An Efficient Cryptographic Protocol Verifier Based on Prolog Rules

Bruno Blanchet
INRIA Rocquencourt*
Domaine de Voluceau B.P. 105
78153 Le Chesnay Cedex, France
Bruno.Blanchet@inria.fr

Abstract

We present a new automatic cryptographic protocol verifier based on a simple representation of the protocol by Prolog rules, and on a new efficient algorithm that determines whether a fact can be proved from these rules or not. This verifier proves secrecy properties of the protocols. Thanks to its use of unification, it avoids the problem of the state space explosion. Another advantage is that we do not need to limit the number of runs of the protocol to analyze it. We have proved the correctness of our algorithm, and have implemented it. The experimental results show that many examples of protocols of the literature, including Skeme [24], can be analyzed by our tool with very small resources: the analysis takes from less than 0.1 s for simple protocols to 23 s for the main mode of Skeme. It uses less than 2 Mb of memory in our tests.

1. Introduction

The design of cryptographic protocols is difficult and error-prone. This can be illustrated by flaws found in existing protocols [1, 6, 11, 25]. It is therefore important to have tools to verify the properties of cryptographic protocols. Several techniques can be used to build such tools: logics, such as the BAN logic [11] used in [23], theorem proving, used in Isabelle [33], rank functions [21], typing [2, 12, 22], abstract interpretation [7, 8, 20, 29], model checking, rewriting, and related techniques, used in Elan [13], Brutus [14], Maude [16], FDR [25], NRL [26], the Interrogator [27], Murφ [28], Athena [35]. Most existing protocol verifiers based on model checking suffer from the problem of the state space explosion, and they need very large resources to verify even relatively simple protocols. Moreover, in general, they limit the number of runs of the protocol to guarantee the termination of the verification process. If there exists an attack that only appears with more runs of the protocol, it will not be discovered. Our solution to these problems relies on two ideas:

- a simple intermediate representation of the protocols;
- a new efficient solving algorithm.

We use Prolog rules to represent the protocol and the attacker. Messages and channels are represented by terms; the fact attacker(M) means that the attacker has the message M; rules give implications between such facts. This can be considered as an abstraction of the multiset rewriting [16] or of the linear logic representation [19]. We perform two interesting abstractions:

- Fresh values considered as functions other messages in the protocol. To give an intuition, when the attacker does not modify messages, different values are used for each pair of participants of the protocol, instead of per session.
- We forget the number of times a message appears to remember only that it has appeared. A step of the protocol can be executed several times instead of only once in each session.

These are keys to avoid limiting the number of runs of protocols: the number of repetitions is simply forgotten. Our approximations are safe, in the sense that if the verifier does not find a flaw in the protocol, then there is no flaw. The verifier therefore provides real security guarantees. In contrast, it may give a false attack against the protocol. However, false attacks are rare, and we have been able to prove the secrecy properties of all the protocols that we have considered. Therefore, we believe that our abstractions could also be useful in other protocol tools. Various protocol representations can be translated into our simple representation. We have built an automatic translator from a restricted version of the applied pi calculus [5], that only handles certain

*This work was partly done while the author was at Bell Labs Research, Lucent Technologies.

equational theories, including the theories used to represent shared- and public-key cryptography (encryption and signatures), hash functions, the Diffie-Hellman key agreement. It is also possible for the user to enter directly the rules representing the protocol, since the representation is simple enough.

Using this representation, we have built a tool to prove secrecy properties of protocols. Indeed, the attacker may have a given message m only if the fact $\mathsf{attacker}(m)$ can be proved from the rules representing the protocol and the abilities of the attacker. However, the usual Prolog solving algorithm loops, due to rules that appear in the description of the attacker. Therefore, we have designed a solving algorithm. This algorithm is novel as far as we know, and it appears to be very efficient in practice. We have applied it to prove secrecy properties of several protocols of the literature, including Skeme [24], or to find attacks against them.

Related work Prolog rules and similar formalisms have already been used in a number of works on cryptographic protocols, for example [16, 26, 27]. We propose a more abstract representation of the protocols, that enables us to design a simpler and faster analysis, and to avoid limiting the number of runs of the protocol, thus improving over most model checkers. Of course, the additional approximations imply that our analysis may not be able to prove that certain protocols are secure (it may give false attacks), but in our experiments our analysis was precise enough to prove secrecy properties of the protocols we have considered. A key problem of previous tools using Prolog rules is termination. Our solving algorithm is a big step towards a solution of this problem.

Two works have already tackled the problems met by classical model checkers. Broadfoot, Lowe and Roscoe [10, 34] do not limit the number of runs of protocols. They recycle nonces, to use only a finite number of them in an infinite number of runs. We achieve the same result by directly reusing the same values for nonces. However, they limit the number of *parallel* runs of protocols. We avoid this limitation. They also allow the attacker to simulate honest agents, but use this technique only for servers. We generalize it to all agents involved in the protocols. In their work as well as in ours, the deduction rules for the attacker must have only equalities in their hypotheses, no inequalities (that is, they are positive deduction systems). Song [35] avoids the state space explosion problem by using the strand space model to verify protocols. This model captures the causal information of messages, in a way similar to our deduction rules. However, our model is more abstract than Song's. She sometimes limits the number of runs of the protocol to guarantee termination, whereas we avoid this limitation.

Automatic protocol verifiers have already been built by using abstract interpretation [7, 8, 20, 29]. The analysis of Bodei et al. [7, 8] is not relational: when a variable appears several times in a message, each occurrence can take different values (in the set of values of the variable). All nonces generated by the same restriction are also considered as equal. These are causes of imprecision that our analysis solves. Moreover, Bodei's analysis only handles shared-key cryptography. The main difference between Monniaux' analysis [29] and ours is that Monniaux represents sets of messages by tree automata, whereas we represent rules that generate these sets. This enables us to gain in efficiency. Goubault [20] extends and improves the efficiency of [29], also using ideas of Bolignano [9]. However, [20] does not allow any term to be used in place of a key. We do not have this limitation. In our experimental results, we obtain an even faster analysis with a simpler framework.

Theorem provers are not fully automatic tools: the user has to intervene to provide information on the proof. Previous works using typing are also better suited for human use than for automatic verifiers: type inference is sometimes difficult [4] and types are in general human-readable. Types provide constraints that can help the designers of new protocols ensure the desired security properties, but existing protocols may not satisfy these constraints even if they are correct. In contrast, our analysis yields a fully automatic protocol verifier.

$M, N ::=$	terms
$\quad x$	variable
$\quad a[M_1, \ldots, M_n]$	name
$\quad f(M_1, \ldots, M_n)$	function application
$F ::=$	fact
$\quad p(M_1, \ldots, M_n)$	predicate application
$R ::=$	rule
$\quad F_1 \wedge \ldots \wedge F_n \rightarrow F$	implication

Figure 1. Syntax of our protocol representation

Overview Section 2 details our protocol representation. Section 3 describes our solving algorithm, and sketches its proof of correctness. Several extensions and optimizations to this algorithm are detailed in Section 5. Section 6 gives experimental results and Section 7 concludes.

2. Protocol representation

A protocol is represented by a set of Prolog rules (clauses), whose syntax is given in Figure 1. The terms represent messages that are exchanged between participants of the protocol. A variable can represent any term. Names are used to represent atomic values, such as keys and nonces. Each principal has the ability of creating new names. Here, the created names are considered as functions of the messages previously received by the principal that creates the name. Thus, a different name is created when the preceding messages are different. This is slightly weaker than the fact that a new name is created at each run of the protocol. As noticed by M. Abadi (personal communication), this approximation is in fact similar to the approximation done in some type systems (such as [4]): the type of the new name depends on the types in the environment. It is enough to handle many protocols, and can be enriched by adding other parameters to the name. The function application is used to build terms: examples of functions are the encryption, or hash functions. Predicates are used to represent facts about these messages. Several predicates can be used, but for a first example, we are going to use only one predicate $\mathsf{attacker}(M)$, meaning "the attacker may have the message M". A rule $F_1 \wedge \ldots \wedge F_n \rightarrow F$ means that if all facts $F_1, \ldots, F_n$ are true, then F is also true. A rule with no hypothesis $\rightarrow F$ is written simply F.

We can illustrate the coding of a protocol on the following simple example (this is a simplification of the Denning-Sacco key distribution protocol [17], omitting certificates and timestamps):

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

There are two principals A and B. sk_A is the secret key of A, pk_A its public key. Similarly, sk_B and pk_B for B. The key k is a new key created by A. A sends this key signed with its private key sk_A and encrypted under its public key pk_B. When B receives this messages, it decrypts the message, and assumes, seeing the signature, that the key k has been generated by A. Then it sends a secret s encrypted under k (this is a shared-key encryption). Only A should be able to decrypt the message and get the secret s. (The second message is not really part of the protocol, we use it to check if the key k can really be used to exchange secrets between A and B. In fact, there is an attack against this protocol [11], so s will not remain secret.)

