From Secrecy to Authenticity in Security Protocols*

Bruno Blanchet[1,2]

[1] Département d'Informatique
École Normale Supérieure, Paris
[2] Max-Planck-Institut für Informatik
Bruno.Blanchet@ens.fr

Abstract. We present a new technique for verifying authenticity in cryptographic protocols. This 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 authentication specification to be verified, but no other annotation. Our technique has been proved correct, implemented, and tested on various protocols from the literature. The experimental results show that we can verify these protocols in less than 1 s.

1 Introduction

The verification of cryptographic protocols has already been the subject of numerous research works. It is indeed particularly important since flaws have been found in many published protocols. A recent trend in this area aims to verify protocols with an unbounded number of sessions, while relying as little as possible on human intervention. This goal is achieved using language-based techniques such as typing or abstract interpretation, that can handle infinite-state systems thanks to safe approximations. These techniques are not complete (a correct protocol can fail to typecheck, or false attacks can be found by abstract interpretation tools), but they are sound (when they do not find attacks, the protocol is guaranteed to satisfy the considered property). This is important for the certification of protocols.

Our goal in this paper is to extend previous work in this line of research by providing a fully automatic technique to verify authenticity in cryptographic protocols, without bounding the number of sessions of the protocol. Intuitively, a protocol authenticates A to B if, when B thinks he talks to A, then he actually talks to A. A simple and widely used definition of authenticity is the formalization by a correspondence property [19, 27], according to the following scheme. When B thinks he has run the protocol with A, he emits a special event `end`.

* This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

When A thinks she runs the protocol with B, she emits another event `begin`. Authenticity is satisfied when B cannot emit his `end` event without A having emitted her `begin` event. There are several variations along this scheme in the literature, and, as we show in the following of the paper, our technique can handle most of them. Our technique is based on a substantial extension of our previous verification technique for secrecy [1, 6]. (This technique for secrecy is also similar to that of [25].) We show that in some cases, a proof of secrecy for a modified protocol can directly give a proof of authenticity for the considered protocol. Then, we show how to extend our secrecy proof technique to handle the general case of authenticity.

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 instructions to emit `begin` and `end` events, to be able to represent the authenticity specification of the protocol. These events are the only required annotation of the protocol; no annotation is needed to help the tool proving authenticity.

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 names. Other extensions define the translation of events. Finally, this set of Horn clauses is passed to a solving algorithm, described in [6, 25]. This solver does not always terminate, but we conjecture that it terminates for a large class of well-designed protocols (see Sect. 4.3). Our experiments also demonstrate that it terminates on many examples of protocols. A minor extension of this solver is required to prove authenticity.

The main advantages of our method could be summarized as follows. It is fully automatic; the user only has to code the protocol and the authentication specification. 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 approximations that we describe in Sect. 4.2. The tool provides sufficient conditions for proving authenticity, but can fail on correct protocols, even if that did not happen in our experiments.

Related work We mainly focus on the few works that automatically verify authenticity in cryptographic protocols, without bounding the number of sessions.

Gordon and Jeffrey define a type system for verifying authenticity in cryptographic protocols, first for shared-key cryptography [14], then for public-key cryptography [15]. Our system allows more general cryptographic primitives (including hash functions and Diffie-Hellman key agreements), and more flexibility (for example, our system can verify the original Otway-Rees protocol [21]). Also, in our system, no annotation is needed, whereas in [14, 15], explicit type casts and checks have to be manually added. However, their system has the advantage that type checking always terminates, whereas in some rare cases, our analyzer does not.

Debbabi et al. [13] also verify authenticity thanks to a representation of protocols by inference rules, very similar to our Horn clauses. However, they verify a weaker notion of authenticity (corresponding to aliveness: if B terminates the protocol, then A must have been alive at some point before), and handle only shared-key cryptography.

A few other methods require little human effort, while supporting an unbounded number of runs: the verifier of [16], based on rank functions, can prove the correctness of or find attacks against protocols with atomic symmetric or asymmetric keys. Theorem proving [22] often requires manual intervention of the user. An exception to this is [12], but it deals only with secrecy. The theorem prover TAPS [11] often succeeds without or with little human intervention.

Model checking [18] in general implies a limit on the number of sessions of the protocol. This problem has been tackled by [7, 8, 23]. 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 [8], but with the additional restriction that the agents must be "factorisable". (Essentially, a single run of the agent has to be split into several runs such that each run contains only one fresh value.) Athena [24] uses strand spaces to reduce the state space, but still sometimes limits the number of sessions to guarantee termination.

Amadio and Prasad [4] already note that authenticity 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 authenticity property is not exactly the same as ours.

Outline Section 2 introduces our process calculus. Section 3 defines secrecy, and the various notions of authenticity that we can verify. Section 4 explains and proves our verification technique for authenticity, and relates it to the verification of secrecy. Most proofs are sketched or omitted because of space constraints. Section 5 gives our experimental results on a selection of cryptographic protocols of the literature.

2 The Process Calculus

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 $f(M_1, \ldots, M_n)$. 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

$M, N ::=$	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
$\mathtt{begin}(M).P$	begin event
$\mathtt{end}(M).P$	end event
$\mathtt{begin\_ex}(M)$	executed begin event
$\mathtt{end\_ex}(M)$	executed end event

Fig. 1. Syntax of the process calculus

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 reduction rules of the form $g(M_1, \ldots, M_n) \rightarrow M$ where $M_1, \ldots, M_n, M$ are terms without free names. We extend these rules by $g(M'_1, \ldots, M'_n) \rightarrow M'$ if and only if there exists a substitution σ and a reduction rule $g(M_1, \ldots, M_n) \rightarrow M$ in $\mathrm{def}(g)$ such that $M'_i = \sigma M_i$ for all $i \in \{1, \ldots, n\}$, and $M' = \sigma M$. We assume that the set $\mathrm{def}(g)$ is finite (it usually contains one or two rules in examples). Note that destructors are defined by reduction rules instead of the equalities used in [1]. This allows destructors to yield several different results non-deterministically. This extension is used in our modeling of Diffie-Hellman agreements[3]. Using constructors and destructors, we can represent data structures and cryptographic operations as summarized in Fig. 2.

