Verification of Cryptographic Protocols: Tagging Enforces Termination

Bruno Blanchet[1,2] and Andreas Podelski[2]

[1] CNRS, Département d'Informatique
École Normale Supérieure, Paris
[2] Max-Planck-Institut für Informatik, Saarbrücken
Bruno.Blanchet@ens.fr, podelski@mpi-sb.mpg.de

Abstract. In experiments with a resolution-based verification method for cryptographic protocols, we could enforce its termination by *tagging*, a syntactic transformation of messages that leaves attack-free executions invariant. In this paper, we generalize the experimental evidence: we prove that the verification method always terminates for tagged protocols.

1 Introduction

The verification of cryptographic protocols is an active research area, see [1–33]. It is important since the design of protocols is error-prone, and testing cannot reveal potential attacks against the protocols. In this paper, we study a verification technique based on Horn clauses and resolution akin to [4, 5, 30]. We consider a protocol that is executed in the presence of an attacker that can listen to the network, compute, and send messages. The protocol and the attacker are translated into a set of Horn clauses such that: if the fact $\mathsf{att}(M)$ is not derivable from the clauses, then the protocol preserves the secrecy of the message M in every possible execution. The correctness verified is stronger than the one required since the executions possible in the Horn clause model include the ones where a send or receive instruction can be applied more than once in the same session. In practice, the difference between the correctness criteria does not show (no false alarm arose in our experiments).

The verification technique consists of the translation into Horn clauses, followed by the checking of the derivability of facts $\mathsf{att}(M)$ by a resolution-based algorithm. It has the following characteristics.

- It can verify protocols with an unbounded number of sessions.
- It can easily handle a variety of cryptographic primitives, including shared-key and public-key cryptography (encryption and signatures), hash functions, message authentication codes (mac), and even a simple model of Diffie-Hellman key agreements. It can also be used to verify authenticity [5].
- It is efficient (many examples of protocols of the literature are verified in less than 0.1 s; see [5]).

The resolution-based verification algorithm has one drawback: it does not terminate in general. In fact, in our experiments, the algorithm did not terminate (went into an infinite loop) when we applied it to the Needham-Schroeder shared-key protocol [4] and several versions of the Woo-Lam shared-key one-way authentication protocol [5]. It is always possible to modify the algorithm to make it work on those cases and any finite number of other cases, but that will not affect its inherent non-termination property (inherent by the undecidability of the problem that it tries to solve). In this paper, we investigate an alternative: tagging the protocol.

Tagging consists in adding a unique constant to each message. This is a syntactic operation done once on the textual representation of the protocol; i.e., no new tags are generated during the execution of the protocol, in contrast to nonces. For instance, to encrypt the message m under the key k, we add the tag c_0 to m, so that the encryption becomes $\mathsf{sencrypt}((\mathsf{c}_0, m), k)$. The tagged protocol retains the intended behaviour of the original protocol; i.e., the attack-free executions are the same. Under attacks, it is possibly more secure. Therefore, tagging is a feature of a good protocol design, as explained e.g. in [2]: the receiver of a message uses the tag to identify it unambiguously; thus tagging prevents *type flaws* that occur when a message is taken for another message. (This is formally proved in [19] for a tagging scheme very similar to ours.) Tagging is also motivated by practical issues: the decoding of incoming messages becomes easier. For all these reasons, tags are already present in protocols such as SSH.

In our experiments (including the protocols mentioned above), we obtained termination after tagging the protocol. In this paper, we give the theory behind the experiments: the resolution-based verification algorithm always terminates on tagged protocols. More precisely, the algorithm terminates on protocols where tags are added to each use of a cryptographic primitive, which may be among: public-key cryptography where keys are atomic, shared-key cryptography (unrestricted), hash functions, and message authentication codes (mac's).

This means that we show termination for a class of protocols that includes many relevant examples.

Overview To verify a protocol, we translate it into a set of Horn clauses. In Section 2, we give an example of such a translation and define the formal protocol models that form the input of the algorithm. We then, in Section 3, define the algorithm to check secrecy properties of a protocol. Sections 2 and 3 provide the background for the results in this paper; they recapitulate and extend material from [4]. In Section 4, we give a number of formal properties of sets of Horn clauses. The properties do not only express that the protocol is tagged, but also that the set of Horn clauses arises as a model of a 'reasonable' protocol. In Section 5, we prove that the algorithm is guaranteed to terminate for tagged protocols, i.e. for Horn clause models that satisfy the properties defined in Section 4. In Section 6, we establish the complexity of the algorithm for tagged protocols. The algorithm can be extended to authentication; in Section 7, we answer the question whether the termination result still holds for the exten-

Tuples:
Constructor: tuple $(M_1, \ldots, M_n)$
Destructors: projections $i\mathsf{th}_n((M_1, \ldots, M_n)) \rightarrow M_i$
Shared-key encryption:
Constructor: encryption of M under the key N, $\mathsf{sencrypt}(M, N)$
Destructor: decryption $\mathsf{sdecrypt}(\mathsf{sencrypt}(M, N), N) \rightarrow M$
Public-key encryption:
Constructors: encryption of M under the public key N, $\mathsf{pencrypt}(M, N)$
public key generation from a secret key N, $\mathsf{pk}(N)$
Destructor: decryption $\mathsf{pdecrypt}(\mathsf{pencrypt}(M, \mathsf{pk}(N)), N) \rightarrow M$
Signatures:
Constructor: signature of M with the secret key N, $\mathsf{sign}(M, N)$
Destructors: signature verification $\mathsf{checksignature}(\mathsf{sign}(M, N), \mathsf{pk}(N)) \rightarrow M$
message without signature $\mathsf{getmessage}(\mathsf{sign}(M, N)) \rightarrow M$
Non-message-revealing signatures:
Constructors: signature of M with the secret key N, $\mathsf{nmrsign}(M, N)$
constant true
Destructor: signature verification $\mathsf{nmrchecksign}(\mathsf{nmrsign}(M, N), \mathsf{pk}(N), M) \rightarrow \mathsf{true}$
One-way hash functions:
Constructor: hash function $\mathsf{hash}(M)$.
Message authentication codes, keyed hash functions:
Constructor: mac of M with key N, $\mathsf{mac}(M, N)$

Fig. 1. Constructors and destructors

sion of the algorithm to authentication. We relate our work to existing work in Section 8.

2 The Horn Clause Model of a Protocol

Cryptographic protocols can be translated into Horn clauses, either by hand, as explained in [4, 30], or automatically, for instance, from a representation of the protocol in an extension of the pi calculus, as in [1].

The terms in the Horn clauses stand for messages. The translation uses one predicate att. The fact $\mathsf{att}(M)$ means that the attacker may have the term M. The fundamental property of this representation is that if $\mathsf{att}(M)$ is not derivable from the clauses, then the protocol preserves the secrecy of M.

The clauses are of two kinds: the clauses in $\mathcal{R}_{\mathrm{Primitives}}$ that depend only on the signature of the cryptographic primitives (they represent computation abilities of the attacker) and the clauses in $\mathcal{R}_{\mathrm{Prot}}$ that one extracts from the protocol itself.

Attacker Clauses ("$\mathcal{R}_{\mathrm{Primitives}}$") The protocols use cryptographic primitives of two kinds: constructors and destructors (see Figure 1). A constructor f is used to build up a new term $f(M_1, \ldots, M_n)$. For example, the term $\mathsf{sencrypt}(M, N)$ is the encoding of the term M with the key N (by shared-key encryption). A destructor g applied to terms $M_1, \ldots, M_n$ yields a term M built up from subterms of $M_1, \ldots, M_n$. It is defined by a finite set $\mathrm{def}(g)$ of equations written as reduction rules $g(M_1, \ldots, M_n) \to M$ where the terms $M_1, ..., M_n, M$ contain only constructors and variables. For example, the rule $\mathsf{sdecrypt}(\mathsf{sencrypt}(M, N), N) \to M$ models the decoding of the term $\mathsf{sencrypt}(M, N)$ with the same key used for the encoding.

The attacker can form new messages by applying constructors and destructors to already obtained messages. This is modeled, for instance, by the following clauses for shared-key encryption.

$$\mathsf{att}(x) \wedge \mathsf{att}(y) \to \mathsf{att}(\mathsf{sencrypt}(x, y)) \qquad (\mathsf{sencrypt})$$

$$\mathsf{att}(\mathsf{sencrypt}(x, y)) \wedge \mathsf{att}(y) \to \mathsf{att}(x) \qquad (\mathsf{sdecrypt})$$

The first clause expresses that if the attacker has the message x and the shared key y, then he can form the message $\mathsf{sencrypt}(x, y)$. The second clause means that if the attacker has the message $\mathsf{sencrypt}(x, y)$ and the key y, then he can obtain the message x (by applying the destructor $\mathsf{sdecrypt}$ and then using the equality between $\mathsf{sdecrypt}(\mathsf{sencrypt}(x, y), y)$ and x according to the reduction rule for $\mathsf{sdecrypt}$).

We furthermore distinguish between *data* and *cryptographic* constructors and destructors and thus, in total, between four kinds of primitives. The set *DataConstr* of data constructors contains those f that come with a destructor g_i defined by $g_i(f(x_1, \ldots, x_n)) \to x_i$ for each $i = 1, \ldots, n$; i.e. g_i is used for selecting the argument of f in the i-th position. It is generally sufficient to have only tuples as data constructors (with projections as destructors). All other constructors are said to be cryptographic constructors; they form the set *CryptoConstr*. We collect all clauses like the two example clauses above, for each of the four cases, in the set $\mathcal{R}_{\mathrm{Primitives}}$ of clauses or *rules* defined below.