2.1. Representation of primitives

Cryptographic primitives are represented by functions. For instance, we represent the public-key encryption by a function $\mathsf{pencrypt}(m, pk)$ which takes two arguments: the message m to encrypt and the public key pk. There is a function pk that builds the public key from the secret key. (We could also have two functions pk and sk to build respectively the public and secret keys from a secret.) The secret key is represented by a name which has no arguments (that is, there exists only one copy of this name) $sk_A[]$ for A and $sk_B[]$ for B. Then $pk_A = \mathsf{pk}(sk_A[])$ and $pk_B = \mathsf{pk}(sk_B[])$.

More generally, we consider two kinds of functions: constructors and destructors. The constructors are the functions that explicitly appear in the terms that represent messages. For instance, $\mathsf{pencrypt}$ and pk are constructors. Destructors manipulate terms. A destructor g can be defined by one or several equations of the form $g(M_1, \ldots, M_n) = M$ where $M_1, \ldots, M_n, M$ are terms that contain only variables and constructors. For instance, the decryption $\mathsf{pdecrypt}$ is a destructor, defined by $\mathsf{pdecrypt}(\mathsf{pencrypt}(m, \mathsf{pk}(sk)), sk) = m$. Other functions are defined similarly:

- For signatures, there is a constructor $\mathsf{sign}(m, sk)$ that is used to represent the message m signed under the secret key sk. A destructor $\mathsf{getmess}$ defined by $\mathsf{getmess}(\mathsf{sign}(m, sk)) = m$ returns the message without its signature, and $\mathsf{checksign}(\mathsf{sign}(m, sk), \mathsf{pk}(sk)) = m$ only returns the message if the signature is valid.

- For shared-key encryption, we have a constructor $\mathsf{sencrypt}$ and a destructor $\mathsf{sdecrypt}$, defined by $\mathsf{sdecrypt}(\mathsf{sencrypt}(m, k), k) = m$.

- A hash function is represented by a constructor h (and no destructor).

- Tuples of arity n are represented by a constructor $(\_, \ldots, \_)$ and n destructors $i\mathsf{th}$ defined by $i\mathsf{th}((x_1, \ldots, x_n)) = x_i$, $i \in \{1, \ldots, n\}$.

2.2. Representation of the abilities of the attacker

We assume that the protocol is executed in the presence of an attacker that can intercept all messages, compute new messages from the messages it has received, and send any message it can build. We first present the encoding of the computation abilities of the attacker. The encoding of the protocol will be detailed below.

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

$$\begin{array}{c} \mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \\ \rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n)). \end{array}$$

If g is a destructor defined by $g(M_1, \ldots, M_n) = M$, this leads to the rule:

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

If g is defined by several equations, there are several rules, one for each equation. The destructors never appear in the rules, they are coded by pattern-matching on their parameters (here $M_1, \ldots, M_n$) in the hypothesis of the rule and generating their result in the conclusion. In the particular case of the public-key encryption, this yields:

$$\begin{aligned}
&\mathsf{attacker}(m) \wedge \mathsf{attacker}(pk) \\
&\quad \rightarrow \mathsf{attacker}(\mathsf{pencrypt}(m, pk)), \\
&\mathsf{attacker}(sk) \rightarrow \mathsf{attacker}(\mathsf{pk}(sk)), \\
&\mathsf{attacker}(\mathsf{pencrypt}(m, \mathsf{pk}(sk)) \wedge \mathsf{attacker}(sk) \\
&\quad \rightarrow \mathsf{attacker}(m),
\end{aligned} \tag{1}$$

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

For signatures, we obtain the rules:

$$\begin{aligned}
&\mathsf{attacker}(m) \wedge \mathsf{attacker}(sk) \rightarrow \mathsf{attacker}(\mathsf{sign}(m, sk)), \\
&\mathsf{attacker}(\mathsf{sign}(m, sk)) \rightarrow \mathsf{attacker}(m),
\end{aligned}$$

where the first rule corresponds to the constructor sign and the second one to the destructor getmess. (The rule for checksign is removed, since it is implied by the rule for getmess.)

The rules above describe the abilities of the attacker. Moreover, the attacker has the public keys. Therefore, we add the rules $\mathsf{attacker}(\mathsf{pk}(sk_A[]))$ and $\mathsf{attacker}(\mathsf{pk}(sk_B[]))$. We also give a name a to the attacker, that will represent all the names it can generate: $\mathsf{attacker}(a[])$.

2.3. Representation of the protocol itself

Now, we describe how the protocol itself is represented. We consider that A and B are willing to talk to any principal, A, B but also malicious principals that are represented by the attacker. Therefore, the first message sent by A can be $\mathsf{pencrypt}(\mathsf{sign}(k, sk_A[]), \mathsf{pk}(x))$ for any x. We leave to the attacker the task to start the protocol with the principal it wants, that is the attacker will send a first message to A, mentioning the public key of principal with which A should talk. This principal can be B, or another principal represented by the attacker. (The attacker can create public keys, by the rule for constructor pk.) Moreover, the attacker can intercept the message sent by A. This yields a rule of the form

$$\begin{aligned}
&\mathsf{attacker}(\mathsf{pk}(x)) \\
&\quad \rightarrow \mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k, sk_A[]), \mathsf{pk}(x))).
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{attacker}(\mathsf{pk}(x)) \\
&\quad \rightarrow \mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[]), \mathsf{pk}(x))).
\end{aligned} \tag{2}$$

Remark. It would also be possible for A to initiate the protocol itself, by choosing randomly the other principal to which it talks, instead of letting the attacker initiate the protocol. In this case, the rule above would be

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

where x is a variable that represents the secret key of the principal talking with A. However, if we want to represent the protocol by a closed process in the applied pi calculus, the variable x must come from an input. That is, A cannot choose randomly the principal to which it talks. If the process is modeled in the applied pi calculus, the attacker sends a message that indicates with which principal A should talk. This yields the rule (2) given above.

B expects a message of the form $\mathsf{pencrypt}(\mathsf{sign}(k', sk), \mathsf{pk}(sk_B[]))$. When such a message is received, it tests whether A has signed the message (that is, B evaluates $\mathsf{checksign}(\mathsf{sign}(k', sk), pk_A)$, and this only succeeds when $sk = sk_A[]$). If so, it assumes that the key k' is only known by A, and sends a secret s encrypted under k'. We assume that the attacker relays the message coming from A, and intercepts the message sent by B. Hence the rule:

$$\begin{aligned}
&\mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k', sk_A[]), \mathsf{pk}(sk_B[]))) \\
&\quad \rightarrow \mathsf{attacker}(\mathsf{sencrypt}(s[], k')).
\end{aligned}$$

With these rules, A cannot play the role of B and vice-versa. If we want that, we can simply add the corresponding rules, that are obtained by swapping A and B in the above rules:

$$\begin{aligned}
&\mathsf{attacker}(\mathsf{pk}(x)) \\
&\quad \rightarrow \mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k_B[\mathsf{pk}(x)], sk_B[]), \mathsf{pk}(x))), \\
&\mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k', sk_B[]), \mathsf{pk}(sk_A[]))) \\
&\quad \rightarrow \mathsf{attacker}(\mathsf{sencrypt}(s_A[], k')).
\end{aligned}$$

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

Remark. When the protocol makes some communications on private channels, on which the attacker cannot a priori listen or send messages, a second predicate can be used: $\mathsf{mess}(C, M)$ meaning "the message M can appear on channel C". In this case, if the attacker manages to get the name of the channel C, it will be able to listen and send messages on this channel. Thus, two new rules have to be added to describe the behavior of the attacker. The attacker can listen on all the channels it has:

$$\mathsf{mess}(x, y) \wedge \mathsf{attacker}(x) \rightarrow \mathsf{attacker}(y).$$

It can send all the messages it has on all the channels it has:

$$\mathsf{attacker}(x) \wedge \mathsf{attacker}(y) \rightarrow \mathsf{mess}(x, y).$$

2.4. Summary

To sum up, a protocol can be represented by three sets of Prolog rules:

- Rules representing the computation abilities of the attacker. There is one rule $\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n))$ for each constructor f, and one rule $\mathsf{attacker}(M_1) \wedge \ldots \wedge \mathsf{attacker}(M_n) \rightarrow \mathsf{attacker}(M)$ for each equation $g(M_1, \ldots, M_n) = M$ defining a destructor g.
- Facts corresponding to the initial knowledge of the attacker. There is a fact $\mathsf{attacker}(a[])$ giving a name to the attacker. In general, there are also facts giving the public keys of the participants and/or their names to the attacker.
- Rules representing the protocol itself. There is one set of rule for each message in the protocol. In the set corresponding to the ith message, sent by principal X, the rules are of the form $\mathsf{attacker}(M_{j_1}) \wedge \ldots \wedge \mathsf{attacker}(M_{j_n}) \rightarrow \mathsf{attacker}(M_i)$ where $M_{j_1}, \ldots, M_{j_n}$ are the patterns of the messages received by X before sending the ith message, and M_i is the pattern of the ith message.