The process calculus provides instructions for emitting events, that will be used in authenticity specifications. The process $\mathtt{begin}(M).P$ emits the event $\mathtt{begin}(M)$, then executes P. The process $\mathtt{end}(M).P$ emits the event $\mathtt{end}(M)$, then executes P. The constructs $\mathtt{begin\_ex}(M)$ and $\mathtt{end\_ex}(M)$ are useful to remember that the events $\mathtt{begin}(M)$ and $\mathtt{end}(M)$ respectively have been executed. They do not appear in processes at the beginning, but are introduced by reductions. We can show that, in our semantics, the number of executions `begin` is equal to the number of occurrences of `begin_ex` in the reduced process, and similarly for `end`.

[3] As explained in [2, 6], the Diffie-Hellman key agreement can be modeled using the equation $f(x, g(y)) = f(y, g(x))$. This equation does not fit directly in the framework of constructors and destructors, but it can be simulated by using two constructors g and h and a destructor f defined by $f(x, y) \rightarrow h(x, y)$ and $f(x, g(y)) \rightarrow h(y, g(x))$.

Tuples:
Constructor: tuple $ntuple(M_1, \ldots, M_n)$
Destructors: projections $ith_n(ntuple(M_1, \ldots, M_n)) \to M_i$
Shared-key encryption:
Constructor: encryption of M under the key N, $sencrypt(M, N)$
Destructor: decryption $sdecrypt(sencrypt(M, N), N) \to M$
Public-key encryption:
Constructors: encryption of M under the public key N, $pencrypt(M, N)$
public key generation from a secret key N, $pk(N)$
Destructor: decryption $pdecrypt(pencrypt(M, pk(N)), N) \to M$
Signatures:
Constructor: signature of M with the secret key N, $sign(M, N)$
Destructors: signature verification $checksign(sign(M, N), pk(N)) \to M$
message without signature $getmessage(sign(M, N)) \to M$
One-way hash functions:
Constructor: hash function $H(M)$.

Fig. 2. Constructors and destructors

The other constructs in the syntax of Fig. 1 come from the pi calculus. The input process $M(x).P$ inputs a message on channel M, and executes P with x bound to the input message. The output process $\overline{M}\langle N\rangle.P$ outputs the message N on the channel M and then executes P. Here, we use an arbitrary term M to represent a channel, but the process blocks if M does not reduce to a name at runtime. 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. Moreover, we define $let\ x = M\ in\ P$ as a syntactic sugar for $P\{M/x\}$, and $if\ M = N\ then\ P\ else\ Q$ as a shorthand for $let\ x = equal(M, N)\ in\ P\ else\ Q$, where the destructor $equal$ is defined by $equal(M, M) \to M$. 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 $M(x).P$, $let\ x = g(M_1, \ldots, M_n)\ in\ P\ else\ Q$. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively. A process is closed if 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 $x_1, \ldots, x_n$ with $M_1, \ldots, M_n$, respectively. A starting process is a process that does not contain executed events.

We extend the standard semantics to handle authenticity (Fig. 3).

As a running example, we consider a simplified version of the Woo and Lam one-way public-key authentication protocol, version of [28], 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.) The protocol is:

Message 1. $A \to B : pk_A$
Message 2. $B \to A : b$
Message 3. $A \to B : \{pk_A, pk_B, b\}_{sk_A}$

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

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

Fig. 3. Structural congruence and reduction

A first sends to B its public key, B replies with a nonce (fresh name), and A sends its public key, the public key of B, and the nonce all signed with its private key sk_A. This protocol can be represented in our calculus by the process P:

$$
\begin{aligned}
&P_A(sk_A, pk_A) = c(x\_pk_B).\mathtt{begin}(x\_pk_B).\overline{c}\langle pk_A\rangle.c(x\_b).\\
&\quad \overline{c}\langle \mathit{sign}((pk_A, x\_pk_B, x\_b), sk_A)\rangle\\
&P_B(sk_B, pk_B, pk_A) = c(x\_pk_A).(\nu b)\overline{c}\langle b\rangle.c(m).\\
&\quad \mathit{let}\ (y\_pk_A, y\_pk_B, y\_b) = \mathit{checksign}(m, x\_pk_A)\ \mathit{in}\\
&\quad \mathit{if}\ y\_pk_A = x\_pk_A\ \mathit{then}\ \mathit{if}\ y\_pk_B = pk_B\ \mathit{then}\ \mathit{if}\ y\_b = b\ \mathit{then}\\
&\quad \mathit{if}\ x\_pk_A = pk_A\ \mathit{then}\ \mathtt{end}(pk_B)\\
&P = (\nu sk_A)(\nu sk_B)\mathit{let}\ pk_A = pk(sk_A)\ \mathit{in}\ \mathit{let}\ pk_B = pk(sk_B)\ \mathit{in}\\
&\quad \overline{c}\langle pk_A\rangle.\overline{c}\langle pk_B\rangle.((!P_A(sk_A, pk_A)) \mid (!P_B(sk_B, pk_B, pk_A)))
\end{aligned}
$$

The channel c is public: The adversary can send and listen on it. We use a single public channel and not two or more channels because the adversary could take a message from one channel and relay it on another channel, thus removing any difference between the channels. The process P begins with the creation of the secret and public keys of A and B. The public keys are output on channel c to model that the adversary has them in its initial knowledge. Then the protocol itself starts: P_A represents A, P_B represents B. Both principals can run an unbounded number of sessions, hence the 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 of its interlocutor (this interlocutor is therefore chosen by the adversary). Then A starts the protocol by emitting a begin event $\texttt{begin}(x\_pk_B)$, whose intuitive meaning is "A has started a session with the host of public key x_pk_B". At the end of P_B, when B thinks he talks with A (that is, when $x\_pk_A = pk_A$), he emits an end event $\texttt{end}(pk_B)$, whose meaning is "B thinks he has completed a session with A".[4] If the protocol is correct, B should only emit $\texttt{end}(pk_B)$ when A has first emitted $\texttt{begin}(pk_B)$. This is what we are going to formalize and check in the following.