Definition 1 (Program for Primitives, $\mathcal{R}_{\mathrm{Primitives}}$). *The program for primitives, $\mathcal{R}_{\mathrm{Primitives}}$, is the union of the four sets of Horn clauses corresponding to each of the four cases of cryptographic primitives:*

- $\mathcal{R}_{\mathrm{CryptoConstr}}$ *is the set of clauses* $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \to \mathsf{att}(f(x_1, \ldots, x_n))$ *where f is a cryptographic constructor.*
- $\mathcal{R}_{\mathrm{DataConstr}}$ *is the set of clauses* $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \to \mathsf{att}(f(x_1, \ldots, x_n))$ *where f is a data constructor.*
- $\mathcal{R}_{\mathrm{CryptoDestr}}$ *is the set of clauses* $\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \to \mathsf{att}(M)$ *where g is a cryptographic destructor with the reduction rule $g(M_1, \ldots, M_n) \to M$.*
- $\mathcal{R}_{\mathrm{DataDestr}}$ *is the set of clauses* $\mathsf{att}(f(x_1, \ldots, x_n)) \to \mathsf{att}(x_i)$ *where f is a data constructor and $i = 1, \ldots, n$.*

Protocol Clauses ("$\mathcal{R}_{\mathrm{Prot}}$") We note $\mathcal{R}_{\mathrm{Prot}}$ the set of *protocol clauses*. These include clauses that directly correspond to send and receive instructions of the protocol and clauses translating the initial knowledge of the attacker.

In a protocol clause of the form

$$\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n) \to \mathsf{att}(M)$$

the term M in the conclusion represents the sent message. The hypotheses correspond to messages received by the same host before sending M. Indeed, the clause means that if the attacker has $M_1, ..., M_n$, he can send these messages to a participant who is then going to reply with M, and the attacker can then intercept this message.

If the initial knowledge of the attacker consists of the set of terms S_{Init} (containing e.g. public keys, host names, and a name N that represents all names that the attacker creates), then it is represented by the facts $\mathsf{att}(M)$ where M is a term in S_{Init}.

We explain protocol clauses on the example of the Yahalom protocol [8]:

Message 1.	$A \rightarrow B:$	(A, N_a)
Message 2.	$B \rightarrow S:$	$(B, \{A, N_a, N_b\}_{K_{bs}})$
Message 3.	$S \rightarrow A:$	$(\{B, K_{ab}, N_a, N_b\}_{K_{as}}, \{A, K_{ab}\}_{K_{bs}})$
Message 4.	$A \rightarrow B:$	$(\{A, K_{ab}\}_{K_{bs}}, \{N_b\}_{K_{ab}})$

In this protocol, two participants A and B wish to establish a session key K_{ab}, with the help of a trusted server S. Initially, A has a shared key K_{as} to communicate with S, and B has a shared key K_{bs} to communicate with S. In the first message, A sends to B his name A and a nonce (fresh value) N_a. Then B creates a nonce N_b and sends to the server his own name B and the encryption $\{A, N_a, N_b\}_{K_{bs}}$ of A, N_a, N_b under the shared key K_{bs}. The server then creates the new (fresh) session key K_{ab}, and sends two encrypted messages to A. The first one $\{B, K_{ab}, N_a, N_b\}_{K_{as}}$ gives the key K_{ab} to A, together with B's name and the nonces (so that A knows that the key is intended to communicate with B). The second message cannot be decrypted by A, so A forwards it to B (message 4). B then obtains the session key K_{ab}. The second part of message 4, $\{N_b\}_{K_{ab}}$, is used to check that A and B really use the same key K_{ab}: B is going to check that he can decrypt the message with the newly received key. We encode only one principal playing each role, since others can be included in the attacker.

Message 1 is represented by the clause

$$\mathsf{att}((\mathsf{host}(\mathsf{Kas}), \mathsf{Na})) \tag{Msg1}$$

meaning that the attacker gets $\mathsf{host}(\mathsf{Kas})$ and Na when intercepting message 1. In this clause, the host name A is represented by $\mathsf{host}(\mathsf{Kas})$. Indeed, the server has a table of pairs (host name, shared key to communicate between that host and the server), and this table can be conveniently represented by a constructor host. This constructor takes as parameter the secret key and returns the host name. So host names are written $\mathsf{host}(k)$. The server can also match a term $\mathsf{host}(k)$ to find back the secret key. The attacker cannot do this operation (he does not have the key table), so there is no destructor clause for host. There is a constructor clause, since the attacker can build new hosts with new host keys:

$$\mathsf{att}(k) \rightarrow \mathsf{att}(\mathsf{host}(k)) \tag{host}$$

Message 2 is represented by the clause:

$$\mathsf{att}((a, na)) \rightarrow \mathsf{att}((\mathsf{host}(\mathsf{Kbs}), \mathsf{sencrypt}((a, na, \mathsf{Nb}(a, na)), \mathsf{Kbs}))) \tag{Msg2}$$

The hypothesis means that a message (a, na) (corresponding to message 1) must be received before sending message 2. It corresponds to the situation in which the attacker sends (a, na) to B, B takes that for message 1, and replies with message 2, which is intercepted by the attacker. (a and na are variables since B accepts any term instead of $\mathsf{host}(\mathsf{Kas})$ and Na.) The nonce N_b is represented by the function $\mathsf{Nb}(a, na)$. Indeed, since a new name is created at each execution, names created after receiving different messages are different. This is modeled by considering names as functions of the messages previously received. The creation of a fresh name corresponds to an existential quantification in the linear logic model of [15]; our modeling of names by functions corresponds to a skolemization of these existential quantifiers. This modeling is slightly weaker than creating a new name at each run of the protocol, but it is correct: if a secrecy property is proved in this model, then it is true [1]. The introduced function symbols will be called "name function symbols". (In message 1, the fresh name Na is a constant because there are no previous messages on which it would depend.)

Message 3 is represented by the clause:

$$\begin{aligned}&\mathsf{att}((\mathsf{host}(kbs), \mathsf{sencrypt}((\mathsf{host}(kas), na, nb), kbs)))\\ &\quad\to \mathsf{att}((\mathsf{sencrypt}((\mathsf{host}(kbs), \mathsf{Kab}(kas, kbs, na, nb), na, nb), kas),\\ &\qquad\qquad \mathsf{sencrypt}((\mathsf{host}(kas), \mathsf{Kab}(kas, kbs, na, nb)), kbs)))\end{aligned} \tag{Msg3}$$

using the same principles. At last, message 4 is represented by

$$\mathsf{att}((\mathsf{sencrypt}((b, k, \mathsf{Na}, nb), \mathsf{Kas}), mb)) \to \mathsf{att}((mb, \mathsf{sencrypt}(nb, k))) \tag{Msg4}$$

The message $\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), k), \mathsf{Kbs})$ cannot be decrypted and checked by A, so it is a variable mb.

The goal of the protocol is to establish a secret shared key K_{ab} between A and B. If the key was a constant, say $\mathsf{K_{ab}}$, then the non-derivability of the fact $\mathsf{att}(\mathsf{K_{ab}})$ from the Horn clauses presented so far would prove its secrecy. However, K_{ab}, as received by A, is a variable k. We therefore use the following fact. The key K_{ab} received by A is secret if and only if some constant $\mathsf{secretA}$ remains secret when A sends it encrypted under the key K_{ab}. Thus, we add a clause that corresponds to the translation of an extra message of the protocol, Message 5. $A \to B:\ \{\mathsf{secretA}\}_{K_{ab}}$.

$$\begin{aligned}&\mathsf{att}((\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kbs}), k, \mathsf{Na}, nb), \mathsf{Kas}), mb))\\ &\quad\to \mathsf{att}(\mathsf{sencrypt}(\mathsf{secretA}, k))\end{aligned} \tag{Msg5}$$

Now, the secrecy of the key K_{ab} received by A can be proved from the non-derivability of the fact $\mathsf{att}(\mathsf{secretA})$ from the set of clauses $\mathcal{R}_{\mathrm{Primitives}} \cup \mathcal{R}_{\mathrm{Prot}}$.

For the Yahalom protocol, the translation yields the union of the following sets of Horn clauses. $\mathcal{R}_{\mathrm{CryptoConstr}}$ contains (sencrypt) and (host), $\mathcal{R}_{\mathrm{CryptoDestr}}$ contains (sdecrypt), $\mathcal{R}_{\mathrm{DataConstr}}$ contains the tuple construction and $\mathcal{R}_{\mathrm{DataDestr}}$ the tuple projections (both not listed), and $\mathcal{R}_{\mathrm{Prot}}$ contains (Msg1), (Msg2), (Msg3), (Msg4) and (Msg5) and three clauses translating the initial knowledge, $\mathsf{att}(\mathsf{N})$, $\mathsf{att}(\mathsf{host}(\mathsf{Kas}))$, and $\mathsf{att}(\mathsf{host}(\mathsf{Kbs}))$.

3 The Resolution-Based Verification Algorithm

To determine whether a fact is derivable from the clauses, we use a resolution-based algorithm explained below. (We use the meta-variables R, H, C, F for rule, hypothesis, conclusion, fact, respectively.)

The algorithm infers new clauses by resolution as follows: From two clauses $R = H \rightarrow C$ and $R' = F \wedge H' \rightarrow C'$ (where F is any hypothesis of R'), it infers $R \circ_F R' = \sigma H \wedge \sigma H' \rightarrow \sigma C'$, where C and F are unifiable and σ is the most general unifier of C and F. The clause $R \circ_F R'$ is the combination of R and R', where R proves the hypothesis F of R'. The resolution is guided by a selection function sel. Namely, $sel(R)$ returns a subset of the hypotheses of R, and the resolution step above is performed only when $sel(R) = \emptyset$ and $F \in sel(R')$.