The rules representing the Denning-Sacco protocol are summarized in Figure 2.

2.5. Approximations

The reader can notice that our representation of protocols is approximate:

- Freshness is modeled by letting new names be functions of messages previously received by the creator of the name in the run of the protocol. When the attacker does not alter messages, this means that different values are used per pair of principals running the protocol instead of per session. When the attacker does alter messages, different values are used when different messages are received.
- A step of the protocol can be completed several times, as long as the previous steps have been completed at least once between the same principals. For instance, in a session between the attacker and a principal A, the attacker sends the first message M_1, A replies with M_2, the attacker can then send two messages in place of the third message, as long as they correspond to the pattern expected by A. That is, the attacker sends M_3 and gets M_4 from A, then the attacker sends M_3' and gets M_4' from A. The attacker can also perform again the ith step, even if further steps have already been performed. Therefore, the actions of the principals are not organized into runs.

But the important point is that the approximations are always performed in the safe direction: if an attack exists in a more precise model, such as multiset rewriting [16], or the applied pi calculus [5], then it also exists in our representation. (We are currently proving that our translation from the applied pi calculus to this representation has this correctness property.) Performing approximations enables us to build a much more efficient verifier, which will be able to handle larger and more complex protocols. Another advantage is that the verifier does not have to limit the number of runs of the protocol. The price to pay for this is that false attacks may be found by the verifier: sequences of rule applications that do not correspond to a protocol run. When a false attack is found, we cannot know whether the protocol is secure or not: a real attack may also exist. A more precise analysis is required in this case. But our representation is precise enough so that false attacks are rare. (This is demonstrated by our experiments, see Section 6.)

2.6. Secrecy

Our goal is to determine secrecy properties: for instance, can the attacker get the secret s ? That is, can the fact $\mathsf{attacker}(s[])$ be inferred from the rules ? If $\mathsf{attacker}(s[])$ can be inferred, the sequence of rules applied to derive $\mathsf{attacker}(s[])$ will lead to the description of an attack.

Our notion of secrecy is similar to that of [4, 7, 12]: a term M is secret if the attacker cannot get it by listening and sending messages, and performing computations. This notion of secrecy is weaker than non-interference, but it is

Computation abilities of the attacker:	
pencrypt	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(pk) \rightarrow \mathsf{attacker}(\mathsf{pencrypt}(m, pk))$
pk	$\mathsf{attacker}(sk) \rightarrow \mathsf{attacker}(\mathsf{pk}(sk))$
pdecrypt	$\mathsf{attacker}(\mathsf{pencrypt}(m, \mathsf{pk}(sk)) \wedge \mathsf{attacker}(sk) \rightarrow \mathsf{attacker}(m)$
sign	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(sk) \rightarrow \mathsf{attacker}(\mathsf{sign}(m, sk))$
getmess	$\mathsf{attacker}(\mathsf{sign}(m, sk)) \rightarrow \mathsf{attacker}(m)$
checksign	removed since implied by getmess
sencrypt	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(k) \rightarrow \mathsf{attacker}(\mathsf{sencrypt}(m, k))$
sdecrypt	$\mathsf{attacker}(\mathsf{sencrypt}(m, k)) \wedge \mathsf{attacker}(k) \rightarrow \mathsf{attacker}(m)$
Initial knowledge of the attacker:	
	$\mathsf{attacker}(\mathsf{pk}(sk_A[])), \quad \mathsf{attacker}(\mathsf{pk}(sk_B[])), \quad \mathsf{attacker}(a[])$
Protocol:	
First message:	$\mathsf{attacker}(\mathsf{pk}(x)) \rightarrow \mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[]), \mathsf{pk}(x)))$
Second message:	$\mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k', sk_A[]), \mathsf{pk}(sk_B[]))) \rightarrow \mathsf{attacker}(\mathsf{sencrypt}(s[], k'))$

Figure 2. Summary of our representation of the Denning-Sacco protocol

adequate to deal with the secrecy of fresh names. Non-interference is better at excluding implicit information flows or flows of parts of compound values. (See [3, Section 6] for further discussion and references.)

Technically, the hypotheses $F_1, \dots, F_n$ of a rule are considered as a multiset. This means that the order of the hypotheses is irrelevant, but the number of times an hypothesis is repeated is important. (This is not related to the ideas of multiset rewriting: the semantics of a rule does not depend on the number of repetitions of its hypotheses, but considering multisets is useful to make explicit the elimination of duplicate hypotheses in our verifier. It will also be useful in the proof of the algorithm.) Formally, a multiset of facts S is a function from facts to integers, such that $S(F)$ is the number of repetitions of F in S. The inclusion on multisets is the point-wise order on functions: $S \subseteq S' \Leftrightarrow \forall F, S(F) \leq S'(F)$. If f is a function from facts to facts, we can extend it to multisets of facts by $(f(S))(F) = \sum_{F' \text{ such that } f(F')=F} S(F')$. This applies in particular when f is a substitution.

We determine whether a given formula can be implied by a given set of rules. This is more precisely defined as follows.

Definition 1 *We define rule implication by:*

$$(H_1 \rightarrow C_1) \Rightarrow (H_2 \rightarrow C_2) \text{ if and only if}$$
$$\exists \sigma, \sigma C_1 = C_2, \sigma H_1 \subseteq H_2$$

where H_1 and H_2 are multisets of hypotheses, σ is a substitution.

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

Definition 2 (Derivability) *Let F be a closed fact, that is, a fact without variable. Let B be a set of rules. F is derivable from B if and only if there exists a finite tree defined as follows:*

1. *Its nodes (except the root) are labelled by rules $R \in B$;*
2. *Its edges are labelled by closed facts;*
3. *If the tree contains a node labelled by R with one incoming edge labelled by F_0 and n outgoing edges labelled by $F_1, \dots, F_n$, then $R \Rightarrow \{F_1, \dots, F_n\} \rightarrow F_0$.*
4. *The root has one outgoing edge, labelled by F.*

Such a tree is a derivation of F from B.

In a derivation, if there is a node labelled by R with one incoming edge labelled by F_0 and n outgoing edges labelled by $F_1, \dots, F_n$, then the rule R can be used to infer F_0 from $F_1, \dots, F_n$. Therefore, there exists a derivation of F from B if and only if F can be inferred from rules in B.

3. Solving algorithm

Our representation is a set of Prolog rules, and our goal is simply to determine whether a given fact can be inferred from these rules or not. This is exactly the problem that is solved by usual Prolog systems. However, we cannot use these systems here, because they would not terminate. For instance, the rule:

$$\mathsf{attacker}(\mathsf{pencrypt}(m, \mathsf{pk}(sk)) \wedge \mathsf{attacker}(sk)$$
$$\rightarrow \mathsf{attacker}(m)$$

leads to considering more and more complex terms, with an unbounded number of encryptions. We could of course limit arbitrarily the depth of terms to solve the problem, but we can do much better than that. Indeed, even when limiting the depth of terms, the complexity of the depth-first search will be very large. There are many rules with conclusion $\mathsf{attacker}(x)$ that can always be applied, when we search a fact of the form $\mathsf{attacker}(M)$. We can get better results with a more guided search.

The main idea to guide the search is to combine pairs of rules by unification, when the unified facts are not of the form $\mathsf{attacker}(x)$. When the consequence of a rule R unifies with one hypothesis of another (or the same) rule R', we can infer a new rule, that corresponds to applying R and R' one after the other. Formally, this is defined as follows:

Definition 3 *Let R and R' be two rules, $R = H \rightarrow C$, $R' = H' \rightarrow C'$. Assume that there exists $F_0 \in H'$ such that: C and F_0 are unifiable, and σ is the most general unifier of C and F_0.*

In this case, we define

$$R \circ_{F_0} R' = \sigma(H \cup (H' - F_0)) \rightarrow \sigma C'.$$

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

$$\begin{aligned}&\mathsf{attacker}(\mathsf{pk}(x)) \wedge \mathsf{attacker}(x)\\&\quad \rightarrow \mathsf{attacker}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[]))\end{aligned}$$

with the substitution $\sigma = \{sk \mapsto x, m \mapsto \mathsf{sign}(k[\mathsf{pk}(x)], sk_A[])\}$. In terms of logic programming, $R \circ_{F_0} R'$ is the result of resolving R' with R upon F_0. Of course, if this operation is applied without limitations, it does not terminate (consider the same rule as above). We specify conditions to limit it. As far as we know, these conditions are new.

Let S be a finite set of facts. We define $F \in_{\mathsf{r}} S$ by: there exists a substitution σ mapping variables to variables such that $\sigma F \in S$. By default, $S = \{\mathsf{attacker}(x)\}$, but the algorithm is also correct with other values of S. The idea is to only unify facts F such that $F \notin_{\mathsf{r}} S$. The precise formal condition is slightly more complex (see below).

