of $Q_X$ with compromise identifier $s_1$ may interact with a copy of $Q_Y$ with compromise identifier $s_0$ and, in this case, the events $\mathtt{event}(M_{\overline{jk}})$ may be executed in the copy of $Q_Y$ with compromise identifier $s_0$.) We obtain this result by adding the compromise identifier $i_c$ as argument of the predicates $\mathrm{m\text{-}event}$ and $\mathrm{event}$ in clauses, and correspondingly adding $s_1$ as argument of $\mathtt{event}(M)$ and $\mathtt{event}(M_j)$, and a fresh variable as argument of the other events $\mathtt{event}(M_{\overline{jk}})$ in queries. We can then prove the correspondence in the same way as in the absence of key compromise. The treatment of correspondences $\mathrm{attacker}(M) \rightsquigarrow \ldots$ and $\mathrm{message}(M, M') \rightsquigarrow \ldots$ in which $M$ and $M'$ do not contain bound names remains unchanged.

# 10 Experimental Results

We have implemented our verifier in Ocaml and have performed tests on various protocols of the literature. The tests reported here concern secrecy and authentication properties for simple examples of protocols. More complex examples have been studied, using our technique for proving correspondences. We do not detail them in this paper, because they have been the subject of specific papers [2, 3, 20].

Our results are summarized in Figure 6, with references to the papers that describe the protocols and the attacks. In these tests, the protocols are fully modeled, including interaction with the server for all versions of the Needham-Schroeder, Woo-Lam shared key, Denning-Sacco, Otway-Rees, and Yahalom protocols. The first column indicates the name of the protocol; we use the following abbreviations: NS for Needham-Schroeder, PK for public-key, SK for shared-key, corr. for corrected, tag. for tagged,

unid. for unidirectional, and bid. for bidirectional. We have tested the Needham-Schroeder shared key protocol with the modeling of key compromise mentioned in Section 9.4, in which the compromised sessions can be executed in parallel with the non-compromised ones (version marked "comp." in Figure 6). The second column indicates the number of Horn clauses that represent the protocol. The third column indicates the total number of resolution steps performed for analyzing the protocol.

The fourth column gives the execution time of our analyzer, in ms, on a Pentium M 1.8 GHz. Several secrecy and agreement specifications are checked for each protocol. The time given is the total time needed to check all specifications. The following factors influence the speed of the system:

- We use secrecy assumptions to speed up the search. These assumptions say that the secret keys of the principals, and the random values of the Diffie-Hellman key agreement in the Skeme protocol, remain secret. On average, the verifier is two times slower without secrecy assumptions, in our tests.

- We mentioned several selection functions, and the speed of the system can vary substantially depending on the selection function. In the tests of Figure 6, we used the selection function $\mathsf{sel}_2$. With $\mathsf{sel}_1$, the system is two times slower on average on Needham-Schroeder shared-key, Otway-Rees, the variant of [64] of Otway-Rees, and Skeme but faster on the bidirectional simplified Yahalom (59 ms instead of 91 ms). The speed is almost unchanged for our other tests. On average, the verifier is 1.8 times slower with $\mathsf{sel}_1$ than with $\mathsf{sel}_2$, in our tests.

  The selection function $\mathsf{sel}_0$ gives approximately the same speed as $\mathsf{sel}_1$, except for Skeme, for which the analysis does not terminate with $\mathsf{sel}_0$. (We comment further on termination below.)

- The tests of Figure 6 have been performed without elimination of redundant hypotheses. With elimination of redundant hypotheses that contain $\mathrm{m}\text{-}\mathrm{event}$ facts, we obtain approximately the same speed. With elimination of all redundant hypotheses, the verifier is 1.3 times slower on average in these tests, because of the time spent testing whether hypotheses are redundant.

When our tool successfully proves that a protocol satisfies a certain specification, we are sure that this specification indeed holds, by our soundness theorems. When our tool does not manage to prove that a protocol satisfies a certain specification, it finds at least one clause and a derivation of this clause that contradicts the specification. The existence of such a clause does not prove that there is an attack: it may correspond to a false attack, due to the approximations introduced by the Horn clause model. However, using an extension of the technique of [6] to events, in most cases, our tool reconstructs a trace of the protocol, and thus proves that there is actually an attack against the considered specification. In the tests of Figure 6, this reconstruction succeeds in all cases for secrecy and non-injective correspondences, in the absence of key compromise. The trace reconstruction is not implemented yet in the presence of key compromise (Section 9.4) or for injective correspondences. (It presents additional difficulties in the latter case, since the trace should execute some event twice and others once in order to contradict injectivity, while the derivation corresponds to the execution