We can use several selection functions. In this paper, we use:

$$sel(H \rightarrow C) = \begin{cases} \emptyset & \text{if all elements of } H \text{ are of the form } \mathsf{att}(x),\ x \text{ variable} \\ \{F\} & \text{where } F \neq \mathsf{att}(x) \text{ and } F \in H, \text{ otherwise} \end{cases}$$

The algorithm uses the following optimizations:

- Decomposition of data constructors: *decomp* takes a clause and returns a set of clauses, built as follows. For each data constructor f, *decomp* replaces recursively all facts $\mathsf{att}(f(M_1, \ldots, M_n))$ with $\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n)$. When such a fact is in the conclusion of a clause, n clauses are created, with the same hypotheses and the conclusions $\mathsf{att}(M_1), \ldots, \mathsf{att}(M_n)$ respectively. With decomposition, the standard clauses for data constructors and projections can be removed. The soundness of this operation follows from the equivalence between $\mathsf{att}(f(M_1, \ldots, M_n))$ and $\mathsf{att}(M_1) \wedge \ldots \wedge \mathsf{att}(M_n)$ in the presence of the clauses $\mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \rightarrow \mathsf{att}(f(x_1, \ldots, x_n))$ and $\mathsf{att}(f(x_1, \ldots, x_n)) \rightarrow \mathsf{att}(x_i)$ in $\mathcal{R}_{\text{DataConstr}}$ and $\mathcal{R}_{\text{DataDestr}}$.
- Elimination of duplicate hypotheses: *elimdup* takes a clause and returns the same clause after keeping only one copy of duplicate hypotheses.
- Elimination of hypotheses $\mathsf{att}(x)$: *elimattx* eliminates hypotheses $\mathsf{att}(x)$ when x does not appear elsewhere in the clause. Indeed, these hypotheses are always true, since the attacker has at least one term.
- Elimination of tautologies: *elimtaut* eliminates all tautologies (that is, clauses whose conclusion is already in the hypotheses) from a set of clauses.
- *simplify* groups all these simplifications. We extend *elimdup* and *elimattx* naturally to sets of clauses, and define *simplify* $=$ *elimtaut* $\circ$ *elimattx* $\circ$ *elimdup* $\circ$ *decomp*.
- *condense*($\mathcal{R}$), for a set of clauses $\mathcal{R}$, applies *simplify* to each clause in $\mathcal{R}$ and then eliminates subsumed clauses. We say that $H_1 \rightarrow C_1$ subsumes $H_2 \rightarrow C_2$ if and only if there exists a substitution σ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$. If $\mathcal{R}$ contains clauses R and R', such that R subsumes R', R' is eliminated. (In that case, R can do all derivations that R' can do.)

We now define the algorithm *saturate*($\mathcal{R}_0$). Starting from *condense*($\mathcal{R}_0$), the algorithm adds clauses inferred by resolution with the selection function *sel* and condenses the set of clauses at each iteration step until a fixpoint is reached. When a fixpoint is reached, $\mathit{saturate}(\mathcal{R}_0)$ consists of the clauses R in the fixpoint such that $\mathit{sel}(R) = \emptyset$. By adapting the proof of [4] to this algorithm, it is easy to show that, for any $\mathcal{R}_0$ coming from a protocol and any closed fact F, F is derivable from $\mathcal{R}_{\mathrm{All}} = \mathcal{R}_0 \cup \mathcal{R}_{\mathrm{DataConstr}} \cup \mathcal{R}_{\mathrm{DataDestr}}$ if and only if it is derivable from $\mathit{saturate}(\mathcal{R}_0) \cup \mathcal{R}_{\mathrm{DataConstr}}$.

Once the clauses of $\mathit{saturate}(\mathcal{R}_0)$ have been computed, we use a standard backward depth-first search to see if a fact can be derived from $\mathit{saturate}(\mathcal{R}_0) \cup \mathcal{R}_{\mathrm{DataConstr}}$. Taking $\mathcal{R}_0 = \mathcal{R}_{\mathrm{CryptoConstr}} \cup \mathcal{R}_{\mathrm{CryptoDestr}} \cup \mathcal{R}_{\mathrm{Prot}}$, if $\mathsf{att}(M)$ cannot be derived from $\mathit{saturate}(\mathcal{R}_0) \cup \mathcal{R}_{\mathrm{DataConstr}}$ then the protocol preserves the secrecy of M.

The optimizations enable us to weaken the conditions that guarantee termination. For instance, the decomposition of data constructors makes it possible to obtain termination without tagging each data constructor application, while other constructors such as encryption must be tagged. In the Yahalom protocol, for example, without decomposition of data constructors, the algorithm would resolve the clause (Msg2) with itself, immediately yielding an infinite loop.

Another consequence of the optimizations is that not all terms in a clause can be variables. Indeed, when $x \in \{x_1, \dots, x_n\}$, the clause $\mathsf{att}(x_1) \wedge \dots \wedge \mathsf{att}(x_n) \rightarrow \mathsf{att}(x)$ is eliminated since it is a tautology. When $x \notin \{x_1, \dots, x_n\}$, all hypotheses are eliminated, so the clause becomes $\mathsf{att}(x)$ and all other clauses are eliminated since they are subsumed by $\mathsf{att}(x)$, so the algorithm stops immediately: all facts can be derived. Thus, when $\mathit{sel}(R) = \emptyset$, the conclusion of R is not of the form $\mathsf{att}(x)$. Therefore, the above selection function prevents resolution steps in which $\mathsf{att}(x)$ is unified with another fact (actually, with any other fact, which can lead to non-termination).

4 Sufficient Conditions for Termination

We are now collecting the formal properties of sets of Horn clauses (logic programs, or programs for short) that together entail termination. The properties for *protocol programs* hold for the translation of every protocol. The properties for *plain* protocol programs hold for the translation of protocols with a restriction on their cryptographic primitives and on their keys (this restriction is satisfied by many interesting protocols, including Yahalom for example). The properties for *tagged* protocol programs hold for the translation of those protocols after they have been tagged. The derivability problem for plain protocol programs is undecidable (as can be easily seen by a reduction to two-counter machines). The restriction to tagged programs makes the problem decidable, as will follow.

Given a clause R of the form $\mathsf{att}(M_1) \wedge \dots \wedge \mathsf{att}(M_n) \rightarrow \mathsf{att}(M_0)$, we say that the terms $M_0, M_1, \dots, M_n$ are the *terms of* R, and we denote the set of terms of R by $\mathit{terms}(R)$.

Definition 2 (Protocol program). *A protocol program is a set of clauses $\mathcal{R}_{\mathrm{All}} = \mathcal{R}_{\mathrm{Primitives}} \cup \mathcal{R}_{\mathrm{Prot}}$ (where $\mathcal{R}_{\mathrm{Primitives}}$ is a program for primitives) that comes with a finite set of closed terms S_0 such that:*

C1. For all clauses R in $\mathcal{R}_{\mathrm{Prot}}$, there exists a substitution σ such that $terms(\sigma R) \subseteq S_0$.
C2. Every two subterms of terms in S_0 of the form $a(\ldots)$ with the same name function symbol a are identical.
C3. The second argument of $\mathsf{pencrypt}$ *in S_0 is of the form $\mathsf{pk}(M)$ for some M.*

The terminology "argument of f in S_0" refers to a term M such that $f(\ldots, M, \ldots)$ is a subterm of a term in S_0. To see why these conditions are satisfied by a translation of a protocol, let us consider the intended messages of the protocol. These are the exchanged messages when the attacker does not intervene and when there is no unexpected interaction between sessions of the protocol. We denote by $M_1, \ldots, M_k$ the closed terms corresponding to these messages. Each participant does not necessarily have a full view of the messages he receives; instead, he accepts all messages that are instances of patterns representing the information he can check. The terms $M_1, \ldots, M_k$ are particular instances of these patterns. So the protocol is represented by clauses R such that there exists σ such that $terms(\sigma R) \subseteq \{M_1, \ldots, M_k\}$. Defining $S_0 = \{M_1, \ldots, M_k\} \cup S_{\mathrm{Init}}$, we obtain C1.

For instance, the intended messages for the Yahalom protocol are

$$M_1 = (\mathsf{host}(\mathsf{Kas}), \mathsf{Na})$$
$$M_2 = (\mathsf{host}(\mathsf{Kbs}), \mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), \mathsf{Na}, M_{N_b}), \mathsf{Kbs}))$$
$$M_3 = (\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kbs}), M_K, \mathsf{Na}, M_{N_b}), \mathsf{Kas}), \mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), M_K), \mathsf{Kbs}))$$
$$M_4 = (\mathsf{sencrypt}((\mathsf{host}(\mathsf{Kas}), M_K), \mathsf{Kbs}), \mathsf{sencrypt}(M_{N_b}, M_K))$$
$$M_5 = \mathsf{sencrypt}(\mathsf{secretA}, M_K)$$

with $M_{N_b} = \mathsf{Nb}(\mathsf{host}(\mathsf{Kas}), \mathsf{Na})$ and $M_K = \mathsf{Kab}(\mathsf{Kas}, \mathsf{Kbs}, \mathsf{Na}, M_{N_b})$. It is easy to check that the clauses (Msg1)–(Msg5) satisfy the condition C1.

Condition C2 models that each name function symbol is created at a unique occurrence in the protocol. Condition C3 means that, in its intended behaviour, the protocol uses public-key encryption only with public keys.