3 Definitions of Secrecy and Authenticity

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 Dolev and Yao. Thus, an adversary is any process that has a set of public names S in its initial knowledge.

Definition 1. *Let S be a finite set of names. The closed process Q is an S-*adversary *if and only if $fn(Q) \subseteq S$ and Q does not contain* `begin`*,* `end`*,* `begin_ex`*, or* `end_ex` *events.*

Secrecy We adopt the definition of secrecy of [1].

Definition 2. *We say that the closed process P* outputs *M on c if and only if $P\ (\rightarrow \cup \equiv)^*\ \overline{c}\langle M\rangle.R \mid R'$ for some processes R and R'. The closed process P* preserves the secrecy *of M from S if and only if $P \mid Q$ does not output M on c for any S-adversary Q and any $c \in S$.*

Non-injective agreement Non-injective agreement means that if an event $\texttt{end}(M)$ is executed, then $\texttt{begin}(M)$ has also been executed.

Definition 3. *The closed starting process P satisfies* non-injective agreement *with respect to S-adversaries if and only if for any S-adversary Q, for any P' such that $P \mid Q\ (\rightarrow \cup \equiv)^*\ P'$, for any M, if $\texttt{end\_ex}(M)$ occurs in P', then $\texttt{begin\_ex}(M)$ also occurs in P'. (In P', the restrictions are renamed such that they bind pairwise different names, and names different from free names.)*

Intuitively, in the example of Sect. 2, when non-injective agreement is satisfied, B cannot emit an `end` event without A having executed a `begin` event, so B cannot terminate the protocol thinking he talks to A, without A being actually involved in the protocol. Moreover, the agreement on the parameter of the events, $x\_pk_B = pk_B$, implies that A actually thinks she talks to B.

By choosing different values for the parameter M, the definition can encode some of Lowe's authentication properties [19] (non-injective agreement, aliveness, and a slightly stronger version of weak agreement).

[4] Note that the end event must not be emitted when B thinks he talks to the adversary. Indeed, in this case, it is correct that no begin event has been emitted by the interlocutor of B, since the adversary never emits events.

Injective agreement Injective agreement furthermore requires that the number of executions of $\mathtt{end}(M)$ is smaller than the number of executions of $\mathtt{begin}(M)$.

Definition 4. *The closed starting process P satisfies* injective agreement *with respect to S-adversaries if and only if for any S-adversary Q, for any P' such that $P \mid Q \ (\rightarrow \cup \equiv)^* \ P'$, for any M, the number of occurrences of* $\mathtt{end\_ex}(M)$ *in P' is at most the number of occurrences of* $\mathtt{begin\_ex}(M)$ *in P'. (In P', the restrictions are renamed such that they bind pairwise different names, and names different from free names.)*

This definition corresponds to Lowe's agreement specification [19], and to Woo and Lam's correspondence assertions [27].

4 Verification

4.1 A first idea

With the additional assumption that, at runtime, *when the events* $\mathtt{begin}(M)$ *or* $\mathtt{end}(M)$ *are executed, M does not contain bound names*, it is easy to prove authenticity by using secrecy, as shown by the following theorem:

Theorem 1. *Let $P = C[\mathtt{begin}(M).P_1, \mathtt{end}(M').P_2]$ be a closed starting process (where $C[]$ is any context), and assume that these are the only* begin *and* end *events in P. Let $P'(N) = C[\textit{if } M = N \textit{ then } 0 \textit{ else } P_1, \overline{c}\langle \mathrm{auth}(M')\rangle.P_2]$, where* auth *is a new function symbol. Assume that for all closed terms N, $P'(N)$ preserves the secrecy of* $\mathrm{auth}(N)$ *from S (with $c \in S$). Then P satisfies non-injective agreement with respect to S-adversaries.*

Intuitively, we use the following proof strategy : we fix a set $E_b = \{\mathtt{begin}(M) \mid M \neq N\}$ (for any N) of allowed begin events, and we modify the process P such that, if it tries to execute a begin event not in E_b, it blocks. We show that the modified process cannot execute $\mathtt{end}(N)$. Indeed, when P tries to execute $\mathtt{begin}(N)$, P' executes 0; and since P' preserves the secrecy of $\mathrm{auth}(N)$, it cannot execute $\overline{c}\langle \mathrm{auth}(N)\rangle$, so P cannot execute $\mathtt{end}(N)$. This implies non-injective agreement: if $\mathtt{begin}(N)$ cannot be executed, then $\mathtt{end}(N)$ cannot be executed.

Then, we could apply standard techniques for checking secrecy, for example that of [1, 6, 25], with only very minor extensions. However, the condition that bound names do not appear in events introduces important limitations: for example, we cannot check the agreement on a session key, nor injective agreement.

Our work in the following of this paper will explain how we can overcome these limitations. First, Theorem 1 has to be modified: in a trace that executes $\mathtt{end}(N)$, N may contain bound names whose scope does not include $\mathtt{begin}(M)$ in the initial process, so the term N cannot be written in the test *if* $M = N$ *then* 0 *else* P_1 in place of the $\mathtt{begin}(M)$ event. Fortunately, in the checker of [6, 25], bound names are represented as functions of the inputs that precede their creation, and these functions have no scope limit. So we are going to use this representation of bound names in our translation into Horn clauses.