The solving algorithm works in two phases, which are described in Figures 3 and 4 respectively. The first phase transforms the rule base into a new one, that implies the same facts. The second one uses a depth-first search to determine whether a fact can be inferred or not from the rules.

The first phase (Figure 3) contains 3 steps. The first step inserts in B the initial rules representing the protocol and the attacker (rules that are in B_0). These rules are simplified by eliminating duplicate hypotheses, and if a rule R implies a rule R', R' is removed (definition of add). The second step is a fixed point iteration, that adds rules created by resolution. The composition of rules R and R' is only added if

- the hypotheses of R contain only facts F which satisfy $F \in_{\mathsf{r}} S$ (i.e. by default they are of the form $\mathsf{attacker}(x)$),
- and the hypothesis F_0 of R' that we unify does not satisfy $F_0 \in_{\mathsf{r}} S$ (i.e. by default F_0 is not of the form $\mathsf{attacker}(x)$).

When a rule is created by this composition, it is added to the rule base B, after simplification (duplicate hypotheses are removed and if a rule R implies a rule R', R' is removed). At last, the third step is to extract from B the new rule base B', by taking only the rules whose all hypotheses F satisfy $F \in_{\mathsf{r}} S$ (by default, this means that F is of the form $\mathsf{attacker}(x)$). The following remarks can help understand the algorithm:

- This algorithm corresponds to a kind of forward search. In a forward search, a fact is unified with an

---

Let B be the rule base, B_0 be the set of rules representing the attacker and the protocol.
We define

$$\mathsf{add}(R, B) = \begin{cases} B & \text{if } \exists R' \in B, R' \Rightarrow R,\\ \{R\} \cup \{R' \in B | R \not\Rightarrow R'\} & \text{otherwise.}\end{cases}$$

We also define $\mathsf{elimdup}(H \rightarrow C) = (H \cap \mathbf{1}) \rightarrow C$, where $\mathbf{1}$ is the multiset which contains one copy of each fact: $\forall F, \mathbf{1}(F) = 1$. The function elimdup eliminates the duplicate hypotheses from a rule.

1. For each $R \in B_0$, $B \leftarrow \mathsf{add}(\mathsf{elimdup}(R), B)$.
2. Let $R \in B$, $R = H \rightarrow C$ and $R' \in B$, $R' = H' \rightarrow C'$. Assume that there exists $F_0 \in H'$ such that:
   (a) $R \circ_{F_0} R'$ is defined;
   (b) $\forall F \in H, F \in_{\mathsf{r}} S$;
   (c) $F_0 \notin_{\mathsf{r}} S$.

   In this case, we execute
   $$B \leftarrow \mathsf{add}(\mathsf{elimdup}(R \circ_{F_0} R'), B).$$
   This procedure is executed until a fixed point is reached.
3. Let $B' = \{(H \rightarrow C) \in B | \forall F \in H, F \in_{\mathsf{r}} S\}$.

Figure 3. First phase: completion of the rule base

hypothesis of a rule, and a new rule is created that contains one hypothesis less. This is performed until no new fact can be inferred.

Here, assume $S = \emptyset$. Hypothesis (b) means that R has no hypothesis, that is, R is a fact. Hypothesis (c) is always true. Then a fact $R = C$ is unified with an hypothesis of the rule R'. In this case, we have exactly a forward search.

When $S \neq \emptyset$, the algorithm resembles a forward search for a modified notion of facts. Let S-facts be the rules $H \to C$, where $\forall F \in H, F \in_{\mathsf{r}} S$. Let S-rules be the rules $H' \to F_S$ where $\forall F \in H', F \notin_{\mathsf{r}} S$ and F_S is an S-fact. Notice that all rules are S-rules, simply by writing first the hypotheses that satisfy $F \notin_{\mathsf{r}} S$. Hypothesis (b) means that R is an S-fact. Hypothesis (c) means that F_0 is an hypothesis of an S-rule R'. The S-fact and S-rule are combined, to give a new S-rule (that can be an S-fact). When all combinations have been performed, only S-facts are kept in B'.

- This algorithm is similar to an unfolding of the logic program [37], but there is one important difference: In the unfolding, a rule R' and an hypothesis F_0 of R' are chosen, and the resolution is performed with *all* rules R whose consequence is unifiable with F_0. Here, the resolution is only applied with rules R whose hypotheses F satisfy $F \in_{\mathsf{r}} S$.

- The speed of the algorithm comes essentially from the fact that there are not many rules $H \to C$ such that $\forall F \in H \cup \{C\}, F \in_{\mathsf{r}} S$. (In our uses of this algorithm, S is always a very small set.) For the other rules, (b) implies $C \notin_{\mathsf{r}} S$, therefore, using the default definition of S, C is not of the form $\mathsf{attacker}(x)$: C cannot be unified with any hypothesis of any rule, only few hypotheses of rules will correspond. Similarly, F_0 is not of the form $\mathsf{attacker}(x)$, so only few conclusions of rules can be unified with F_0. Therefore, in general, few unifications are performed, and the algorithm is very fast.

We prove that the rules in B' imply exactly the same facts as the rules in B_0.

Lemma 1 (Correctness of phase 1) *Let F be a closed fact. F is derivable from the rules in B_0 if and only if F is derivable from the rules in B'.*

Proof sketch We only give a proof sketch here, a detailed proof can be found in Appendix A.

Assume that F is derivable from B_0 and consider a derivation of F from B_0. The key idea of the proof is the following. Assume that the rules R and R' are applied one after the other in the derivation of F. Also assume that these rules have been combined by $R \circ_{F_0} R'$, yielding rule R''. In this case, we replace R and R' by R'' in the derivation of F. When no more replacement can be done, we show that all the hypotheses F_0 of the remaining rules satisfy $F_0 \in_{\mathsf{r}} S$. Then all these rules are in B', and we have built a derivation of F from B'. The converse is easy to prove. $\square$

We define $\mathsf{derivablerec}(R, B'')$ by

1. $\mathsf{derivablerec}(R, B'') = \emptyset$ if $\exists R' \in B'', R' \Rightarrow R$;
2. $\mathsf{derivablerec}(\emptyset \to C, B'') = \{C\}$ otherwise;
3. $\mathsf{derivablerec}(R, B'') =$ $\cup\{\mathsf{derivablerec}(\mathsf{elimdup}(R' \circ_{F_0} R), \{R\} \cup B'') | R' \in B', F_0$ such that $R' \circ_{F_0} R$ is defined $\}$ otherwise.

$\mathsf{derivable}(F) = \mathsf{derivablerec}(\{F\} \to F, \emptyset)$.

Figure 4. Second phase: backward depth-first search

The second phase (Figure 4) searches the facts that can be inferred from B'. This is simply a backward depth-first search. The search is performed by calling $\mathsf{derivablerec}(R, B'')$ with two parameters: a rule R and a set of rules B''. The hypotheses of R are the facts that we currently want to prove. Its conclusion is an instance of the fact F that we initially wanted to prove. Moreover, the rule R is always a consequence of the rules in B': the conclusion of R can be proved by rules of B' from the hypotheses of R. The set B'' is the set of rules that we have already seen during the search. $\mathsf{derivablerec}(\{F\} \to F, B'')$ returns the set of instances of F that can be proved.

If R is implied by a rule in B'', the current branch of the search fails: this is a cycle, we are looking for instances of facts that we have already looked for (first point in the definition of $\mathsf{derivablerec}$). We backtrack to try finding another derivation of the goal. If R has no hypothesis, the search succeeds: the conclusion of R is proved (second point of the definition of $\mathsf{derivablerec}$). Otherwise, we have to go on searching. We try to use rule $R' \in B'$ to prove one of the hypotheses of R, F_0. That is, we call $\mathsf{derivablerec}$ with the rule $R' \circ_{F_0} R$ (in which F_0 has been replaced by the hypotheses of R', R' and R being instantiated so that the conclusion of R' and F_0 are unified).

The following theorem gives the correctness of the whole algorithm. It shows that we can use our algorithm to determine whether a fact is derivable or not from the initial rules. The first part of the theorem shows that when calling $\mathsf{derivable}$ with a not necessarily closed fact F', the instances of F' that can be derived from the rules are the instances of the facts returned by $\mathsf{derivable}(F')$. The second part deals with the particular case of closed facts.

Theorem 2 (Correctness) *Let F be a closed fact. Let F' such that there exists a substitution σ such that $\sigma F' = F$. F is derivable from the rules in B_0 if and only if $\exists F'' \in \mathsf{derivable}(F'), \exists \sigma, F = \sigma F''$.*

In particular, F is derivable from B_0 if and only if $F \in \mathsf{derivable}(F)$.

Proof sketch Using Lemma 1, we only have to prove that F is derivable from B' if and only if $\exists F'' \in \mathsf{derivable}(F'), \exists \sigma, F = \sigma F''$.