Definition 3 (Plain protocol program). *A plain protocol program is a protocol program $\mathcal{R}_{\mathrm{All}}$ with associated set of closed terms S_0, such that:*

C4. The only constructors and destructors are those of Figure 1, plus host.
C5. The arguments of pk *and* host *in S_0 are atomic constants.*

Condition C5 essentially means that the protocol only uses pairs of atomic keys for public key cryptography, and atomic keys for long-term secret keys.

Tagging a protocol is a simple syntactic annotation of messages. We add a tag to each application of a primitive $\mathsf{sencrypt}$, $\mathsf{pencrypt}$, sign, $\mathsf{nmrsign}$, hash, mac, such that two applications of the same primitive with the same tag have the same parameters. For example, after tagging the Yahalom protocol becomes:

Message 1. $A \to B : (A, N_a)$
Message 2. $B \to S : (B, \{\mathsf{c}_1, A, N_a, N_b\}_{K_{bs}})$
Message 3. $S \to A : (\{\mathsf{c}_2, B, K_{ab}, N_a, N_b\}_{K_{as}}, \{\mathsf{c}_3, A, K_{ab}\}_{K_{bs}})$
Message 4. $A \to B : (\{\mathsf{c}_3, A, K_{ab}\}_{K_{bs}}, \{\mathsf{c}_4, N_b\}_{K_{ab}})$

If the original protocol translates to a plain protocol program, its tagged version translates to a tagged protocol program, as defined below.

Definition 4 (Tagged protocol program). *A tagged protocol program is a plain protocol program $\mathcal{R}_{\mathrm{All}}$ with associated set of closed terms S_0 such that:*

C6. If $f \in \{\mathsf{sencrypt}, \mathsf{pencrypt}, \mathsf{sign}, \mathsf{nmrsign}, \mathsf{hash}, \mathsf{mac}\}$ occurs in a term in S_0 or in $terms(R)$ for $R \in \mathcal{R}_{\mathrm{Prot}}$, then its first argument is the tuple $(\mathsf{c}, M_1, \ldots, M_n)$ for some constant c and terms $M_1, \ldots, M_n$.
C7. Every two subterms of terms in S_0 of the form $f((\mathsf{c}, \ldots), \ldots)$ with the same primitive $f \in \{\mathsf{sencrypt}, \mathsf{pencrypt}, \mathsf{sign}, \mathsf{nmrsign}, \mathsf{hash}, \mathsf{mac}\}$ and the same tag c are identical.

The condition that constant tags appear in $terms(R)$ (Condition C6) means that honest protocol participants always check the tags of received messages (something that the informal description of a tagged protocol leaves implicit) and send tagged terms. The condition also expresses that the initial knowledge of the attacker consists of tagged terms.

For example, the algorithm does not terminate on the Needham-Schroeder shared-key protocol without tags. It terminates after adding tags. In this protocol, we have two messages of the form:

$$\text{Message 4. } B \to A : \{N_B\}_K$$
$$\text{Message 5. } A \to B : \{N_B - 1\}_K$$

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

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

$$H \wedge \mathsf{att}(\mathsf{sencrypt}(n, k)) \to \mathsf{att}(\mathsf{sencrypt}(\mathsf{minusone}(n), k))$$

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

$$\mathsf{att}(\mathsf{sencrypt}(n, K)) \to \mathsf{att}(\mathsf{sencrypt}(\mathsf{minusone}(n), K)) \qquad \text{(Loop)}$$

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

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

$$\mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_1, n), K)) \to \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_2, \mathsf{minusone}(n)), K)) \qquad \text{(NoLoop)}$$

and the previous loop is removed because c_2 does not unify with c_1. The fact $\mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_2, \mathsf{minusone}(N_B)), K))$ is derived, but this does not yield a loop.

Figure 2 shows a number of examples of protocols from the literature. The verifier does not terminate on two protocols, Needham-Schroeder shared key and

Protocol	Tagging	Time (ms)
Needham-Schroeder shared key [24]	Original	Non-termination
Needham-Schroeder shared key	Tagged	27
Woo-Lam shared key [31]	Original	Non-termination
Woo-Lam shared key	Tagged	7
Woo-Lam shared key [33]	Original	8
Woo-Lam shared key corrected	Tagged	7
Needham-Schroeder public key [24]	Original	51
Denning-Sacco [14]	Original	12
Woo-Lam public key [31]	Original	6
Yahalom [8]	Original	21
Otway-Rees [26]	Original	29

Fig. 2. Tagging and termination of the algorithm for a few protocols

Woo-Lam shared key, which are not tagged, and terminates after addition of tags to these protocols. One can also notice that it terminates on many protocols without tags. We partly explain this constatation in Remark 1 below. For the Woo-Lam shared key protocol of [33], adding tags also corrects a flaw in the protocol, because this protocol is subject to a type flaw attack.

5 Termination Proof

Instead of giving the termination proof in one big step, we first consider a special case (Section 5.1), and then describe the modification of the first proof that yields the proof for the general case (Section 5.2).

The special case is defined in terms of the sets $Params_{\mathsf{pk}}$ and $Params_{\mathsf{host}}$ of arguments of pk resp. host in S_0, namely by the condition that these sets each have at most one element.

This restriction is meaningful in terms of models of protocols: it corresponds to merging several keys. In the example of the Yahalom protocol, this means that, in the clauses, the keys Kas and Kbs should be replaced with a single key, k_0 (so the host names $A = \mathsf{host}(\mathsf{Kas})$ and $B = \mathsf{host}(\mathsf{Kbs})$ are replaced with a single name $\mathsf{host}(\mathsf{k}_0)$). When studying secrecy, merging all keys of honest hosts in this way helps to model cases in which one host plays several roles in the protocol. The secrecy for the clauses with merged keys implies secrecy for the protocol without merged keys. However, this merging is not acceptable for authenticity [5]. This is why we also consider the general case in Section 5.2.

5.1 The Special Case of One Key

We now define weakly tagged programs by the conditions that we use in the first termination proof. In the special case, these conditions are strictly more general than tagged protocol programs. This plays a role to deduce termination for protocols that are not explicitly tagged (see Remark 1).

A term is said to be *non-data* when it is not of the form $f(\ldots)$ with f in *DataConstr*. The set $sub(S)$ contains the subterms of terms in the set S.

The set *tagGen* contains the non-variable non-data subterms of terms of clauses in $\mathcal{R}_{\mathrm{Prot}}$ and of terms $M_1, \ldots, M_n$ in clauses of the form $\mathsf{att}(f(M_1, \ldots, M_n)) \wedge \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_m) \rightarrow \mathsf{att}(x)$ in $condense(\mathcal{R}_{\mathrm{CryptoDestr}})$ (this is the form required in W1 below). This set summarizes the terms that appear in the clauses and that should be tagged.

Definition 5 (Weakly tagged programs). *A program $\mathcal{R}_{\mathrm{All}}$ of the form $\mathcal{R}_{\mathrm{All}} = \mathcal{R}_{\mathrm{Primitives}} \cup \mathcal{R}_{\mathrm{Prot}}$ (where $\mathcal{R}_{\mathrm{Primitives}}$ is a program for primitives) is weakly tagged if there exists a finite set of closed terms S_0 such that:*

W1. All clauses in the set $\mathcal{R}'_{\mathrm{CryptoDestr}} = condense(\mathcal{R}_{\mathrm{CryptoDestr}})$ are of the form

$$\mathsf{att}(f(M_1, \ldots, M_n)) \wedge \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_m) \rightarrow \mathsf{att}(x)$$

where $f \in CryptoConstr$, x is one of $M_1, \ldots, M_n$, and $f(M_1, \ldots, M_n)$ is more general than every term of the form $f(\ldots)$ in $sub(S_0)$.

W2. For all clauses R in $\mathcal{R}_{\mathrm{Prot}}$, there exists a substitution σ such that $terms(\sigma R) \subseteq S_0$.

W3. If two terms M_1 and M_2 in tagGen unify, N_1 is an instance of M_1 in $sub(S_0)$, and N_2 is an instance of M_2 in $sub(S_0)$, then $N_1 = N_2$.

Condition W3 is the key of the termination proof. We are going to show the following invariant: all terms in the generated clauses are instances of terms in *tagGen* and have instances in $sub(S_0)$. This condition makes it possible to prove that, when unifying two terms satisfying the invariant, the result of the unification also satisfies the invariant; this is because the instances in $sub(S_0)$ of those two terms are in fact equal. Condition W1 guarantees that this continues to hold if only one of the two terms satisfies the invariant and the other stems from a clause in $\mathcal{R}'_{\mathrm{CryptoDestr}}$.

Proposition 1. *A tagged protocol program where $Params_{\mathsf{host}}$ and $Params_{\mathsf{pk}}$ each have at most one element, is weakly tagged.*

Proof. For condition W1, the clauses for sdecrypt, pdecrypt, and getmessage are:

$$\mathsf{att}(\mathsf{sencrypt}(x, y)) \wedge \mathsf{att}(y) \rightarrow \mathsf{att}(x) \qquad (\mathsf{sdecrypt})$$
$$\mathsf{att}(\mathsf{pencrypt}(x, \mathsf{pk}(y))) \wedge \mathsf{att}(y) \rightarrow \mathsf{att}(x) \qquad (\mathsf{pdecrypt})$$
$$\mathsf{att}(\mathsf{sign}(x, y)) \rightarrow \mathsf{att}(x) \qquad (\mathsf{getmessage})$$

and they satisfy condition W1 provided that all public-key encryptions in S_0 are of the form $\mathsf{pencrypt}(M_1, \mathsf{pk}(M_2))$ (that is C3). The clauses for checksignature and nmrchecksign are

$$\mathsf{att}(\mathsf{sign}(x, y)) \wedge \mathsf{att}(\mathsf{pk}(y)) \rightarrow \mathsf{att}(x) \qquad (\mathsf{checksignature})$$
$$\mathsf{att}(\mathsf{nmrsign}(x, y)) \wedge \mathsf{att}(\mathsf{pk}(y)) \wedge \mathsf{att}(x) \rightarrow \mathsf{att}(\mathsf{true}) \qquad (\mathsf{nmrchecksign})$$

These two clauses are subsumed respectively by the clauses for $\mathsf{getmessage}$ (given above) and true (which is simply $\mathsf{att}(\mathsf{true})$ since true is a zero-ary constructor), so they are eliminated by *condense*, i.e., they are not in $\mathcal{R}'_{\mathrm{CryptoDestr}}$. (This is important, because they do not satisfy condition W1.)