Second, when translating the process into Horn clauses, we have to make sure that different terms are represented by different values in the checker, so that the else branch of the test $M = N$ can be precisely analyzed. (If different terms are merged in the checker, the checker can never be sure that M is equal to N, and so it must consider that the else branch may always be taken. This leads to an unacceptable loss of precision for the analysis of authenticity.) This is not true in the presence of bound names in the checkers of [6, 25], since names created by the same restriction after receiving the same inputs are merged. This merging can be avoided by adding a parameter to names, that we call a *session identifier*. Such a session identifier indicates which copy of the restriction created the name, and different copies will have different session identifiers.

Third, to prove injective agreement, we have to relate corresponding sessions of the two participants of the protocol. This is also done using session identifiers.

4.2 Translation into Horn clauses

We consider a closed starting process P_0 representing the protocol to check. We assume that different variables and names have different identifiers in P_0. (This can be achieved by renaming.) We also assume that P_0 contains a single occurrence of `begin` and a single occurrence of `end`. In most cases, when an authenticity specification contains several `begin` and `end` events, each occurrence of `end` should correspond to a fixed occurrence of `begin`. A constant element in the parameters of `begin` and `end` can enforce this property: $\mathtt{end}(c_1, \ldots)$ corresponds to $\mathtt{begin}(c_1, \ldots)$, $\mathtt{end}(c_2, \ldots)$ to $\mathtt{begin}(c_2, \ldots)$, and so on. The implementation supports this case, by verifying each correspondence independently: it first checks the correspondence between $\mathtt{begin}(c_1, \ldots)$ and $\mathtt{end}(c_1, \ldots)$, then for c_2, etc.

Given the closed starting process P_0 and a set of names S, the protocol checker builds a set of Horn clauses, representing the protocol in parallel with any S-adversary. These clauses contain "patterns" ranged over by p, p' and generated by the grammar:

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

For each name a in P_0 we have a corresponding pattern construct $a[p_1, \ldots, p_n, i_1, \ldots, i_{n'}]$. 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. If a is a free name, then the arity of this function is 0. If a is bound by a restriction $(\nu a)P$ in P_0, then this arity is the total number of input statements, destructor applications, and replications above the restriction $(\nu a)P$ in the abstract syntax tree of P_0. The first parameters of $a[\ldots]$ are patterns $p_1, \ldots, p_n$ corresponding to input statements and destructor applications. The last parameters of $a[\ldots]$ are session identifiers $i_1, \ldots, i_{n'}$ that are integers or variables (taken in a set V_s disjoint from the set V_o of ordinary variables). There is one session identifier for each replication above the restriction (νa). Closed patterns are named types, and denoted by T.

The clauses use 4 predicates: attacker, message, begin, and end. The fact $\mathrm{attacker}(p)$ means that the attacker may have p, $\mathrm{message}(p, p')$ that the message p' may appear on channel p, $\mathrm{begin}(p, \rho)$ that the event `begin` has been executed with a parameter corresponding to p and an environment ρ, and $\mathrm{end}(p, s)$ that `end` has been executed in session s with a parameter corresponding to p.

$F ::=$	facts
$\quad \mathrm{attacker}(p)$	attacker knowledge
$\quad \mathrm{message}(p, p')$	channel messages
$\quad \mathrm{begin}(p, \rho)$	begin event
$\quad \mathrm{end}(p, s)$	end event

The rules comprise rules for the attacker and rules for the protocol, defined below. These rules form the set $B_{P_0,S}$. The predicate $\mathrm{begin}(p, \rho)$ is defined by a set of closed facts B_b, that do not belong to $B_{P_0,S}$. The set B_b corresponds to the translation into Horn clauses of the set E_b used in the proof of Theorem 1.

Rules for the attacker The rules describing the attacker are essentially 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 $b[i]$, instead of only one fresh name $b[]$. 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 authenticity test to be correctly implemented, as seen in Sect. 4.1. The abilities of the attacker are then represented by the following rules:

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

$$\mathrm{attacker}(b[i]) \text{ where } b \notin \mathit{fn}(P_0) \text{ and } (\nu b) \text{ does not occur in } P_0 \tag{Rn}$$

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

$$\begin{array}{l}\text{For each destructor } g,\\ \quad \text{for each reduction } g(M_1, \ldots, M_n) \rightarrow M \text{ in } \mathrm{def}(g),\\ \qquad \mathrm{attacker}(M_1) \wedge \ldots \wedge \mathrm{attacker}(M_n) \Rightarrow \mathrm{attacker}(M)\end{array} \tag{Rg}$$

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

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

The rule (Init) represents the initial knowledge of the attacker. The rule (Rn) means that the attacker can generate an unbounded number of new names. The rules (Rf) and (Rg) mean that the attacker can apply all operations to all terms it has, (Rf) for constructors, (Rg) for destructors. Rule (Rl) means that the attacker can listen on all the channels it has, and (Rs) that it can send all the messages it has on all the channels it has.