Essentially, derivablerec performs a classical depth-first search of the rule base B' to find the desired fact. This search is stopped in case of cycle. F is derivable if and only if it is found by the depth-first search. The detailed proof can be found in Appendix A. □

4. Termination

4.1. Termination of the basic algorithm

The fixed point iteration of the first phase does not always terminate, even if it terminates in most examples of protocols. We will see below several ways to force its termination.

The following proposition shows that the depth-first search of the second phase terminates on the rule base B' built by the first phase.

Proposition 3 *If F is closed and $S = \{\mathsf{attacker}(x)\}$, then $\mathsf{derivable}(F)$ terminates. (Otherwise, the termination of $\mathsf{derivable}(F)$ is not guaranteed.)*

Proof sketch The hypotheses of the rules in B' are smaller than the conclusion. Hence the depth-first search considers smaller and smaller terms, and thus terminates. The proof can be found in Appendix B. □

4.2. Detecting (and solving) some non-termination cases

Assume that a rule $R = \{F_0\} \to C$ is inferred, with $C = \sigma F_0$, where σ is such that $\exists x, (x \in \mathsf{fv}(\sigma x) \wedge x \neq \sigma x)$, where $\mathsf{fv}(M)$ is the set of variables in the term M. Assume that $F_0 \notin_{\mathsf{r}} S$, and there exists a derivation of an instance of F_0 (more precisely, the verifier generates a rule $H' \to \sigma' F_0$, whose hypotheses $F \in H'$ satisfy $F \in_{\mathsf{r}} S$).

Then, in general, the completion process (phase 1) does not terminate. Indeed, the rule R can be combined with the rule $H' \to \sigma' F_0$, yielding $H' \to \sigma' C = H' \to \sigma' \sigma F_0$, then this rule can combined again with R, yielding $H' \to \sigma' \sigma C = H' \to \sigma' \sigma^2 F_0$. We can go on this way, and obtain for all integers n: $H' \to \sigma' \sigma^n F_0$, and in general none of these rules implies another, therefore all these rules will be generated by the solving algorithm.

An example of such an exploding rule is:

$$\mathsf{attacker}(f(x)) \to \mathsf{attacker}(f(g(x))).$$

A solution to this non-termination case is of course to add F_0 to the set S, thus forbidding the above combinations of the rule R. Another solution is to add a new rule, that implies all the previous rules, for n large enough. For example, let α be a renaming of variables that has an image disjoint from the variables appearing in H'. Then the rule $H' \to \alpha \sigma^{n_0} C$ implies all the previous rules for $n \geq n_0$. When such a rule is present, the previous rules are automatically removed, and the non-termination is avoided. From the point of view of abstract interpretation, adding a new rule in this way can be considered as a widening [15], that we use to force the convergence of the fixed point iteration.

This non-termination case can be detected automatically by the verifier.

4.3. Enforcing termination

Termination can be enforced by limiting the depth of terms. Each term that starts at a depth greater than a limit fixed by the user is replaced by a new variable.

This way, if a fact can be generated by the system without depth limitation, it can also be generated by the system with depth limitation. The converse is of course wrong. The system remains correct (if it says that a protocol does not leak certain secrets, then the protocol definitely does not leak these secrets), but some precision is lost.

In practice, the algorithm terminates for many protocols without limiting the depth of terms. That is why, by default, the depth of terms is not limited in our tool. Moreover, limiting the depth of terms that appear in the rules does not limit the depth of terms that can be built by the attacker, since even with rules of bounded depth, the attacker can create terms of unbounded depth. Therefore, even if limiting the depth of terms in the rules leads to approximations, it is more precise than limiting the depth of terms in the usual depth-first search algorithm of Prolog.

5. Optimizations and extensions

5.1. Tuples

The tuples are denoted by $(M_1, \ldots, M_n)$. Tuples of different arity are considered as different functions. But the user does not have to define each of these functions: they are all built-in.

The attacker rules:

$$\mathsf{attacker}(M_1) \wedge \ldots \wedge \mathsf{attacker}(M_n) \to \mathsf{attacker}((M_1, \ldots, M_n))$$
$$\mathsf{attacker}((M_1, \ldots, M_n)) \to \mathsf{attacker}(M_i)$$

are also built-in, and treated in a specially optimized way. Indeed, these rules mean that $\mathsf{attacker}((M_1,\dots,M_n))$ is derivable if and only if $\forall i \in \{1,\dots,n\}$, $\mathsf{attacker}(M_i)$ is derivable. When a fact of the form $\mathsf{attacker}((M_1,\dots,M_n))$ is met, it is replaced by $\mathsf{attacker}(M_1) \wedge \dots \wedge \mathsf{attacker}(M_n)$. If this replacement is done in the conclusion of a rule $H \rightarrow \mathsf{attacker}((M_1,\dots,M_n))$, n rules are created: $H \rightarrow \mathsf{attacker}(M_i)$ for each $i \in \{1,\dots,n\}$. This replacement is of course done recursively: if M_i itself is a tuple, it is replaced again.

Notice that (x,y,z), $(x,(y,z))$ and $((x,y),z)$ are different terms. Tuples are different from the concatenation. This can have important consequences. For instance, the Otway-Rees protocol [32] is flawed when using concatenation, not when using tuples. Similarly, the simplified version of the Yahalom protocol of [11] is correct from the point of view of secrecy when using tuples, whereas it is flawed when using concatenation [36, Attack 1]. (The second attack of [36] exposes an authentication flaw, not a secrecy problem.) Of course, the implementation of the protocol must correspond to the model of the verifier.

5.2. Removing useless rules and useless hypotheses

If a rule has a conclusion which is already in the hypotheses, this rule does not generate new facts. Such rules are therefore removed as soon as they are encountered by our verifier.

If a rule $H \rightarrow C$ contains in its hypotheses $\mathsf{attacker}(x)$, where x is a variable that does not appear elsewhere in the rule, the hypothesis $\mathsf{attacker}(x)$ can be removed. Indeed, the attacker always has at least a message. Therefore $\mathsf{attacker}(x)$ is always satisfied.

5.3. Secrecy assumptions

When the user knows that a fact will not be derivable, he can tell it to the verifier. (When this fact is of the form $\mathsf{attacker}(M)$, the user tells that M remains secret.) The tool then removes all rules which have this fact in their hypotheses. At the end of the computation, the program checks that the fact is indeed underivable from the obtained rules. If the user has given erroneous information, an error message is displayed. Even in this case, the verifier never wrongly claims that a protocol is secure.

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

5.4. Diffie-Hellman key agreement

The Diffie-Hellman key agreement [18] enables two principals to build a shared secret. It is used as an elementary step in more complex protocols, such as Skeme [24].

Formally, the Diffie-Hellman key agreement can be modeled by using two functions f and g that satisfy the equation

$$f(y, g(x)) = f(x, g(y)). \tag{3}$$

In practice, the functions are $f(x,y) = y^x \mod p$ and $g(x) = \alpha^x$ where p is prime and α is a generator of $\mathbb{Z}_p^*$. The equation $f(y,g(x)) = (\alpha^x)^y \mod p = (\alpha^y)^x \mod p = f(x,g(y))$ is satisfied. In our verifier, following the ideas used in the applied pi calculus [5], we do not consider the underlying number theory; we work abstractly with the equation (3). The Diffie-Hellman key agreement involves two principals A and B. A chooses a random name x_0, and sends $g(x_0)$ to B. Similarly, B chooses a random name x_1, and sends $g(x_1)$ to A. Then A computes $f(x_0, g(x_1))$ and B computes $f(x_1, g(x_0))$. Both values are equal by (3), and they are secret: assuming that the attacker cannot have x_0 or x_1, it can compute neither $f(x_0, g(x_1))$ nor $f(x_1, g(x_0))$.

The equation (3) cannot be written directly in our framework that uses only constructors and destructors. Nevertheless, it can be encoded as follows: the constructors are g, h_0, and h_1, and f is a destructor defined by

$$\begin{aligned} f(y, g(x)) &= h_1(x,y), \\ f(x, g(y)) &= h_1(x,y), \\ f(x,y) &= h_0(x,y). \end{aligned}$$

Notice that this definition of f is non-deterministic: a term such as $f(a, g(b))$ can be reduced to $h_1(a,b)$, $h_1(b,a)$, and $h_0(a, g(b))$. When two terms M_1 and M_2 are equal according to the equation (3) then there exists a common term M such that both M_1 and M_2 reduce to M. (Both sides reduce to $h_1(x,y)$ when there is at least one g in the second parameter of f, and to $h_0(x,y)$ otherwise.) The equation is then modeled correctly.

5.5. Key compromise

The weakness of some protocols is that when an attacker manages to get some session keys, then it can also get the secrets of other sessions. Such a problem appears for example in the Needham-Schroeder shared-key protocol. It can be detected by our protocol verifier.