| Protocol | # cl. | # res. steps | Time (ms) | Cases with attacks Secrecy | $A$ | $B$ | Ref. |
|---|---|---|---|---|---|---|---|
| NS PK [61] | 32 | 1988 | 95 | Nonces $B$ | None | All | [54] |
| NS PK corr. [54] | 36 | 1481 | 51 | None | None | None | |
| Woo-Lam PK [71] | 23 | 104 | 7 | | | All | [41] |
| Woo-Lam PK corr. [73] | 27 | 156 | 6 | | | None | |
| Woo-Lam SK [47] | 25 | 184 | 8 | | | All | [8] |
| Woo-Lam SK corr. [47] | 21 | 244 | 4 | | | None | |
| Denning-Sacco [38] | 30 | 440 | 18 | Key $B$ | | All | [5] |
| Denning-Sacco corr. [5] | 30 | 438 | 16 | None | | Inj | |
| NS SK [61], tag. | 31 | 2721 | 41 | None | None | None | |
| NS SK corr. [62], tag. | 32 | 2102 | 57 | None | None | None | |
| NS SK [61], tag., comp. | 50 | 25241 | 167 | Key $B$ | None | Inj | [38] |
| NS SK corr. [62], tag., comp. | 53 | 23956 | 225 | None | None | None | |
| Yahalom [27] | 26 | 1515 | 34 | None | Key | None | |
| Simpler Yahalom [27], unid. | 21 | 1479 | 30 | None | Key | None | |
| Simpler Yahalom [27], bid. | 24 | 3685 | 91 | None | All | None | [68] |
| Otway-Rees [63] | 34 | 1878 | 59 | None | Key | Inj,Key | [27] |
| Simpler Otway-Rees [5] | 28 | 1934 | 31 | None | All | All | [64] |
| Otway-Rees, variant of [64] | 35 | 3349 | 87 | Key $B$ | All | All | [64] |
| Main mode of Skeme [53] | 39 | 4139 | 154 | None | None | None | |

Figure 6: Experimental results

of events once, with badly related session identifiers.) In the cases in which trace reconstruction is not implemented, we have checked manually that the protocol is indeed subject to an attack, so our tool found no false attack in the tests of Figure 6: for all specifications that hold, it has proved them.

The last four columns give the results of the analysis. The column "Secrecy" concerns secrecy properties, the column $A$ concerns agreement specifications $\mathrm{event}(e(x_1, \ldots, x_n)) \rightsquigarrow [\mathrm{inj}]\ \mathrm{event}(e'(x_1, \ldots, x_n))$ in which $A$ executes the event $\mathtt{event}(e(M_1, \ldots, M_n))$, the column $B$ agreement specifications $\mathrm{event}(e(x_1, \ldots, x_n)) \rightsquigarrow [\mathrm{inj}]\ \mathrm{event}(e'(x_1, \ldots, x_n))$ in which $B$ executes the event $\mathtt{event}(e(M_1, \ldots, M_n))$. The last column gives the reference of the attacks when attacks are found. The first six protocols of Figure 6 (Needham-Schroeder public key and Woo-Lam one-way authentication protocols) are authentication protocols. For them, we have tested non-injective and recent injective agreement on the name of the participants, and non-injective and injective full agreement (agreement on all atomic data). For the Needham-Schroeder public key protocol, we have also tested the secrecy of nonces. "Nonces $B$" means that the nonces $N_a$ and $N_b$ manipulated by $B$ may not be secret, "None" means all tested specifications are satisfied (there is no attack), "All" that our tool finds an attack against all tested specifications. The Woo and Lam protocols are *one-way* authentication protocols: they are intended to authenticate $A$ to $B$, but not $B$ to $A$, so we have only tested them with $B$ containing $\mathtt{event}(e(M_1, \ldots, M_n))$.