Rules for the protocol The translation $[\![P]\!]\rho h$ of a process P is a set of rules, where the environment ρ is a sequence of mappings $x \mapsto p$ and $a \mapsto p$ from variables and names to patterns, and h is a sequence of facts of the form message(p, p'), and begin(p, ρ). The empty sequence is denoted by $\emptyset$, the concatenation of a fact F to the sequence h is denoted by $h \wedge F$. The concatenation of a mapping $u \mapsto p$ to ρ is denoted by $\rho[u \mapsto p]$, where u is a name or a variable. When M is a term, $\rho(M)$ is defined by considering ρ as a substitution: $\rho(u) = p$ when $u \mapsto p$ is in ρ, and $\rho(f(M_1, \ldots, M_n)) = f(\rho(M_1), \ldots, \rho(M_n))$ where f is a constructor. When V is a set of variables, $\rho(V)$ is a sequence of patterns containing p when $x \mapsto p$ is in ρ and $x \in V$, in the order of the sequence ρ.

$$[\![0]\!]\rho h = \emptyset$$
$$[\![P \mid Q]\!]\rho h = [\![P]\!]\rho h \cup [\![Q]\!]\rho h$$
$$[\![!P]\!]\rho h = [\![P]\!](\rho[i \mapsto i])h \text{ where } i \text{ is a new variable (session identifier)}$$
$$[\![(\nu a)P]\!]\rho h = [\![P]\!](\rho[a \mapsto a[\rho(V_o), \rho(V_s)]])h$$
$$[\![M(x).P]\!]\rho h = [\![P]\!](\rho[x \mapsto x])(h \wedge \text{message}(\rho(M), x))$$
$$[\![\overline{M}\langle N\rangle.P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \text{message}(\rho(M), \rho(N))\}$$
$$[\![\mathit{let}\ x = g(M_1, \ldots, M_n)\ \mathit{in}\ P\ \mathit{else}\ Q]\!]\rho h = \cup\{[\![P]\!]((\sigma\rho)[x \mapsto \sigma' p'])(\sigma h)$$
$$\mid g(p'_1, \ldots, p'_n) \to p' \text{ is in def}(g) \text{ and } (\sigma, \sigma') \text{ is a most general pair of}$$
$$\text{substitutions such that } \sigma\rho(M_1) = \sigma' p'_1, \ldots, \sigma\rho(M_n) = \sigma' p'_n\} \cup [\![Q]\!]\rho h$$
$$[\![\mathtt{begin}(M).P]\!]\rho h = [\![P]\!]\rho(h \wedge \text{begin}(\rho(M), \rho_{|V_o \cup V_s}))$$
$$[\![\mathtt{end}(M).P]\!]\rho h = [\![P]\!]\rho h \cup \{h \Rightarrow \text{end}(\rho(M), \rho(V_s))\}$$

The translation of a process is a set of Horn clauses that enable us to prove that it sends certain messages or executes certain events. The sequence h keeps track of begin events that have happened and of messages received by the process, since these may trigger other messages. Replication inserts a new session identifier i in the environment ρ. The idea is that this session identifier takes a different value for each copy of the replicated process. Replication is otherwise ignored, because all Horn clauses are applicable arbitrarily many times.

For restriction, we replace the restricted name a in question with the pattern $a[\rho(V_o), \rho(V_s)]$. This pattern contains the previous inputs in $\rho(V_o)$ and session identifiers in $\rho(V_s)$.

To make the intuition on session identifiers more precise, we can consider instrumented processes in which the replication is annotated with an inequality $i \geq n$, indicating the session identifier i and the values it can take. The process $!^{i \geq n} P$ then represents copies of P for each $i \geq n$ in parallel, that is, $P\{n/i\} \mid P\{n+1/i\} \mid P\{n+2/i\} \mid \ldots$. The semantic rule $!P \to P \mid !P$ then becomes $!^{i \geq n} P \to P\{n/i\} \mid !^{i \geq n+1} P$. In the process, we replace the names by their corresponding patterns. Since the pattern contains the session identifiers, and these session identifiers get different values for different copies of the process, different names are mapped to different patterns, even when they are created by the same restriction. Such an instrumentation is formalized and used in the proofs.

The translation of a `begin` event adds an hypothesis to h, meaning that P can only be executed if the `begin` event has been executed first. This is similar to the translation of a `begin` event into a test in Theorem 1. The begin fact has the part of the environment ρ containing variables as second parameter. This parameter will be useful for verifying injective agreement. In subsequent manipulations of this environment (in particular renamings), only the image is modified, the variables in the domain of ρ are left unchanged. (They are precisely used to remember names of variables in the original process.)

The translation of an `end` event adds a clause, meaning that the `end` event is triggered when all conditions in h are true. We can again notice similarity with the translation of an output, as suggested by Theorem 1. The second parameter of the end fact is the sequence of session identifiers of the session in which the `end` event is executed. This parameter is used for verifying injective agreement.

The other rules are as in [1]. 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.

This translation of the protocol into Horn clauses introduces several approximations. We consider that the *else* clause of a destructor can always be executed. The actions are considered as implicitly replicated, since the rules can be executed any number of times. The exception to this point 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.

If $c \in S$, we replace all occurrences of $\text{message}(c[], M)$ by $\text{attacker}(M)$ in the rules. Indeed, both facts are equivalent by the rules (Rl) and (Rs).

Summary and results Let $\rho = \{a \mapsto a[] \mid a \in \mathit{fn}(P_0)\}$. The rule base for P_0 is:

$$B_{P_0,S} = [\![P_0]\!]\rho\emptyset \cup \{\text{attacker}(a[]) \mid a \in S\} \cup \{(\text{Rn}), (\text{Rf}), (\text{Rg}), (\text{Rl}), (\text{Rs})\}$$

The rules for the process P of the end of Sect. 2 are rules for the attacker, plus:

$$\text{attacker}(pk(sk_A[])) \qquad \text{attacker}(pk(sk_B[]))$$

$$\text{attacker}(x\_pk_B) \wedge \text{begin}(x\_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto x\_pk_B\}) \Rightarrow \text{attacker}(pk(sk_A[]))$$

$$\text{attacker}(x\_pk_B) \wedge \text{begin}(x\_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto x\_pk_B\}) \wedge \text{attacker}(x\_b) \Rightarrow \text{attacker}(sign((pk(sk_A[]), x\_pk_B, x\_b), sk_A[]))$$

$$\text{attacker}(x\_pk_A) \Rightarrow \text{attacker}(b[x\_pk_A, i_B])$$

$$\text{attacker}(x\_pk_A) \wedge \text{attacker}(sign((x\_pk_A, pk(sk_B[]), b[x\_pk_A, i_B]), sk_A[])) \Rightarrow \text{end}(pk(sk_B[]), i_B)$$