Condition W2 is identical to condition C1. We now prove condition W3. Let

$$\begin{aligned} S_1 = \{&f((\mathsf{c}_i, x_1, \dots, x_n), x'_2, \dots, x'_{n'}) \mid \\ &\quad f \in \{\mathsf{sencrypt}, \mathsf{pencrypt}, \mathsf{sign}, \mathsf{nmrsign}, \mathsf{hash}, \mathsf{mac}\}\} \\ &\cup \{a(x_1, \dots, x_n) \mid a \text{ name function symbol}\} \\ &\cup \{\mathsf{pk}(x), \mathsf{host}(x)\} \cup \{c \mid c \text{ atomic constant}\} \end{aligned}$$

By condition C4, the only term in *tagGen* that comes from clauses of $\mathcal{R}'_{\mathrm{CryptoDestr}}$ is $\mathsf{pk}(x)$. Using condition C6, all terms in *tagGen* are instances of terms in S_1 (noticing that *tagGen* does not contain variables). Using conditions C2, C5, C7, and the fact that $\mathit{Params}_{\mathsf{pk}}$ and $\mathit{Params}_{\mathsf{host}}$ have at most one element, each term in S_1 has at most one instance in $\mathit{sub}(S_0)$.

If M_1 and M_2 in *tagGen* unify, they are both instances of the same element M' in S_1 (since different elements of S_1 do not unify with each other). Let N_1 and N_2 be any instances of M_1 and M_2 (respectively) in $\mathit{sub}(S_0)$. Then N_1 and N_2 are instances of $M' \in S_1$ in $\mathit{sub}(S_0)$ so $N_1 = N_2$. Thus we obtain W3. □

Remark 1. Some protocols are in fact weakly tagged without explicitly adding constant tags. For instance, this is true for the Woo and Lam public key protocol (after merging the keys sk_A and sk_S):

Message 1.	$A \to B :$	A
Message 2.	$B \to A :$	N
Message 3.	$A \to B :$	$\{A, B, N\}_{sk_A}$
Message 4.	$B \to S :$	A
Message 5.	$S \to B :$	$A, \{A, pk_A\}_{sk_S}$

Indeed, since different encryptions in the protocol have a different arity, we can take $\mathsf{pencrypt}((x_1, \dots, x_n), x')$ in S_1 in the proof above, and use the same reasoning as above to prove the condition W3. This shows both that type flaws cannot happen in the original protocol, and that the algorithm also terminates on the original protocol. We can say that the protocol is "implicitly tagged": the arity replaces the tag. This situation happens in some other examples: In figure 2, this is also the case of the Denning-Sacco protocol. This can partly explain why the algorithm often terminates even for protocols without explicit tags.

A term is *top-tagged* when it is an instance of a term in *tagGen*. Intuitively, referring to the case of explicit constant tags, top-tagged terms are terms whose top function symbol is tagged. A term is *fully tagged* when all its non-variable non-data subterms are top-tagged.

We next show the invariant that all terms in the generated clauses are non-data, fully tagged, and have instances in $\mathit{sub}(S_0)$. Using this invariant, we show that the size of an instance in $sub(S_0)$ of a clause obtained by resolution from R and R' is smaller than the size of an instance of R or R' in $sub(S_0)$. This implies the termination of the algorithm.

Let us define the size of a term M, $size(M)$, as usual, and the size of a clause by $size(\mathsf{att}(M_1)\wedge\ldots\wedge\mathsf{att}(M_n) \rightarrow \mathsf{att}(M)) = size(M_1)+\ldots+size(M_n)+size(M)$. The hypotheses of clauses form a multiset, so when we compute $size(\sigma R)$ and the substitution σ maps several hypotheses to the same fact, this fact is counted several times in *size*. Intuitively, the size of clauses can increase during resolution, because the unification can instantiate terms. However, the size of their corresponding closed instance in $sub(S_0)$ decreases.

Proposition 2. *Assuming a weakly tagged program (Definition 5) and* $\mathcal{R}_0 = \mathcal{R}_{\text{CryptoConstr}} \cup \mathcal{R}_{\text{CryptoDestr}} \cup \mathcal{R}_{\text{Prot}}$, *the computation of* $saturate(\mathcal{R}_0)$ *terminates.*

Proof. We show by induction that all rules R generated from $\mathcal{R}_0$ either are in $\mathcal{R}_{\text{CryptoConstr}} \cup \mathcal{R}'_{\text{CryptoDestr}}$, or are such that the terms of R are non-data, fully tagged, and mapped to $sub(S_0)$ by a substitution σ, i.e., $terms(\sigma R) \subseteq sub(S_0)$.

First, we can easily show that all rules in $condense(\mathcal{R}_0)$ satisfy this property.

If we combine by resolution two rules in $\mathcal{R}_{\text{CryptoConstr}} \cup \mathcal{R}'_{\text{CryptoDestr}}$, we in fact combine one rule of $\mathcal{R}_{\text{CryptoConstr}}$ with one rule of $\mathcal{R}_{\text{CryptoDestr}}$. The resulting rule is a tautology by condition W1, so it is eliminated immediately.

Otherwise, we combine by resolution a rule R such that the terms of R are non-data and fully tagged, and there exists a substitution σ such that $terms(\sigma R) \subseteq sub(S_0)$, with a rule R' such that one of 1., 2., or 3. holds.

1. The terms of R' are non-data and fully tagged, there exists a substitution σ' such that $terms(\sigma' R') \subseteq sub(S_0)$, and $sel(R') = \emptyset$ (in which case $sel(R) \neq \emptyset$).
2. $R' \in \mathcal{R}_{\text{CryptoConstr}}$.
3. $R' \in \mathcal{R}'_{\text{CryptoDestr}}$.

Let R'' be the rule obtained by resolution of R and R'. We show that the terms of R'' are fully tagged, and there exists a substitution σ'' such that $terms(\sigma'' R'') \subseteq sub(S_0)$ and $size(\sigma'' R'') < size(\sigma R)$.

Let $M_0, \ldots, M_n$ be the terms of R, $\mathsf{att}(M_0)$ being the atom of R on which we resolve. In all cases, the terms of R' are $M', x_1, \ldots, x_{n'}$, the variables $x_1, \ldots, x_{n'}$ occur in M' and are pairwise distinct variables, and $\mathsf{att}(M')$ is the atom of R' on which we resolve. (In case 1, because $sel(R') = \emptyset$ and by the optimizations *elimattx* and *elimdup*; in case 2, by definition of constructor rules; in case 3, by W1.) The terms M_0 and M' unify, let σ_u be their most general unifier. Then the terms of R'' are $\sigma_u x_1, \ldots, \sigma_u x_{n'}, \sigma_u M_1, \ldots, \sigma_u M_n$. By the choice of the selection function, the terms M_0 and M' are not variables.

We know that $\sigma M_0, \ldots, \sigma M_n$ are in $sub(S_0)$. We show that there exists σ' such that $\sigma M_0 = \sigma' M'$.

- In case 1, there exists σ' such that $\sigma' M' \in sub(S_0)$. The terms M_0 and M' are non-data fully tagged, so all their non-variable non-data subterms are top-tagged. In particular, since they are not variables, M_0 and M' themselves are top-tagged, i.e., M_0 is an instance of some $N_0 \in \mathit{tagGen}$ and M' is an instance of some $N'_0 \in \mathit{tagGen}$. Since M_0 and M' unify, so do N_0 and N'_0, $\sigma' M'$ is an instance of N'_0 in $\mathit{sub}(S_0)$, σM_0 is an instance of N_0 in $\mathit{sub}(S_0)$, so by condition W3, $\sigma' M' = \sigma M_0$.

- In case 2, M' is of the form $f(x_1, \ldots, x_{n'})$. Since M_0 is not a variable and unifies with M', M_0 has root symbol f, so σM_0 is an instance of M'.
- In case 3, by condition W1, M' is more general than every term in $\mathit{sub}(S_0)$ with the same root symbol, hence the instance σM_0 of the term M_0 that is unifiable with M' and thus has the same root symbol.

The substitution equal to σ on the variables of R and to σ' on the variables of R' is then a unifier of M_0 and M'. Since σ_u is the most general unifier, there exists σ'' such that $\sigma''\sigma_u$ is equal to σ on the variables of R, and to σ' on the variables of R'. Thus the terms of $\sigma'' R''$ are $\sigma' x_1, \ldots, \sigma' x_{n'}, \sigma M_1, \ldots, \sigma M_n$. The terms $\sigma' x_1, \ldots, \sigma' x_{n'}$ are subterms of $\sigma' M' = \sigma M_0$ which is in $\mathit{sub}(S_0)$, so they are also in $\mathit{sub}(S_0)$. So all terms of $\sigma'' R''$ are in $\mathit{sub}(S_0)$.