Numerous versions of the Woo and Lam shared-key protocol have been published

in the literature [71], [8], [5, end of Example 3.2], [5, Example 6.2], [73], [47] (flawed and corrected versions). Our tool terminates and proves the correctness of the corrected versions of [8] and of [47]; it terminates and finds an attack on the flawed version of [47]. (The messages received or sent by $A$ do not depend on the host $A$ wants to talk to, so $A$ may start a session with the adversary $C$, and the adversary can reuse the messages of this session to talk to $B$ in $A$'s name.) We can easily see that the versions of [71] and [5, Example 6.2] are also subject to this attack, even if our tool does not terminate on them. The only difference between the protocol of [47] and that of [71] is that [47] adds tags to distinguish different encryption sites. As shown in Section 8.1, adding tags enforces termination. Our tool finds the attack of [30, bottom of page 52] on the versions of [5, end of Example 3.2] and [73]. For example, the version of [73] is

$$\begin{array}{lll}
\text{Message 1.} & A \rightarrow B: & A \\
\text{Message 2.} & B \rightarrow A: & N_B \\
\text{Message 3.} & A \rightarrow B: & \{A, B, N_B\}_{K_{AS}} \\
\text{Message 4.} & B \rightarrow S: & \{A, B, \{A, B, N_B\}_{K_{AS}}\}_{K_{BS}} \\
\text{Message 5.} & S \rightarrow B: & \{A, B, N_B\}_{K_{BS}}
\end{array}$$

and the attack is

$$\begin{array}{lll}
\text{Message 1.} & I(A) \rightarrow B: & A \\
\text{Message 2.} & B \rightarrow I(A): & N_B \\
\text{Message 3.} & I(A) \rightarrow B: & N_B \\
\text{Message 4.} & B \rightarrow I(A): & \{A, B, N_B\}_{K_{BS}} \\
\text{Message 5.} & I(A) \rightarrow B: & \{A, B, N_B\}_{K_{BS}}
\end{array}$$

In message 3, the adversary sends $N_B$ instead of $\{A, B, N_B\}_{K_{AS}}$. $B$ cannot see the difference and, acting as defined in the protocol, $B$ unfortunately sends exactly the message needed by the adversary as message 5. So $B$ thinks he talks to $A$, while $A$ and $S$ can perfectly be dead. The attack found against the version of [5, end of Example 3.2] is very similar.

The last five protocols exchange a session key, so we have tested agreement on the names of the participants, and agreement on both the participants and the session key (instead of full agreement, since agreement on the session key is more important than agreement on other values). In Figure 6, "Key $B$" means that the key obtained by $B$ may not be secret, "Key" means that agreement on the session key is wrong, "Inj" means that injective agreement is wrong, "All" and "None" are as before.

In the Needham-Schroeder shared key protocol [61], the last messages are

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

where $N_B$ is a nonce. Representing $N_B - 1$ with a function $\mathsf{minusone}(x) = x - 1$, with associated destructor $\mathsf{plusone}$ defined by $\mathsf{plusone}(\mathsf{minusone}(x)) \rightarrow x$, the algorithm does not terminate with the selection function $\mathsf{sel}_0$. The selection functions $\mathsf{sel}_1$ or $\mathsf{sel}_2$ given in Section 8.2 however yield termination. We can also notice that the purpose of the subtraction is to distinguish the reply of $A$ from $B$'s message. As mentioned in [5], it would be clearer to have:

Message 4. $B \rightarrow A$: $\{\text{Message } 4 : N_B\}_K$
Message 5. $A \rightarrow B$: $\{\text{Message } 5 : N_B\}_K$

We have used this encoding in the tests shown in Figure 6. Our tool then terminates with selection functions $\mathsf{sel}_0$, $\mathsf{sel}_1$, and $\mathsf{sel}_2$. [21] explains in more detail why these two messages encoded with minusone prevent termination with $\mathsf{sel}_0$, and why the addition of tags "Message 4", "Message 5" yields termination. Adding the tags may strengthen the protocol (for instance, in the Needham-Schroeder shared key protocol, it prevents replaying Message 5 as a Message 4), so the security of the tagged version does not imply the security of the original version. As mentioned in [5], using the tagged version is a better design choice because it prevents confusing different messages, so this version should be implemented. Our tool also does not terminate on Skeme with selection function $\mathsf{sel}_0$, for an authentication query, but terminates with selection functions $\mathsf{sel}_1$ or $\mathsf{sel}_2$. All other examples of Figure 6 terminate with the three selection functions $\mathsf{sel}_0$, $\mathsf{sel}_1$, and $\mathsf{sel}_2$.

Among the examples of Figure 6, only the Woo-Lam shared key protocol, flawed and corrected versions of [47] and the Needham-Schroeder shared key protocol have explicit tags. Our tool terminates on all other protocols, even if they are not tagged. The termination can partly be explained by the notion of "implicitly tagged" protocols [21]: the various messages are not distinguished by explicit tags, but by other properties of their structure, such as the arity of the tuples that they contain. In Figure 6, the Denning-Sacco protocol and the Woo-Lam public key protocol are implicitly tagged. Still, the tool terminates on many examples that are not even implicitly tagged.