The first two rules correspond to the outputs in P; they mean that the adversary has the public keys of the participants. The third and fourth rules correspond to the outputs in P_A. For example, the fourth rule means that if the attacker has x_pk_B and x_b and the event `begin`(x_pk_B) is allowed, then the attacker can get $sign((pk(sk_A[]), x\_pk_B, x\_b), sk_A[])$, because P_A sends this message after receiving x_pk_B and x_b and executing the `begin` event. The variables in the domain of the environment in the begin fact are underlined to mark that they are not renamed when manipulating rules. The fifth rule corresponds to the output in P_B and the last one to the `end` event in P_B.

The following theorems give sufficient conditions for proving authenticity.

Theorem 2. *Assume that, for any B_b, if* $\mathrm{end}(T, s)$ *is derivable from $B_{P_0,S} \cup B_b$, then* $\mathrm{begin}(T, \rho) \in B_b$ *for some ρ. Then P_0 satisfies non-injective agreement with respect to S-adversaries.*

Intuitively, the condition means that $\mathrm{end}(T, s)$ cannot be derived without having $\mathrm{begin}(T, \rho)$ for some ρ in the hypotheses. Then the theorem says that `end`(M) cannot be executed without having first executed `begin`(M).

Theorem 3. *Let* $\mathrm{getsession}(\rho)$ *be the sequence of terms contained at the occurrence o in $\rho(x)$ for some x and o, assuming that o corresponds to the occurrence of s in a subterm of $\rho(x)$ of the form $a[\ldots, s, \ldots]$ where the restriction (νa) is under the last replication above* `end` *in P_0. Otherwise,* $\mathrm{getsession}(\rho)$ *is undefined.*

Assume that, for any B_b, if $\mathrm{end}(T, s)$ *is derivable from $B_{P_0,S} \cup B_b$, then* $\mathrm{begin}(T, \rho) \in B_b$ *with* $\mathrm{getsession}(\rho) = s$. *Then P_0 satisfies injective agreement with respect to S-adversaries.*[5]

The intuition behind this result is as follows. Assume that we add the session identifier s of the `end` event (i_B in the example), to the parameter M of the events `begin` and `end`. Then non-injective agreement on (M, s) implies injective agreement. Indeed, only one `end` event can be executed for each value of s. Therefore, `end`(M) executed n times corresponds to `end`(M, s_1), ... `end`(M, s_n) executed for n distinct values $s_1, \ldots, s_n$, so `begin`(M, s_1), ..., `begin`(M, s_n) have been executed, so `begin`(M) has been executed at least n times.

We can easily add s to `end`(M), but for `begin`(M), the session identifier s is in general not in scope. This session identifier is going to appear in data received by the process, which are collected in environment ρ. So we add the environment ρ as parameter to the begin fact. The function getsession is going to extract the session identifier s from ρ, so that intuitively we prove non-injective agreement between `begin`$(M_1, \mathrm{getsession}(\rho))$ and `end`(M_2, s), to obtain injective agreement.

If we delay the emission of `begin` inside the same session, we can only decrease the number of emissions of `begin`:

Lemma 1. *Let $P_1 = C[$*`begin`*$(M).D[P]]$, where no replication occurs in $D[]$ above the hole $[]$. Let $P_2 = C[D[$*`begin`*$(M).P]]$. If P_2 satisfies injective agreement with respect to S-adversaries, then so does P_1.*

[5] Actually, we can prove, with the same hypotheses, that P_0 satisfies *recent* injective agreement, in the sense that the runtimes of the sessions that emit the `begin` and `end` events overlap. This point is not detailed here because of space constraints.

4.3 Solving algorithm

To determine whether a fact is derivable from $B_{P_0,S} \cup B_b$, we adapt the solving algorithm of [6]. Very roughly, the algorithm combines rules by resolution, starting from a set of rules B_0. From two rules $R = H \Rightarrow C$ and $R' = H' \Rightarrow C'$ it infers $\sigma(H \cup (H' - F_0)) \Rightarrow \sigma C'$, where C and $F_0 \in H'$ are unifiable, σ is the most general unifier of C and F_0, $\mathrm{sel}(R) = \emptyset$, $F_0 \in \mathrm{sel}(R')$, and sel is a selection function: $\mathrm{sel}(H \rightarrow C) = \emptyset$ if $H \subseteq \mathit{Unselectable}$, $\mathrm{sel}(H \rightarrow C) = \{F_0\}, F_0 \in H - \mathit{Unselectable}$ of maximum size otherwise, with $\mathit{Unselectable} = \{\mathrm{attacker}(x), x \text{ variable}\}$. So facts of the form $\mathrm{attacker}(x)$ are never selected. (In some rare cases, more sophisticated selection functions can be necessary to obtain termination, as explained in [6, extended version]. For simplicity, we do not consider this case here.) When no new rules can be added, we define $\mathrm{saturate}(B_0)$ as the set of obtained rules R such that $\mathrm{sel}(R) = \emptyset$. As shown in [6], a fact F is derivable from B_0 if and only if it is derivable from $\mathrm{saturate}(B_0)$. Intuitively, saturate transforms a set of rules into an equivalent but simpler one: the hypotheses of the resulting rules are all of the form $\mathrm{attacker}(x)$.

We conjecture that, for protocols using only public channels, public-key cryptography with atomic keys, shared-key cryptography and hash functions, the algorithm using the function sel above terminates when each encryption, signature, and hash computation 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 tag c_0 to m, so that the encryption becomes $\mathit{sencrypt}((c_0, m), k)$ where the tag c_0 is a different constant for each encryption in the protocol. 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. We are currently proving this point with A. Podelski.