Moreover, $\mathit{size}(\sigma'' R'') < \mathit{size}(\sigma R)$. Indeed, $x_1, \ldots, x_{n'}$ occur in M' and are different variables. So $\sigma' x_1, \ldots, \sigma' x_{n'}$ are disjoint subterms of $\sigma' M'$, and M' does not consist of only a variable, so $\mathit{size}(\sigma' x_1) + \ldots + \mathit{size}(\sigma' x_{n'}) < \mathit{size}(\sigma' M') = \mathit{size}(\sigma M_0)$, and $\mathit{size}(\sigma'' R'') < \mathit{size}(\sigma M_0) + \ldots + \mathit{size}(\sigma M_n) = \mathit{size}(\sigma R)$.

We show that the terms of R'' are fully tagged.

- In case 1, since σ_u is the most general unifier of fully tagged terms, we can show that, for all x, $\sigma_u x$ is fully tagged, so for all fully tagged terms M, we can show that $\sigma_u M$ is fully tagged, so the terms of R'' are fully tagged.
- In case 2, for x among $x_1, \ldots, x_{n'}$, $\sigma_u x$ is a subterm of M_0, so is fully tagged. The terms $\sigma_u M_1, \ldots, \sigma_u M_n$ are equal to $M_1, \ldots, M_n$, also fully tagged.
- In case 3, $M' = f(M'_1, \ldots, M'_m)$ and $M_0 = f(M''_1, \ldots, M''_m)$, so σ_u is also the most general unifier of the pairs (M'_1, M''_1), $\ldots$, (M'_m, M''_m) of fully tagged terms. So we conclude as in case 1.

Finally, the terms of R'' are fully tagged, $\mathit{terms}(\sigma'' R'') \subseteq \mathit{sub}(S_0)$, and $\mathit{size}(\sigma'' R'') < \mathit{size}(\sigma R)$.

Then it is easy to show that all rules $R_{\mathrm{s}} \in \mathit{simplify}(R'')$ obtained after simplification of R'' have non-data fully tagged terms and satisfy $\mathit{terms}(\sigma'' R_{\mathrm{s}}) \subseteq \mathit{sub}(S_0)$, and $\mathit{size}(\sigma'' R_{\mathrm{s}}) < \mathit{size}(\sigma R)$. Indeed, all rules in $\mathit{decomp}(R'')$ satisfy this property. (The decomposition of data constructors transforms fully tagged terms into non-data fully tagged terms.) This property is preserved by *elimdup* and *elimattx*.

Therefore, for all generated rules R, there exists σ such that $\mathit{size}(\sigma R)$ is smaller than the maximum initial value of $\mathit{size}(\sigma R)$ for a rule of the protocol. There is a finite number of such rules (since $\mathit{size}(R) \leq \mathit{size}(\sigma R)$). So the algorithm terminates. $\square$

The termination of the backward depth-first search for closed facts is easy to show, for example by a proof similar to that of [4]. Essentially, the size of the goal decreases, because the size of the hypotheses of each clause is smaller than the size of the conclusion. (Recall that all terms of hypotheses of clauses of $saturate(\mathcal{R}_0) \cup \mathcal{R}_{\mathrm{DataConstr}}$ are variables that occur in the conclusion.) So we obtain:

Theorem 1. *The resolution-based verification algorithm terminates for weakly tagged programs and closed facts.*

As a corollary, by Proposition 1, we obtain the same result for tagged protocol programs, when $Params_{\mathsf{host}}$ and $Params_{\mathsf{pk}}$ have at most one element.

5.2 Handling Several Keys

The extension to several arguments of pk or of host requires an additional step. We define a homomorphism h from terms to terms that replaces all elements of $Params_{\mathsf{pk}}$ and of $Params_{\mathsf{host}}$ with a special constant k_0. We extend h to facts, clauses, and sets of clauses naturally. For the protocol program $h(\mathcal{R}_{\mathrm{Prot}})$, $Params_{\mathsf{pk}}$ and $Params_{\mathsf{host}}$ each have at most one element. So by Proposition 1, when $\mathcal{R}_{\mathrm{Prot}}$ is a tagged protocol program, $h(\mathcal{R}_{\mathrm{Prot}})$ is a weakly tagged program.

Let $\mathcal{R}_{\mathrm{Prot}}$ be any program such that $h(\mathcal{R}_{\mathrm{Prot}})$ is a weakly tagged program. We consider a "less optimized algorithm" in which elimination of duplicate hypotheses and of tautologies are performed only for facts of the form $\mathsf{att}(x)$ and elimination of subsumed clauses is performed only for the condensing of rules of $\mathcal{R}_{\mathrm{CryptoDestr}}$. We observe that Theorem 1 holds also for the less optimized algorithm, with the same proof, so this algorithm terminates on $h(\mathcal{R}_{\mathrm{Prot}})$. All resolution steps possible for the less optimized algorithm applied to $\mathcal{R}_{\mathrm{Prot}}$ are possible for the less optimized algorithm applied to $h(\mathcal{R}_{\mathrm{Prot}})$ as well (more terms are unifiable, and the remaining optimizations of the less optimized algorithm commute with the application of h). Then the less optimized algorithm terminates on $\mathcal{R}_{\mathrm{Prot}}$. We can show that then the original, fully optimized algorithm also terminates.

In particular, the algorithm terminates for all tagged protocol programs and for implicitly tagged protocols, such as the Woo and Lam public key and Denning-Sacco protocols without tags by Remark 1.

Theorem 2. *The resolution-based verification algorithm terminates for tagged protocol programs and closed facts.*

We recall that a tagged protocol program may be obtained by translating a protocol after tagging, and that the algorithm checks the non-derivability of the closed fact $\mathsf{att}(M)$, which shows the secrecy of the message M.

6 Complexity

We will next show that the algorithm performs in exponential time in the size of the input protocol. We need to define what we mean by the size of the input protocol. Formally, the input of the algorithm is not the informal description of a protocol; instead, it is its translation in the form of a set of Horn clauses.

The set of Horn clauses by itself, however, is not sufficient to define the input size. Remember that the terms in the Horn clauses represent only a partial information about the messages being sent (the information that the participants have). The full information about the set of 'intended' messages, which is present in the informal description of the input protocol, is formalized in a protocol program. A protocol program (see Definition 2) consists of a set of Horn clauses $\mathcal{R}_{\mathrm{All}}$ together with the set S_0 of closed terms formalizing the intended messages. The set $\mathcal{R}_{\mathrm{All}}$ consists of the set $\mathcal{R}_{\mathrm{Prot}}$ of protocol clauses translating the protocol and the set $\mathcal{R}_{\mathrm{Primitives}}$, the program for primitives, which is fixed by the signature of the protocol. Therefore, we define that the parameter for the input size is the size of $\mathcal{R}_{\mathrm{Prot}}$ and S_0 (the size of a set of clauses and terms is the total size of all their terms).

The cost analysis of our algorithm does not change if we exchange its input parameter against one that is polynomially smaller or larger. Below we define the parameter n for the input size; n is the maximal size of a possible instance σR of a protocol clause R such that the terms in σR are subterms of S_0.

$$n \;=\; max\{size(\sigma R) \mid R \in \mathcal{R}_{\mathrm{Prot}},\; terms(\sigma R) \subseteq sub(S_0)\} \tag{1}$$

The size of $\mathcal{R}_{\mathrm{Prot}}$ and S_0 is at most quadratic in that number.

The parameter n is smaller than the size of the messages in the informal description of the protocol and in the initial knowledge of the attacker, after replacing each name by the corresponding term that we introduce by skolemization.

Theorem 3. *If a protocol is translated to a protocol program with the set $\mathcal{R}_{\mathrm{Prot}}$ of protocol clauses in $\mathcal{R}_0$ and the set S_0 of intended messages, the resolution-based verification algorithm is exponential in the size of $\mathcal{R}_{\mathrm{Prot}}$ and S_0.*

Equivalently, the algorithm is exponential in the input parameter n defined in (1).

Proof. We define the set $\mathcal{R}$ of all clauses R derived from the set $\mathcal{R}_{\mathrm{Prot}}$ of protocol clauses (and the other clauses in $\mathcal{R}_0$, which are part of $\mathcal{R}_{\mathrm{Primitives}}$, the program for primitives, fixed by the signature of the protocol). By the proof of Proposition 2 (see the last paragraph of the proof), the size of each such clause R is smaller than the size of a possible instance $\sigma R'$ of a protocol clause R' such that the terms in $\sigma R'$ are subterms of S_0.

$$\mathcal{R} \subseteq \{R \mid size(R) \leq n\}$$

Given n, there are exponentially many clauses R whose size is smaller than n. Thus, the number of clauses in $\mathcal{R}$ is exponential; so the cost of the algorithm is also exponential. □

Exponential Example To show that our cost estimation for the algorithm is not too conservative, we give an example of a tagged protocol program on which the algorithm is indeed exponential.

$$\mathsf{att}(\mathsf{c}_i), i \in \{1, \dots, n\}$$
$$\mathsf{att}(\mathsf{k}_0)$$
$$\mathsf{att}(x) \wedge \mathsf{att}(y) \to \mathsf{att}(\mathsf{sencrypt}(x, y)) \qquad (\mathsf{sencrypt})$$
$$\mathsf{att}(x) \to \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_i, a(x)), \mathsf{k}_0)), i \in \{1, \dots, n\} \qquad (\text{Msg } i)$$
$$\mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_1, x_1), \mathsf{k}_0)) \wedge \dots \wedge \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_n, x_n), \mathsf{k}_0)) \to \mathsf{att}(b(x_1, \dots, x_n)) \qquad (\text{Last msg})$$

This example contains $2n + 3$ rules, and is of size $\mathcal{O}(n)$. The selection is empty for the first four rules. It is not empty for the last rule. Without loss of generality, we assume that the first hypothesis is selected in the last rule. Then, the last rule can resolve with the rules (sencrypt) and (Msg i). If it resolves with (sencrypt), we obtain

$$\begin{aligned}&\mathsf{att}(\mathsf{c}_1) \wedge \mathsf{att}(x_1) \wedge \mathsf{att}(\mathsf{k}_0) \wedge \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_2, x_2), \mathsf{k}_0)) \\ &\quad \wedge \dots \wedge \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_n, x_n), \mathsf{k}_0)) \to \mathsf{att}(b(x_1, x_2, \dots, x_n))\end{aligned}$$

Depending on the selection, the hypotheses $\mathsf{att}(\mathsf{c}_1)$ and $\mathsf{att}(\mathsf{k}_0)$ might be removed by resolving with one of the first two rules. In all cases, a fact $\mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_i, x_i), \mathsf{k}_0))$ finally gets selected, and we come back to the situation of rule (Last msg) with one hypothesis less.