The strategy to model the compromise of some session keys is as follows. We say that a name is a session name if it is created at each session of the protocol. (In general, all names except long-term secret keys are session names.) We add a parameter (session identifier) to each session name a.

The session identifier of a is a given constant s_0 when a has been created during a session compromised by the attacker. The session identifier is s_1 when a has been created in a session that has not been compromised. We define a predicate comp such that $\mathsf{comp}(M)$ is true when all session names in M have session identifier s_0. This can be defined by the following rules:

For each constructor f,
$\mathsf{comp}(x_1) \wedge \ldots \wedge \mathsf{comp}(x_k) \rightarrow \mathsf{comp}(f(x_1, \ldots, x_k))$
For each session name a,
$\mathsf{comp}(x_1) \wedge \ldots \wedge \mathsf{comp}(x_k) \rightarrow \mathsf{comp}(a[s_0, x_1, \ldots, x_k])$
For each non-session name a,
$\mathsf{comp}(x_1) \wedge \ldots \wedge \mathsf{comp}(x_k) \rightarrow \mathsf{comp}(a[x_1, \ldots, x_k])$

We define a predicate $\mathsf{attacker}_0$ by the rules normally used to encode the protocol, with session identifier s_0, and a predicate $\mathsf{attacker}_1$ by the same rules with session identifier s_1. Then we add rules

$$\mathsf{comp}(x_1) \wedge \ldots \wedge \mathsf{comp}(x_k) \\ \rightarrow \mathsf{attacker}_0(a[s_0, x_1, \ldots, x_k])$$

for each session name a. These rules express that the attacker $\mathsf{attacker}_0$ has the names of session identifier s_0. Moreover, we add the rule

$$\mathsf{attacker}_0(x) \rightarrow \mathsf{attacker}_1(x).$$

The intuitive meaning of the predicates is the following: $\mathsf{attacker}_0(M)$ is true if and only if M can be obtained by the attacker by compromising the sessions of identifier s_0; $\mathsf{attacker}_1(M)$ is true if M can be obtained by the attacker in a non-compromised session, using the knowledge obtained in the compromised sessions. We can then use our tool to query the fact $\mathsf{attacker}_1(s[s_1])$, where s is a session secret. If this fact is underivable, then the protocol does not have the weakness mentioned above: the attacker cannot have the secret s of a session that it has not compromised. In contrast, it is normal that $\mathsf{attacker}_1(s[s_0])$, since the attacker has compromised the sessions of identifier s_0.

Our translation from the applied pi calculus can automatically add the rules described above to model the compromise of session keys. Also notice that our solving algorithm uses $S = \{\mathsf{comp}(x), \mathsf{attacker}_0(x), \mathsf{attacker}_1(x)\}$ in this case.

Remark. We could also use a single predicate $\mathsf{attacker}$ instead of $\mathsf{attacker}_0$ and $\mathsf{attacker}_1$. However, this would yield a less precise model, leading to more false attacks. For example, we could not prove that the corrected version of the Needham-Schroeder shared key protocol [31] is secure with this model.

6. Experimental results

We have implemented our verifier in Ocaml, and have performed tests on a Pentium MMX 233 MHz, under Linux 2.0.32 (RedHat 5.0). The results are summarized in Figure 5, with references to the papers that describe the protocols and the attacks. In these tests, the protocols are fully modeled, including interaction with the server for all versions of the Needham-Schroeder, Denning-Sacco, Otway-Rees, and Yahalom protocols. We use secrecy assumptions to speed up the search. These assumptions say that the secret keys of the principals, and the random values of the Diffie-Hellman key agreement and the session keys in the Skeme protocol, remain secret. Thanks to these secrecy assumptions, the analysis time of Skeme is 23 s instead of 70 s. The column "#Rules" indicates the number of Prolog rules in our representation of each protocol. The large number of rules for the Needham-Schroeder shared-key protocol comes from the encoding of the compromise of session keys. In the Needham-Schroeder shared key protocol, the last messages are

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

where N_B is a nonce. Representing this with a function $\mathsf{minusone}(x) = x - 1$, this yields a loop in our verifier, since it believes that 1 can be subtracted any number of times from N_B. Moreover, the techniques mentioned previously to force termination lead to a false attack. The purpose of the subtraction is to distinguish the reply of A from B's message. As mentioned in [6], it would be clearer to have:

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

We use this encoding. Our tool then terminates, and the analysis is precise. There was no other termination problem in the tests of Figure 5.

These results show that our analysis can be used to verify secrecy properties of standard cryptographic protocols, in a small amount of time. It takes only a very small amount of memory (less than 2 Mb in all these tests).

7. Conclusion

We believe that our protocol verifier can provide new possibilities to verify cryptographic protocols: it is very efficient, and thus can handle complex protocols; it also avoids limiting the number of runs of the protocol. This is achieved by using a simple representation of the protocol, and a new solving algorithm.

Directions for further work include generalizing the tool to be able to handle general equational theories. A more

Protocol	Result	#Rules	Time (ms)
Needham-Schroeder public key [30]	Attack [25]	14	70
Needham-Schroeder public key corrected [25]	Secure	14	60
Needham-Schroeder shared key [30, 11]	Attack [17]	47	760
Needham-Schroeder shared key corrected [31]	Secure	51	1190
Denning-Sacco [17]	Attack [11]	15	40
Denning-Sacco corrected [11]	Secure	15	40
Otway-Rees [32]	Secure	9	270
Otway-Rees, variant of [33]	Attack [33]	9	260
Yahalom [11]	Secure	10	110
Simpler Yahalom [11]	Secure	10	310
Main mode of Skeme [24]	Secure	23	23070

Figure 5. Experimental results

general study of the termination of the algorithm would also be interesting, to find conditions that guarantee the termination of the basic algorithm, and new ways of forcing termination when these conditions are not satisfied.

Acknowledgments

This work owes much to discussions with Martín Abadi. I am very grateful to him for what he taught me. I would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] M. Abadi. Explicit Communication Revisited: Two New Attacks on Authentication Protocols. *IEEE Transactions on Software Engineering*, 23(3):185–186, Mar. 1997.

[2] M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, Sept. 1999.

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

[4] M. Abadi and B. Blanchet. Secrecy Types for Asymmetric Communication. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, volume 2030 of *Lecture Notes on Computer Science*, pages 25–41, Genova, Italy, Apr. 2001. Springer Verlag.

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

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

[7] C. Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, Jan. 2000.

[8] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Control Flow Analysis for the π-calculus. In *International Conference on Concurrency Theory (Concur'98)*, volume 1466 of *Lecture Notes on Computer Science*, pages 84–98. Springer Verlag, Sept. 1998.

[9] D. Bolignano. Towards a Mechanization of Cryptographic Protocol Verification. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes on Computer Science*, pages 131–142. Springer Verlag, 1997.

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

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

[12] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and Group Creation. In C. Palamidessi, editor, *CONCUR 2000: Concurrency Theory*, volume 1877 of *Lecture Notes on Computer Science*, pages 365–379. Springer Verlag, Aug. 2000.

[13] H. Cirstea. Specifying Authentication Protocols Using Rewriting and Strategies. In I. Ramakrishnan, editor, *Practical Aspects of Declarative Languages (PADL'01)*, volume 1990 of *Lecture Notes on Computer Science*, pages 138–152, Las Vegas, Nevada, Mar. 2001. Springer Verlag.

[14] E. M. Clarke, S. Jha, and W. Marrero. Using State Space Exploration and a Natural Deduction Style Message Derivation Engine to Verify Security Protocols. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, June 1998.

[15] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the fourth international symposium PLILP'92 (Programming Language Implementation and Logic Program-*

ming), Lecture Notes on Computer Science, pages 269–295. Springer Verlag, Aug. 1992.

[16] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols*, Indianapolis, Indiana, 25 June 1998.

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

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

[19] 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)*, Trento, Italy, 5 July 1999.

[20] J. Goubault-Larrecq. A Method for Automatic Cryptographic Protocol Verification (Extended Abstract), invited paper. In *Fifth International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'2000)*, Cancún, Mexique, May 2000. Springer-Verlag.

[21] 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.

[22] M. Hennessy and J. Riely. Information Flow vs. Resource Access in the Asynchronous Pi-Calculus. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, Lecture Notes on Computer Science, pages 415–427. Springer Verlag, 2000.

[23] D. Kindred and J. M. Wing. Fast, Automatic Checking of Security Protocols. In *USENIX 2nd Workshop on Electronic Commerce*, pages 41–52, Nov. 1996.

[24] 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. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

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

[26] C. Meadows. A Model of Computation for the NRL Protocol Analyzer. In *Proceedings of 1994 Computer Security Foundations Workshop (CSFW-7)*, Franconia, New Hampshire, June 1994. IEEE Computer Society.

[27] J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, Feb. 1987.

[28] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[29] D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. In *Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes on Computer Science*, pages 149–163. Springer Verlag, Sept. 1999.