In the case of authenticity, we modify the selection function such that it *never* selects facts of the form $\mathrm{begin}(p, \rho)$. (Then saturate returns rules containing hypotheses $\mathrm{attacker}(x)$ and $\mathrm{begin}(p, \rho)$.) An easy extension of [6] shows that:

Theorem 4. *F is derivable from $B_0 \cup B_b$ if and only if it is derivable from* $\mathrm{saturate}(B_0) \cup B_b$.

Intuitively, if an hypothesis $\mathrm{begin}(p, \rho)$ appears in all rules of $\mathrm{saturate}(B_0)$ that prove $\mathrm{end}(p, s)$, then this hypothesis is necessary to derive $\mathrm{end}(p, s)$. We obtain the following condition to prove non-injective agreement:

Theorem 5. *Assume that, for each $R \in \mathrm{saturate}(B_{P_0,S})$ such that $R = H \Rightarrow \mathrm{end}(p, s)$, we have $\mathrm{begin}(p, \rho) \in H$ for some ρ. Then P_0 satisfies non-injective agreement with respect to S-adversaries.*

Proof sketch. If $\mathrm{end}(T, s)$ is derivable from $B_{P_0,S} \cup B_b$, then it is derivable from $\mathrm{saturate}(B_{P_0,S}) \cup B_b$, by Theorem 4. The last rule R of this derivation has $\mathrm{end}(p, s')$ in its conclusion $R = H \Rightarrow \mathrm{end}(p, s')$, so by hypothesis, $\mathrm{begin}(p, \rho') \in H$. So $\mathrm{begin}(T, \rho)$ must be derivable from $\mathrm{saturate}(B_{P_0,S}) \cup B_b$. Since the only rules that prove $\mathrm{begin}(T, \rho)$ are the closed facts of B_b, $\mathrm{begin}(T, \rho) \in B_b$. So we have non-injective agreement by Theorem 2. $\square$

To prove injective agreement thanks to Theorem 3 and Lemma 1, we have to find the right position of $\texttt{begin}(M)$, since Lemma 1 allows to move it, and the value of the getsession function. We proceed as follows. First, we replace ρ in the hypothesis $\text{begin}(p, \rho)$ of each rule, by an environment that contains all variables defined at the output that creates the rule and that are not defined under a replication under $\texttt{begin}(M)$, instead of only the variables defined at the $\texttt{begin}(M)$ event. That is, $B'_{P_0,S}$ is the set of rules defined as $B_{P_0,S}$ except that

$$\llbracket \overline{M}\langle N\rangle.P \rrbracket \rho h = \llbracket P \rrbracket \rho h \cup \{\ h\{\rho_{|V_o \cup V_s}/X\} \Rightarrow \text{message}(\rho(M), \rho(N))\ \}$$
$$\llbracket !P \rrbracket \rho h = \llbracket P \rrbracket (\rho[i \mapsto i])(h\{\rho_{|V_o \cup V_s}/X\})$$
$$\llbracket \texttt{begin}(M).P \rrbracket \rho h = \llbracket P \rrbracket \rho (h \wedge \text{begin}(\rho(M), X))$$

where X is a special variable. Let $\text{occ}(s, p)$ be the set of occurrences of the sequence of session identifiers s in the pattern p that appear in a subpattern of the form $a[\ldots, s, \ldots]$, where the restriction (νa) is under the last replication above $\texttt{end}$ in P_0. Let $\text{occ}(s, \rho) = \{(x, o) \mid x \in \mathit{dom}(\rho), o \in \text{occ}(s, \rho(x))\}$ and $\text{occ}(B) = \cap_{H,p,s \text{ such that } H \Rightarrow \text{end}(p,s) \in B} \cup_{\rho \text{ such that begin}(p,\rho) \in H} \text{occ}(s, \rho)$.

Theorem 6. *If* $\text{occ}(\text{saturate}(B'_{P_0,S})) \neq \emptyset$*, then* P_0 *satisfies injective agreement with respect to* S*-adversaries.*

Proof sketch. Consider $(x, o) \in \text{occ}(\text{saturate}(B'_{P_0,S}))$. If $\texttt{begin}(M)$ is not under the binder of x in P_0, we move it just under the binder of x. Let P_1 be the obtained process. The rules of $B_{P_1,S}$ are the rules of $B'_{P_0,S}$, except that the hypotheses $\text{begin}(p, \rho)$ in which ρ does not contain x are removed, and in the other hypotheses $\text{begin}(p, \rho)$, ρ is restricted to contain only variables defined above x, including x. The rules in $\text{saturate}(B_{P_1,S})$ are transformed in the same way, so we still have $(x, o) \in \text{occ}(\text{saturate}(B_{P_1,S}))$. So for all $R \in \text{saturate}(B_{P_1,S})$, if $R = H \Rightarrow \text{end}(p, s)$ then $\text{begin}(p, \rho) \in H$ with $o \in \text{occ}(s, \rho(x))$. Similarly to the proof of Theorem 5, if $\text{end}(T, s)$ is derivable from $B_{P_1,S} \cup B_b$, then $\text{begin}(T, \rho) \in B_b$ with $o \in \text{occ}(s, \rho(x))$. The function $\text{getsession}(\rho)$ is chosen to return the term at occurrence o in $\rho(x)$, so $\text{getsession}(\rho) = s$. By Theorem 3, P_1 satisfies injective agreement with respect to S-adversaries, and by Lemma 1 so does P_0. $\square$