If it resolves with (Msg i), we obtain

$$\begin{aligned}&\mathsf{att}(x_1') \wedge \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_2, x_2), \mathsf{k}_0)) \\ &\quad \wedge \dots \wedge \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_n, x_n), \mathsf{k}_0)) \to \mathsf{att}(b(a(x_1'), x_2, \dots, x_n))\end{aligned}$$

A fact $\mathsf{att}(\mathsf{sencrypt}((\mathsf{c}_i, x_i), \mathsf{k}_0))$ gets selected, and we also come back to the situation of rule (Last msg) with one hypothesis less (and with $a(x_1')$ instead of x_1 in the conclusion).

In the end, we obtain 2^n rules, of the form

$$H_1 \wedge \dots \wedge H_n \to \mathsf{att}(b(M_1, \dots, M_n))$$

where for each i in $1, \dots, n$, either $H_i = \mathsf{att}(x_i')$ and $M_i = a(x_i')$, or $H_i = \mathsf{att}(x_i)$ (and perhaps $\mathsf{att}(\mathsf{c}_i)$ and/or $\mathsf{att}(\mathsf{k}_0)$) and $M_i = x_i$. (Intuitively, these 2^n rules must be generated because no other rules can be used to derive the 2^n facts $\mathsf{att}(b(M_1, \dots, M_n))$ where M_i is either $a(\mathsf{k}_0)$ or k_0.)

The algorithm is quite efficient in our examples. It would be interesting to know whether there exists a natural class of 'realistic' protocol programs for which the algorithm is polynomial.

7 Extension to Authenticity

In [5], the Horn clause verification technique has been extended to verify authenticity properties of protocols, specified by correspondence assertions [32].

For simplicity, we focus here on the case of non-injective agreement [22]: The protocol can execute events $\mathtt{begin}(M)$ and $\mathtt{end}(M)$, and one wants to show that if the protocol executes $\mathtt{end}(M)$, then it must have executed $\mathtt{begin}(M)$ for the same value of M.

The proof technique for authenticity involves two extensions to the Horn clause model:

- Fresh names are functions not only of the messages previously received but also of a *session identifier*: an argument that takes a different value for each session of the protocol. This makes it possible to distinguish different names created after receiving the same messages.
- New predicates are added. A fact $\mathsf{begin}(M)$ means that the event $\mathtt{begin}(M)$ has been executed, while a fact $\mathsf{end}(M)$ means that the event $\mathtt{end}(M)$ may have been executed. The facts $\mathsf{begin}(M)$ can appear only in the hypothesis of clauses, and the selection function never selects them. The facts $\mathsf{end}(M)$ can appear only in the conclusion of clauses.

The system shows that, for any closed term M, to prove a fact $\mathsf{end}(M)$, a fact $\mathsf{begin}(M)$ must be present in the hypotheses. As shown in [5], this implies that, for all closed terms M, if the protocol executes $\mathtt{end}(M)$, then it must have executed $\mathtt{begin}(M)$. To prove this, we modify the selection function such that it never selects begin facts:

$$sel(H \to C) = \begin{cases} \emptyset & \text{if all elements of } H \text{ are of the form } \mathsf{att}(x) \text{ or } \mathsf{begin}(M) \\ \{F\} & \text{where } F \neq \mathsf{att}(x),\ F \neq \mathsf{begin}(M), \text{ and } F \in H, \text{ otherwise} \end{cases}$$

and we show that all clauses in $saturate(\mathcal{R}_0)$ whose conclusion is $\mathsf{end}(M')$ for some M' contain $\mathsf{begin}(M')$ in their hypothesis.

We can extend our termination result for tagged protocols to authenticity proofs. We add the following condition to the definition of protocol programs and of weakly tagged programs:

C1'. For all clauses R in $\mathcal{R}_{\mathrm{Prot}}$, all variables that occur in the conclusion of R also occur in att facts in its hypothesis, except session identifiers. Session identifiers occur only at specific positions as arguments of names function symbols $a(\ldots)$, and only session identifiers occur at those positions.

This condition is true for all protocols when Horn clauses are built as explained in [5]. We also add the following optimization to *simplify*:

- Elimination of useless begin facts: *elimbegin* eliminates begin facts in which a variable x occurs, and x only occurs in begin facts and in $\mathsf{att}(x)$ hypotheses.

This optimization is always sound, because it creates a stronger clause. It does not lead to a loss of precision in the case of authenticity. Indeed, assume that $\mathsf{begin}(M)$ contains a variable which does not occur in the conclusion. This is preserved by resolution, so when we obtain a clause $\mathsf{begin}(M') \wedge H \to \mathsf{end}(M'')$, where $\mathsf{begin}(M')$ comes from $\mathsf{begin}(M)$, M' cannot be equal to M'', so this occurrence of $\mathsf{begin}(M')$ cannot be used to prove authenticity. However, in the more general case when we want to know which `begin` events must be executed to reach a given `end` event, this optimization leads to a loss of precision (it may miss some `begin` events). That is why this optimization was present in early implementations which verified only authenticity, and was later abandoned. We could consider reintroducing it when testing authenticity, if we had termination problems on practical examples, which was not the case up to now.

Note that, after this optimization, the fact $\mathsf{att}(x)$ will also be eliminated by *elimattx*. Thanks to this optimization, in a clause R such that $sel(R) = \emptyset$, all variables that occur in the hypothesis also occur in the conclusion. Indeed, if a variable occurs only in the hypothesis, either it occurs in a begin fact, and this fact is eliminated by *elimbegin*, or it occurs only in $\mathsf{att}(x)$, and this fact is eliminated by *elimattx*.

Theorem 4. *The resolution-based verification algorithm that proves authenticity terminates for weakly tagged programs and for tagged protocol programs.*

Proof. As explained at the end of Sect. 3, the conclusion of a rule R is not of the form $\mathsf{att}(x)$ when $sel(R) = \emptyset$. The presence of begin facts complicates the proof of this point. This is why Condition C1' is useful. We can show that Condition C1' is in fact true for all clauses generated by the algorithm (it is preserved by resolution).

Using this property, if $sel(R) = \emptyset$, then the conclusion of R is not $\mathsf{att}(x)$. Indeed, if the conclusion of R was $\mathsf{att}(x)$, x is not a session identifier, since it is not an argument of a name function symbol, so x would occur in the att facts in the hypothesis of R. Since $sel(R) = \emptyset$, all these hypotheses are $\mathsf{att}(x_i)$ for some variables x_i, then one of the hypotheses would be $\mathsf{att}(x)$, so R would be a tautology, so it would have been removed by *elimtaut*. This proves the result.

So we still have the property that facts $\mathsf{att}(x)$ are never unified during the resolution. The proof of Prop. 2 then needs only minor changes. In case 1, the terms of R' are $M', x_1, \ldots, x_{n'}, M'_1, \ldots, M'_m$, where we resolve on $\mathsf{att}(M')$ and R' has hypotheses $\mathsf{att}(x_1)$, …, $\mathsf{att}(x_{n'})$, $\mathsf{begin}(M'_1)$, …, $\mathsf{begin}(M'_m)$. Thanks to the optimization *elimbegin*, $x_1, \ldots, x_{n'}$ occur in M'. By hypothesis, there exists a substitution σ' such that $\sigma' M'$, $\sigma' x_1$, …, $\sigma' x_{n'}$, $\sigma' M'_1, \ldots, \sigma' M'_m$ are in $sub(S_0)$. As in the proof of Prop. 2, we build σ'', such that the terms of $\sigma'' R''$ are $\sigma' x_1$, …, $\sigma' x_{n'}$, $\sigma' M'_1$, …, $\sigma' M'_m$, σM_1, …, σM_n. By definition of σ and σ', these terms are in $sub(S_0)$.

We do not count begin facts in the definition of the size of a clause. (Otherwise, we could have $size(\sigma'' R'') \geq size(\sigma R)$ because of begin facts coming from R'.) We show that $size(\sigma'' R'') < size(\sigma R)$ and that the terms of R'' are fully tagged as in Prop. 2. To prove termination, we have to show that there is a finite number of rules with $size(R)$ smaller than a given constant. The size of att facts is bounded by this constant, and the number of variables in begin facts is also bounded by this constant thanks to the optmization *elimbegin*. Then the size of begin facts is also bounded, since they must be more general than terms in $sub(S_0)$. This proves termination.