[30] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

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

[32] D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *Operating Systems Review*, 21(1):8–10, 1987.

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

[34] 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.

[35] D. X. Song. Athena: a New Efficient Automatic Checker for Security Protocol Analysis. In *Proc. of 12th IEEE Computer Security Foundation Workshop (CSFW-12)*, Mordano, Italy, June 1999.

[36] P. Syverson. A Taxonomy of Replay Attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop (CSFW-94)*, pages 131–136, 1994.

[37] H. Tamaki and T. Sato. Unfold/Fold Transformation of Logic Programs. In S. Åke Tärnlund, editor, *Proceedings of the Second International Logic Programming Conference (ICLP'84)*, pages 127–138, Uppsala, Sweden, July 1984.

A. Proof of correctness of our algorithm

Lemma 4 *At the end of the first phase, B satisfies the following properties:*

1. $\forall R \in B_0, \exists R' \in B, R' \Rightarrow R;$
2. *Let $R \in B$, $R = H \to C$ and $R' \in B$, $R' = H' \to C'$. Assume that there exists $F_0 \in H'$ such that:*
   (a) $R \circ_{F_0} R'$ *is defined;*
   (b) $\forall F \in H, F \in_{\mathsf{r}} S;$
   (c) $F_0 \notin_{\mathsf{r}} S.$

   In this case, there exists $R'' \in B$, $R'' \Rightarrow R \circ_{F_0} R'$.

Proof To prove the first property, let $R \in B_0$. We show that during the whole execution of phase 1, $\exists R' \in B, R' \Rightarrow R$.

At the beginning, we execute the instruction $B \leftarrow \mathsf{add}(\mathsf{elimdup}(R), B)$. If there exists no $R' \in B$ such that $R' \Rightarrow \mathsf{elimdup}(R)$, $\mathsf{elimdup}(R)$ is added to B. We have $\mathsf{elimdup}(R) \Rightarrow R$ (with σ the identity). Therefore, after the execution of this instruction $\exists R' \in B, R' \Rightarrow R$.

Assume that we execute $B \leftarrow \mathsf{add}(R'', B)$, and before this execution $\exists R' \in B, R' \Rightarrow R$. Either R' is kept in B, then this property is true after the execution of add. Or R' is removed, and $R'' \Rightarrow R'$. Then $R'' \Rightarrow R$ ($\Rightarrow$ is transitive) and the property is still satisfied.

The second property simply means that the fixed point is reached at the end of phase 1 (using $\mathsf{elimdup}(R \circ_{F_0} R') \Rightarrow R \circ_{F_0} R'$). □

Lemma 5 *If $R \circ_{F_0} R'$ is defined, $R_1 \Rightarrow R$ and $R'_1 \Rightarrow R'$ then either there exists F_1 such that $R_1 \circ_{F_1} R'_1$ is defined and $R_1 \circ_{F_1} R'_1 \Rightarrow R \circ_{F_0} R'$, or $R'_1 \Rightarrow R \circ_{F_0} R'$.*

Proof Let $R = H \to C$, $R' = H' \to C'$, $R_1 = H_1 \to C_1$, $R'_1 = H'_1 \to C'_1$. By renaming the variables, we can arrange such that the variables of R_1 and R'_1 are distinct. Then there exists a substitution σ such that $\sigma C_1 = C$, $\sigma H_1 \subseteq H, \sigma C'_1 = C', \sigma H'_1 \subseteq H'$.

We have $R \circ_{F_0} R' = \sigma'(H \cup (H' - F_0)) \to \sigma' C'$. We have two cases.

First case: $\exists F_1 \in H'_1, \sigma F_1 = F_0$. Since $R \circ_{F_0} R'$ is defined, F_0 and C are unifiable, let σ' be the most general unifier. $\sigma'\sigma F_1 = \sigma'\sigma C_1$, then F_1 and C_1 are unifiable, therefore $R_1 \circ_{F_1} R'_1$ is defined. Let σ_1 be the most general unifier. There exists σ'_1 such that $\sigma'\sigma = \sigma'_1\sigma_1$. We have $R_1 \circ_{F_1} R'_1 = \sigma_1(H_1 \cup (H'_1 - F_1)) \to \sigma_1 C'_1$, $\sigma'_1\sigma_1(H_1 \cup (H'_1 - F_1)) = \sigma'\sigma(H_1 \cup (H'_1 - F_1)) \subseteq \sigma'(H \cup (H' - F_0))$, $\sigma'_1\sigma_1 C'_1 = \sigma'\sigma C'_1 = \sigma' C'$. Then $R_1 \circ_{F_1} R'_1 \Rightarrow R \circ_{F_0} R'$.

Second case: $\sigma H'_1 \subseteq H' - F_0$. Then $\sigma H'_1 \subseteq (H \cup (H' - F_0))$ and $\sigma C'_1 = C'$. Therefore $R'_1 \Rightarrow R \circ_{F_0} R'$. □

Lemma 1 (Correctness of phase 1) *Let F be a closed fact. F is derivable from B_0 if and only if F is derivable from B'.*

Proof Assume that F is derivable from B_0 and consider a derivation of F from B_0. For each rule R in B_0, there exists a rule R' in B such that $R' \Rightarrow R$ (Lemma 4, Property 1).

Assume that R is the label of a node with an incoming edge labelled F and n outgoing edges labelled $F_1, \dots, F_n$. We have $R \Rightarrow \{F_1, \dots, F_n\} \to F$. Then $R' \Rightarrow \{F_1, \dots, F_n\} \to F$ ($\Rightarrow$ transitive).

Therefore, we can replace the node labelled R by a node labelled R'. This way, we obtain a derivation of F from B.

Assume that there are two nodes n and n' in this derivation of F, linked by an edge from n' to n labelled C_1. Assume that n is labelled R and n' is labelled R'. Let H be the set of labels of outgoing edges of n, H' the same for n', C' the label of the incoming edge of n'. Then $(H \to C_1) \circ_{C_1} (H' \to C')$ is defined (with a substitution σ being the identity). By Lemma 5, there exists F such that $R \circ_F R'$ is defined, and two cases may arise: either $R \circ_F R' \Rightarrow (H \to C_1) \circ_{C_1} (H' \to C')$, or $R' \Rightarrow (H \to C_1) \circ_{C_1} (H' \to C')$.

- In the first case, assume that the hypotheses (b) and (c) of Lemma 4, Property 2 are satisfied. Then there exists $R'' \in B$ such that $R'' \Rightarrow R \circ_F R'$. Then $R'' \Rightarrow (H \to C_1) \circ_{C_1} (H' \to C') = H \cup (H' - C_1) \to C'$ ($\Rightarrow$ transitive). Then the two nodes n and n' can be replaced by a node n'' labelled R''. Indeed, the outgoing edges of n and n' (excluding the edge from n' to n) are labelled by elements of H and $H' - C_1$. And the incoming edge of n' is labelled by C'.
- In the second case, we remove n, and link directly its incoming and outgoing edges to n'. We have $R' \Rightarrow (H \to C_1) \circ_{C_1} (H' \to C') = H \cup (H' - C_1) \to C'$, and outgoing edges of n' are now labelled by elements of $H \cup (H' - C_1)$, its incoming edge by C'.

We perform this replacement process as long as there exist nodes on which it can be applied. Once the replacement process is done, we show that the remaining rules are all in B'.

- The rules labelling leaves of the tree are all in B' since they have no hypotheses.
- Let n' be a node such that all sons of n' are labelled by a rule in B'. Therefore, the hypothesis (b) is satisfied for all sons n of n' (the hypotheses F of the rule R labelling n satisfy $F \in_{\mathsf{r}} S$). Since n and n' cannot have been merged with another node by the above replacement process, hypothesis (c) is not satisfied for all sons n of n'. Then all hypotheses F_0 of the rule labelling n' satisfy $F_0 \in_{\mathsf{r}} S$. That is, n' is also labelled by a rule of B'.

By induction, this proof shows that all nodes are labelled by a rule of B', which is the expected result.

For the converse implication, notice that if a fact is derivable from B' then it is derivable from B, and that all rules added to B do not create new derivable terms: when composing two rules R and R', the created rule can derive terms that could also by derived by R and R'. □

Theorem 2 *Let F be a closed fact. Let F' such that there exists a substitution σ such that $\sigma F' = F$. F is derivable from the rules in B_0 if and only if $\exists F'' \in \mathsf{derivable}(F'), \exists \sigma, F = \sigma F''$.*

In particular, F is derivable from B_0 if and only if $F \in \mathsf{derivable}(F)$.

Proof Using Lemma 1, we only have to prove that F is derivable from B' if and only if $\exists F'' \in \mathsf{derivable}(F'), \exists \sigma, F = \sigma F''$.