For the Yahalom protocol, we show that, if $B$ thinks that $k$ is a key to talk with $A$, then $A$ also thinks that $k$ is a key to talk with $B$. The converse is clearly wrong, because the session key is sent from $A$ to $B$ in the last message, so the adversary can intercept this message, so that $A$ has the key but not $B$.

For the Otway-Rees protocol, we do not have agreement on the session key, since the adversary can intercept messages in such a way that one participant has the key and the other one has no key. There is also an attack in which both participants get a key, but not the same one [45]. The latter attack is not found by our tool, since it stops with the former attacks.

For the simplified version of the Otway-Rees protocol given in [5], $B$ can execute its event $\texttt{event}(e(M_1, \ldots, M_n))$ with $A$ dead, and $A$ can execute its event $\texttt{event}(e(M_1, \ldots, M_n))$ with $B$ dead. As Burrows, Abadi, and Needham already noted in [27], even the original protocol does not guarantee to $B$ that $A$ is alive (attack against injective agreement that we also find). [47] said that the protocol satisfied its authentication specifications, because they showed that neither $A$ nor $B$ can conclude that $k$ is a key for talking between $A$ and $B$ without the server first saying so. (Of course, this property is also important, and could also be checked with our verifier.)

# 11 Conclusion

We have extended previous work on the verification of security protocols by logic programming techniques, from secrecy to a very general class of correspondences, including not only authentication but also, for instance, correspondences that express that the

messages of the protocol have been sent and received in the expected order. This technique enables us to check correspondences in a fully automatic way, without bounding the number of sessions of the protocols. This technique also yields an efficient verifier, as the experimental results demonstrate.

### Acknowledgments

We would like to thank Martín Abadi, Jérôme Feret, Cédric Fournet, and Andrew Gordon for helpful discussions on this paper. This work was partly done at Max-Planck-Institut für Informatik, Saarbrücken, Germany.

## References

[1] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[2] M. Abadi and B. Blanchet. Computer-assisted verification of a protocol for certified email. *Science of Computer Programming*, 58(1–2):3–27, Oct. 2005. Special issue SAS'03.

[3] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*, 10(3):1–59, July 2007.

[4] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, London, United Kingdom, Jan. 2001. ACM Press.

[5] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.

[6] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 140–154, Aix-en-Provence, France, June 2005. IEEE.

[7] R. Amadio and S. Prasad. The game of the name in cryptographic tables. In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science - ASIAN'99*, volume 1742 of *Lecture Notes on Computer Science*, pages 15–27, Phuket, Thailand, Dec. 1999. Springer.

[8] R. Anderson and R. Needham. Programming Satan's computer. In J. van Leeuven, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes on Computer Science*, pages 426–440. Springer, 1995.

[9] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 19–100. North Holland, 2001.

[10] M. Backes, A. Cortesi, and M. Maffei. Causality-based abstraction of multiplicity in security protocols. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 355–369, Venice, Italy, July 2007. IEEE.

[11] M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO 1993*, volume 773 of *Lecture Notes on Computer Science*, pages 232–249, Santa Barbara, California, Aug. 1993. Springer.

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

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

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

[15] B. Blanchet. Automatic proof of strong secrecy for security protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[16] B. Blanchet. Automatic proof of strong secrecy for security protocols. Technical Report MPI-I-2004-NWG1-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany, July 2004.

[17] B. Blanchet. Security protocols: From linear to classical logic by abstract interpretation. *Information Processing Letters*, 95(5):473–479, Sept. 2005.

[18] B. Blanchet. Automatic verification of correspondences for security protocols. Report arXiv:0802.3444v1, 2008. Available at `http://arxiv.org/abs/0802.3444v1`.

[19] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, Feb.–Mar. 2008.

[20] B. Blanchet and A. Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Symposium on Security and Privacy*, pages 417–431, Oakland, CA, May 2008. IEEE.

[21] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, Mar. 2005. Special issue FoSSaCS'03.

[22] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[23] P. Broadfoot, G. Lowe, and B. Roscoe. Automating data independence. In *6th European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *Lecture Notes on Computer Science*, pages 175–190, Toulouse, France, Oct. 2000. Springer.

[24] P. J. Broadfoot and A. W. Roscoe. Embedding agents within the intruder to detect parallel attacks. *Journal of Computer Security*, 12(3/4):379–408, 2004.