In the case of authenticity, no depth-first search is performed, so Prop. 2 immediately implies termination for weakly tagged programs. We handle the case of tagged protocol programs with several keys as in the proof of Theorem 2. □

When the optimization *elimbegin* is turned off, we can find examples of tagged protocols on which our algorithm for authenticity does not terminate. For example, consider the clause:

$$\mathsf{begin}(h((\mathsf{c}_1, x, y))) \wedge \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}, x), \mathsf{k})) \wedge \mathsf{att}(y) \rightarrow \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}, y), \mathsf{k})) \quad (2)$$

The selected hypothesis is $\mathsf{att}(\mathsf{sencrypt}((\mathsf{c}, x), \mathsf{k}))$. By the resolution with the constructor clauses and with $\mathsf{att}(\mathsf{k})$, $\mathsf{att}(\mathsf{c})$, we can obtain

$$\mathsf{begin}(h((\mathsf{c}_1, x, y))) \wedge \mathsf{att}(x) \wedge \mathsf{att}(y) \rightarrow \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}, y), \mathsf{k}))$$

This clause has no selected hypothesis and resolves with (2), and the result of this resolution again resolves with (2), so we obtain an infinite sequence of clauses:

$$\begin{aligned} &\mathsf{begin}(h((\mathsf{c}_1, x_1, x_2))) \wedge \ldots \wedge \mathsf{begin}(h((\mathsf{c}_1, x_{n-1}, x_n))) \\ &\quad \wedge \mathsf{att}(x_1) \wedge \ldots \wedge \mathsf{att}(x_n) \rightarrow \mathsf{att}(\mathsf{sencrypt}((\mathsf{c}, x_n), \mathsf{k})) \end{aligned}$$

This example then does not terminate. Note that the clause (2) is not very natural since it creates a term $\mathsf{sencrypt}((\mathsf{c}, y), \mathsf{k})$ from a term $\mathsf{sencrypt}((\mathsf{c}, x), \mathsf{k})$ with the same tag c. One would rather expect the participant to use a new tag when creating a new term. However, this clause can belong to a tagged protocol program. Indeed, we choose S_0 containing $h((\mathsf{c}_1, \mathsf{k}', \mathsf{k}'))$ and $\mathsf{sencrypt}((\mathsf{c}, \mathsf{k}'), \mathsf{k})$ for some constant k', and x and y are instantiated to the same value k' by the substitution that maps terms of clauses to $sub(S_0)$. The clause (2) can also be inferred during the resolution algorithm from more natural clauses coming from tagged protocols, although such a situation did not occur during our experiments.

8 Related Work

The verification problem of cryptographic protocols is undecidable [15], so one either restricts the problem, or approximates it.

Decision procedures have been published for restricted cases. In the case of a bounded number of sessions, for protocols using public-key cryptography with atomic keys and shared-key cryptography, protocol insecurity is NP-complete [29], and decisions procedures appear in [12, 23, 29]. When messages are bounded and no nonces are created, secrecy is DEXPTIME-complete [15]. Strong syntactic restrictions on protocols also yield decidability: [10] for an extension of ping-pong protocols, [3] with a bound on the number of parallel sessions, and restricted matching on incoming messages (in particular, this matching should be linear and independent of previous messages). Model-checking also provides a decision technique for a bounded number of sessions [21] (with additional conditions). It has been extended, with approximations, to an unbounded number of sessions using data independence techniques [6, 7, 28], for sequential runs, or when the agents are "factorisable". (Essentially, a single run of the agent has to be split into several runs, such that each run contains only one fresh value.)

On the other hand, some analyses terminate for all protocols, but at the cost of approximations. For instance, control-flow analysis [25] runs in cubic time, but does not preserve relations between components of messages, hence introduces an important approximation. Interestingly, the proof that control flow analysis runs in cubic time also relies on the study of a particular class of Horn clauses. Techniques using tree automata [17] and rank functions [20] also provide a terminating but approximate analysis. Moreover, the computation algorithm of rank functions assumes atomic keys.

It has been shown in [19] that tagging prevents type flaw attacks. It may be possible to infer from [19] that the depth of closed terms can be bounded in the search for an attack. This yields the decidability by exhaustive search, but does not imply the termination of our algorithm (in particular, because clauses can have an unbounded number of hypotheses, so there is an infinite number of clauses with a bounded term depth). [18] shows that two protocols are independent when messages of different protocols cannot be mixed (i.e. unified). Tagging relies on the same idea: it prevents mixing a message with another, here in the same protocol, and this is the key of our termination proof.

As for the approach based on Horn clauses, Weidenbach [30] already gave conditions under which his algorithm terminates. These conditions may give some idea of why the algorithm terminates on protocols. They do not seem to apply to many examples of cryptographic protocols. Comon and Cortier [11] show that an algorithm using ordered binary resolution, ordered factorization and splitting terminates on protocols which "blindly copy" at most one term in each message. A blind copy happens when a participant sends back part of a message he received without looking at what is contained inside this part. In contrast, our result puts no limit on the number of blind copies, but requires tagging.

Other techniques such as theorem proving [27] in general require human intervention, even if some cases can be proved automatically [9, 13]. In general, typing [1, 16] requires human intervention in the form of type annotations, that can be automatically checked. The idea of tagging already appears in [16] in a different context (tagged union types).

9 Conclusion

We have given the theory behind an experimental observation: tagging a protocol enforces the termination of the resolution-based verification technique used. Our work has an obvious consequence to protocol design, namely when one agrees that a design choice in view of a-posteriori verification is desirable.

Our termination result for *weakly* tagged protocols explains only in part another experimental observation, namely the termination for protocols without explicit tags. Although many of those are weakly tagged, some of them are not (for instance, the Needham-Schroeder public key protocol). The existence of a termination condition that applies also to those cases is open.

References

1. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th ACM Symp. on Principles of Programming Languages (POPL'02)*, pages 33–44, Jan. 2002.
2. M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.
3. R. Amadio and W. Charatonik. On Name Generation and Set-Based Analysis in the Dolev-Yao Model. In *CONCUR'02 - Concurrency Theory*, volume 2421 of *LNCS*, pages 499–514. Springer, Aug. 2002.
4. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, June 2001.
5. B. Blanchet. From Secrecy to Authenticity in Security Protocols. In *9th Internat. Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer, Sept. 2002.
6. P. Broadfoot, G. Lowe, and B. Roscoe. Automating Data Independence. In *6th European Symp. on Research in Computer Security (ESORICS'00)*, volume 1895 of *LNCS*, pages 175–190. Springer, Oct. 2000.
7. P. J. Broadfoot and A. W. Roscoe. Capturing Parallel Attacks within the Data Independence Framework. In *15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 147–159, June 2002.
8. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.
9. E. Cohen. TAPS: A First-Order Verifier for Cryptographic Protocols. In *13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 144–158, 2000.
10. H. Comon, V. Cortier, and J. Mitchell. Tree Automata with One Memory, Set Constraints, and Ping-Pong Protocols. In *Automata, Languages and Programming, 28th Internat. Colloq., ICALP'01*, volume 2076 of *LNCS*, pages 682–693. Springer, July 2001.
11. H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *In Proc. 14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *LNCS*, pages 148–164. Springer, June 2003.
12. R. Corin and S. Etalle. An Improved Constraint-Based System for the Verification of Security Protocols. In *9th Internat. Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 326–341. Springer, Sept. 2002.
13. V. Cortier, J. Millen, and H. Rueß. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 97–108, June 2001.
14. D. E. Denning and G. M. Sacco. Timestamps in Key Distribution Protocols. *Commun. ACM*, 24(8):533–536, Aug. 1981.
15. N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols (FMSP'99)*, July 1999.
16. A. Gordon and A. Jeffrey. Authenticity by Typing for Security Protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, June 2001.

17. J. Goubault-Larrecq. A Method for Automatic Cryptographic Protocol Verification (Extended Abstract), invited paper. In *5th Internat. Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'00)*, volume 1800 of *LNCS*, pages 977–984. Springer, May 2000.
18. J. D. Guttman and F. J. T. Fábrega. Protocol Independence through Disjoint Encryption. In *Proceedings, 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 24–34, July 2000.
19. 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, July 2000.
20. 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, July 2000.
21. 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, 1996.
22. G. Lowe. A Hierarchy of Authentication Specifications. In *10th IEEE Computer Security Foundations Workshop (CSFW '97)*, June 1997.
23. J. Millen and V. Shmatikov. Constraint Solving for Bounded-Process Cryptographic Protocol Analysis. In *8th ACM Conf. on Computer and Communications Security (CCS'01)*, pages 166–175, 2001.
24. R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.
25. F. Nielson, H. R. Nielson, and H. Seidl. Cryptographic Analysis in Cubic Time. In *TOSCA 2001 - Theory of Concurrency, Higher Order Languages and Types*, volume 62 of *ENTCS*, Nov. 2001.
26. D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *Operating Systems Review*, 21(1):8–10, 1987.
27. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6(1–2):85–128, 1998.
28. A. W. Roscoe and P. J. Broadfoot. Proving Security Protocols with Model Checkers by Data Independence Techniques. *J. Computer Security*, 7(2, 3):147–190, 1999.
29. M. Rusinovitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 174–187, June 2001.
30. C. Weidenbach. Towards an Automatic Analysis of Security Protocols in First-Order Logic. In *16th Internat. Conf. on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 314–328. Springer, July 1999.
31. T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, Jan. 1992.
32. T. Y. C. Woo and S. S. Lam. A Semantic Model for Authentication Protocols. In *IEEE Symposium on Research in Security and Privacy*, pages 178–194, May 1993.
33. 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.