Let us prove the direct implication. We consider a derivation of F from B'. We cut this derivation on certain edges, and remove the branches that start from these edges. We call the remaining part a partial derivation of F. Let $F_1, \dots, F_n$ be the labels of the cut edges. We prove that $\exists R', R' \Rightarrow \{F_1, \dots, F_n\} \to F, \mathsf{derivablerec}(R', B'') \subseteq \mathsf{derivable}(F')$ and $\forall R'' \in B'', R'' \not\Rightarrow R'$. The proof is by induction on the number of nodes in the partial derivation.

If there are no nodes in the partial derivation, that is we have cut the edge starting from the root, let $R' =$

$\{F'\} \to F'$, $B'' = \emptyset$. We have $\mathsf{derivablerec}(R', B'') = \mathsf{derivable}(F')$ hence the result.

For the induction step, consider a partial derivation with $k+1$ nodes. Let n be a node of this derivation whose all outgoing edges have been cut (a leaf of the partial derivation). Assume that n is labelled by R, that its incoming edge is labelled by F_1, its outgoing edges by $F'_1, \dots, F'_{n'}$. The other edges that have been cut to build the partial derivation are labelled by $F_2, \dots F_n$. By induction hypothesis on the partial derivation without node n, there exists R' such that $R' \Rightarrow \{F_1, \dots, F_n\} \to F$, $\mathsf{derivablerec}(R', B'') \subseteq \mathsf{derivable}(F')$ and $\forall R'' \in B'', R'' \not\Rightarrow R'$. We show that there exists R_f such that $R_f \Rightarrow \{F'_1, \dots, F'_{n'}, F_2, \dots, F_n\} \to F$, $\mathsf{derivablerec}(R_f, B'') \subseteq \mathsf{derivable}(F')$ and $\forall R'' \in B'', R'' \not\Rightarrow R_f$. By definition of a derivation, $R \Rightarrow \{F'_1, \dots, F'_{n'}\} \to F_1$. Notice that the composition $(\{F'_1, \dots, F'_{n'}\} \to F_1) \circ_{F_1} (\{F_1, \dots, F_n\} \to F) = \{F'_1, \dots F'_{n'}, F_2, \dots, F_n\} \to F$ is defined (the unifier σ being the identity). Then by Lemma 5, two cases may arise. First case: $R' \Rightarrow \{F'_1, \dots F'_{n'}, F_2, \dots, F_n\} \to F$, and the expected result is obvious with $R_f = R'$. Second case: there exists F'' such that $R \circ_{F''} R' \Rightarrow \{F'_1, \dots F'_{n'}, F_2, \dots, F_n\} \to F$. Let $R_0 = \mathsf{elimdup}(R \circ_{F''} R')$. Then, by transitivity of $\Rightarrow$, $R_0 \Rightarrow \{F'_1, \dots F'_{n'}, F_2, \dots, F_n\} \to F$. By the step (c) of the definition of derivablerec, $\mathsf{derivablerec}(R_0, \{R'\} \cup B'') \subseteq \mathsf{derivable}(F')$. If $\forall R_1 \in \{R'\} \cup B'', R_1 \not\Rightarrow R_0$, we have the expected result with $R_f = R_0$. Otherwise, $\exists R_1 \in \{R'\} \cup B'', R_1 \Rightarrow R_0$. By transitivity of $\Rightarrow$, $R_1 \Rightarrow \{F'_1, \dots F'_{n'}, F_2, \dots, F_n\} \to F$. There is an older call to derivablerec, of the form $\mathsf{derivablerec}(R_1, B_1)$, such that $\mathsf{derivablerec}(R_1, B_1) \subseteq \mathsf{derivable}(F')$. If B_1 satisfies $\forall R_2 \in B_1, R_2 \not\Rightarrow R_1$, we have the result with $R_f = R_1$. Otherwise, we go on taking a previous call to derivablerec as above. The process terminates, since B'' is finite.

We can apply the result we have just proved to the particular case when the partial derivation is in fact the whole derivation of F. We obtain $\exists R', R' \Rightarrow \emptyset \to F$ and $\mathsf{derivablerec}(R', B'') \subseteq \mathsf{derivable}(F')$, $\forall R'' \in B'', R'' \not\Rightarrow R'$. Therefore $R' = \emptyset \to F''$, with $\sigma F'' = F$. $\mathsf{derivablerec}(R', B'') = \{F''\}$. (The case (a) of the definition of derivablerec cannot be applied because of the condition $\forall R'' \in B'', R'' \not\Rightarrow R'$.) Then $F'' \in \mathsf{derivable}(F')$. We have the expected result.

The proof of the converse inclusion is left to the reader. (essentially, the rule $R \circ_F R'$ does not generate facts that cannot be generated by applying R and R'). □

B. Termination

Lemma 3 *If F is closed and $S = \{\mathsf{attacker}(x)\}$, then* $\mathsf{derivable}(F)$ *terminates.*

Proof $\mathsf{derivablerec}(R, B'')$ is only called with $R = \{F\} \to F$ or $R = \mathsf{attacker}(M_1) \wedge \dots \wedge \mathsf{attacker}(M_n) \to F$ where $M_1, \dots, M_n$ are closed terms, or a variable that appears only once. This is proved by induction in the following. Moreover, we prove that the pair $p =$ (total size of the $M_1, \dots, M_n$ that are closed terms, number of $M_1, \dots, M_n$ that are variables) ordered lexicographically strictly decreases. This decrease proves the termination.

At the beginning, the rule is indeed $R = \{F\} \to F$. For recursive calls to derivablerec, the rule is $R' \circ_{F_0} R$, where $R' = \mathsf{attacker}(x_1) \wedge \dots \wedge \mathsf{attacker}(x_k) \to F'$.

1. First case: F_0 is a closed fact.

After unification of F' and F_0, x_i is substituted by a closed term N_i if x_i appears in F'. Otherwise, x_i remains unchanged, and we define $N_i = x_i$.

- If $R = \{F\} \to F$, the resulting rule $R' \circ_{F_0} R$ is $\mathsf{attacker}(N_1) \wedge \dots \wedge \mathsf{attacker}(N_k) \to F$.
- Otherwise, $R = \mathsf{attacker}(M_1) \wedge \dots \wedge \mathsf{attacker}(M_n) \to F$, $F_0 = \mathsf{attacker}(M_i)$. Assume that M_i is a closed term. The resulting rule is $\mathsf{attacker}(N_1) \wedge \dots \wedge \mathsf{attacker}(N_k) \wedge \mathsf{attacker}(M_1) \wedge \dots \wedge \mathsf{attacker}(M_{i-1}) \wedge \mathsf{attacker}(M_{i+1}) \wedge \dots \wedge \mathsf{attacker}(M_n) \to F$. Moreover, the $N_1, \dots, N_k$ that are closed terms are disjoint subterms of M_i, therefore the total size of the $N_1, \dots, N_k$ that are closed terms is strictly smaller than the size of M_i (except when $R' = \mathsf{attacker}(x) \to \mathsf{attacker}(x)$, but in this case, $R' \circ_{F_0} R = R$, and the call $\mathsf{derivablerec}(R' \circ_{F_0} R, \{R\} \cup B'')$ stops immediately because $R' \circ_{F_0} R = R \in \{R\} \cup B''$, by the first point of the definition of derivablerec). Therefore the total size of the closed terms in the hypotheses strictly decreases. Hence the pair p ordered lexicographically strictly decreases.

2. Second case: $R = \mathsf{attacker}(M_1) \wedge \dots \wedge \mathsf{attacker}(M_n) \to F$, $F_0 = \mathsf{attacker}(M_i)$, $M_i = x_i$.

If R' has some hypotheses, the resulting rule is $R' \circ_{F_0} R = \mathsf{attacker}(x_1) \wedge \dots \wedge \mathsf{attacker}(x_k) \wedge \mathsf{attacker}(M_1) \wedge \dots \wedge \mathsf{attacker}(M_{i-1}) \wedge \mathsf{attacker}(M_{i+1}) \wedge \dots \wedge \mathsf{attacker}(M_n) \to F$. We clearly have $R' \circ_{F_0} R \Rightarrow R$. Therefore the call $\mathsf{derivablerec}(R' \circ_{F_0} R, \{R\} \cup B'')$ stops immediately because $R' \circ_{F_0} R \Rightarrow R$ and $R \in \{R\} \cup B''$.

If R' has no hypothesis, the resulting rule is $R' \circ_{F_0} R = \mathsf{attacker}(M_1) \wedge \dots \wedge \mathsf{attacker}(M_{i-1}) \wedge \mathsf{attacker}(M_{i+1}) \wedge \dots \wedge \mathsf{attacker}(M_n) \to F$, and the total size of closed terms in the hypotheses is constant, whereas the number of variables strictly decreases. Hence the pair p ordered lexicographically strictly decreases. □

Remark. The cases $R' = \mathsf{attacker}(x) \to \mathsf{attacker}(x)$ and $F_0 = \mathsf{attacker}(x_i)$ are removed by the optimizations of Section 5.2.