[25] M. Bugliesi, R. Focardi, and M. Maffei. Analysis of typed analyses of authentication protocols. In *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 112–125, Aix-en-Provence, France, June 2005. IEEE Comp. Soc. Press.

[26] M. Bugliesi, R. Focardi, and M. Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[27] 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.

[28] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. Deciding the security of protocols with Diffie-Hellman exponentiation and products in exponents. In P. K. Pandya and J. Radhakrishnan, editors, *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference*, volume 2914 of *Lecture Notes on Computer Science*, pages 124–135, Mumbai, India, Dec. 2003. Springer.

[29] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An NP decision procedure for protocol insecurity with XOR. *Theoretical Computer Science*, 338(1–3):247–274, June 2005.

[30] J. Clark and J. Jacob. A survey of authentication protocol literature: Version1.0. Technical report, University of York, Department of Computer Science, Nov. 1997.

[31] E. Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11(2):189–216, 2003.

[32] H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In *Symposium on Logic in Computer Science (LICS'03)*, pages 271–280, Ottawa, Canada, June 2003. IEEE Computer Society.

[33] V. Cortier, J. Millen, and H. Rueß. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 97–108, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[34] C. J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, Nov. 2006.

[35] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.

[36] H. de Nivelle. *Ordering Refinements of Resolution*. PhD thesis, Technische Universiteit Delft, Oct. 1995.

[37] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi. A new algorithm for the automatic verification of authentication protocols: From specifications to flaws and attack scenarios. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, New Jersey, Sept. 1997.

[38] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, Aug. 1981.

[39] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov. 1976.

[40] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, Mar. 1983.

[41] A. Durante, R. Focardi, and R. Gorrieri. CVS at work: A report on new failures upon some cryptographic protocols. In V. Gorodetski, V. Skormin, and L. Popyack, editors, *Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS'01)*, volume 2052 of *Lecture Notes on Computer Science*, pages 287–299, St. Petersburg, Russia, May 2001. Springer.

[42] 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.

[43] S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.

[44] S. Escobar, C. Meadows, and J. Meseguer. Equational cryptographic reasoning in the Maude-NRL protocol analyzer. *Electronic Notes in Theoretical Computer Science*, 171(4):23–36, July 2007.

[45] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[46] A. Gordon and A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In M. Okada, B. Pierce, A. Scedriv, H. Tokuda, and A. Yonezawa, editors, *Software Security – Theories and Systems, Mext-NSF-JSPS International Symposium, ISSS 2002*, volume 2609 of *Lecture Notes on Computer Science*, pages 263–282, Tokyo, Japan, Nov. 2002. Springer.

[47] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

[48] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2004.

[49] J. Goubault-Larrecq, M. Roger, and K. N. Verma. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 64(2):219–251, Aug. 2005.

[50] J. D. Guttman and F. J. T. Fábrega. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2002.

[51] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 255–268, Cambridge, England, July 2000.

[52] J. Heather and S. Schneider. A decision procedure for the existence of a rank function. *Journal of Computer Security*, 13(2):317–344, 2005.

[53] 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`.

[54] 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.

[55] G. Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop (CSFW '97)*, pages 31–43, Rockport, Massachusetts, June 1997. IEEE Computer Society.

[56] C. Lynch. Oriented equational logic programming is complete. *Journal of Symbolic Computation*, 21(1):23–45, 1997.

[57] C. Meadows and P. Narendran. A unification algorithm for the group Diffie-Hellman protocol. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, Jan. 2002.

[58] C. A. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[59] J. Millen and V. Shmatikov. Symbolic protocol analysis with an abelian group operator or Diffie-Hellman exponentiation. *Journal of Computer Security*, 13(3):515–564, 2005.

[60] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Mur$\varphi$. In *1997 IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[61] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

[62] R. M. Needham and M. D. Schroeder. Authentication revisited. *Operating Systems Review*, 21(1):7, 1987.

[63] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[64] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[65] 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.

[66] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. *Theoretical Computer Science*, 299(1–3):451–475, Apr. 2003.

[67] D. X. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[68] P. Syverson. A taxonomy of replay attacks. In *7th IEEE Computer Security Foundations Workshop (CSFW-94)*, pages 131–136, Franconia, New Hampshire, June 1994. IEEE Computer Society.

[69] P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes, and Cryptography*, 7(1/2):27–59, 1996.

[70] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–328, Trento, Italy, July 1999. Springer.

[71] T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, Jan. 1992.

[72] 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.

[73] T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. In D. Denning and P. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 319–355. ACM Press and Addison-Wesley, Oct. 1997.