In the example of Sect. 2, the first rule for P_A is unchanged, and the second one becomes:

$$\text{attacker}(x\_pk_B) \wedge \text{attacker}(x\_b) \wedge \text{begin}(x\_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto x\_pk_B,$$
$$\underline{x\_b} \mapsto x\_b\}) \Rightarrow \text{attacker}(\mathit{sign}((\mathit{pk}(\mathit{sk}_A[]), x\_pk_B, x\_b), \mathit{sk}_A[]))$$

instead of having $\text{begin}(x\_pk_B, \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto x\_pk_B\})$ in all rules for P_A. The solving algorithm then yields

$$\text{begin}(\mathit{pk}(\mathit{sk}_B[]), \{\underline{i_A} \mapsto i_A, \underline{x\_pk_B} \mapsto \mathit{pk}(\mathit{sk}_B[]), \underline{x\_b} \mapsto b[\mathit{pk}(\mathit{sk}_A[]), i_B]\})$$
$$\Rightarrow \text{end}(\mathit{pk}(\mathit{sk}_B[]), i_B)$$

	# rules	Time (ms)
Needham-Schroeder public key [20]	21	25
Needham-Schroeder public key corrected [18]	21	16
Woo-Lam public key [26]	18	4
Woo-Lam public key corrected [28]	18	6
Woo-Lam shared key [14]	16	6
Woo-Lam shared key corrected [14]	15	5
Simpler Yahalom [9], unidirectional	10	29
Simpler Yahalom [9], bidirectional	13	101
Otway-Rees [21]	12	62
Simpler Otway-Rees [3]	12	10
Main mode of Skeme [17]	19	67

Fig. 4. Experimental results

Since x_b contains i_B, we move `begin` under the input that binds x_b. The process P_A then becomes:

$$P_{A2}(sk_A, pk_A) = c(x\_pk_B).\overline{c}\langle pk_A\rangle.c(x\_b).\mathtt{begin}(x\_pk_B)\ldots$$

We take getsession$(\rho) = i$ if $\rho(x\_b) = b[p, i]$, and getsession is undefined otherwise, so that getsession$(\rho) = i_B$ for the values of the variables obtained above. Thanks to Theorem 3, we prove that the modified process containing P_{A2} satisfies injective agreement with respect to $\{c\}$-adversaries, and by Lemma 1, we obtain the same result for P.

5 Experimental Results and Conclusion

Our experimental results are summarized in Fig. 4. The tests have been performed on a Pentium III, 1Ghz processor, with a verifier implemented in Ocaml 3.04. The protocols are given with a reference to the paper in which they appeared. The second column indicates the number of rules that represent the protocol. The third one gives the mean time needed to check one specification for each protocol.

The first six protocols (Needham-Schroeder public key and Woo-Lam protocols) are authentication protocols. For them, we have tested non-injective and injective agreement on the name of the participants, and on all atomic data.

For the Woo and Lam shared-key protocol, our tool proves the correctness of the corrected versions of [5] and [14]; it finds an attack on the flawed version of [14]. (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 [26] and [3, 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 [14] and that of [26] is that [14] adds tags to distinguish different encryption sites. Using tags is certainly a good design practice, and our analyzer terminates when tags are added. Our tool finds the attack of [10, bottom of page 52] on the versions of [3, end of Example 3.2] and [28].

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.

For all protocols, except the non-termination for two versions of Woo-Lam shared key, we found known attacks against flawed protocols, and proved the correctness of the corrected versions. No false attack was found. These results show that our fully automatic technique yields an efficient verifier for authenticity.

Acknowledgments

We would like to thank Martín Abadi for stimulating discussions on this work, Jérôme Feret and Andrew Gordon for helpful comments on a draft of this paper.

References

1. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th Annual ACM Symposium on Principles of Programming Languages (POPL 2002)*, pages 33–44, Portland, Oregon, Jan. 2002. ACM Press.
2. M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *28th Annual ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, London, United Kingdom, Jan. 2001. ACM Press.
3. M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.
4. 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 *LNCS*, pages 15–27, Phuket, Thailand, Dec. 1999. Springer Verlag.
5. R. Anderson and R. Needham. Programming Satan's Computer. In J. van Leeuven, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 426–440. Springer Verlag, 1995.
6. 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. Extended version available at `http://www.di.ens.fr/~blanchet/longcsfw14.ps.gz`.
7. P. Broadfoot, G. Lowe, and B. Roscoe. Automating Data Independence. In *6th European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *LNCS*, pages 175–190, Toulouse, France, Oct. 2000. Springer Verlag.
8. P. J. Broadfoot and A. W. Roscoe. Internalising agents in CSP protocol models. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, Jan. 2002.
9. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.
10. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version1.0. Technical report, University of York, Department of Computer Science, Nov. 1997.
11. E. Cohen. TAPS: A First-Order Verifier for Cryptographic Protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 144–158, Cambridge, England, July 2000.

12. 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.
13. 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.
14. A. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
15. A. Gordon and A. Jeffrey. Types and Effects for Asymmetric Cryptographic Protocols. In *15th IEEE Computer Security Foundations Workshop (CSFW-15)*, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.
16. J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 132–143, Cambridge, England, July 2000.
17. H. Krawczyk. SKEME: A Versatile Secure Key Exchange Mechanism for Internet. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996.
18. 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 *LNCS*, pages 147–166. Springer Verlag, 1996.
19. G. Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, Rockport, Massachusetts, June 1997. IEEE Computer Society.
20. R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.
21. D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *Operating Systems Review*, 21(1):8–10, 1987.
22. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
23. 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.
24. D. X. Song. Athena: a New Efficient Automatic Checker for Security Protocol Analysis. In *12th IEEE Computer Security Foundation Workshop (CSFW-12)*, Mordano, Italy, June 1999.
25. 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 Verlag.
26. T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, Jan. 1992.
27. T. Y. C. Woo and S. S. Lam. A Semantic Model for Authentication Protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 1993.
28. T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. In D. Denning and P. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*. ACM Press and Addison-Wesley, Oct. 1997.