Using Horn Clauses for Analyzing Security Protocols

Bruno BLANCHET [1]
CNRS, École Normale Supérieure, INRIA, Paris, France

Abstract. This chapter presents a method for verifying security protocols based on an abstract representation of protocols by Horn clauses. This method is the foundation of the protocol verifier ProVerif. It is fully automatic, efficient, and can handle an unbounded number of sessions and an unbounded message space. It supports various cryptographic primitives defined by rewrite rules or equations. Even if we focus on secrecy in this chapter, this method can also prove other security properties, including authentication and process equivalences.

Keywords. Automatic verification, security protocols, Horn clauses, secrecy.

Introduction

Security protocols can be verified by an approach based on Horn clauses; the main goal of this approach is to prove security properties of protocols in the Dolev-Yao model in a fully automatic way without bounding the number of sessions or the message space of the protocol. In contrast to the case of a bounded number of sessions in which decidability results could be obtained (see Chapters *" Verifying a bounded number of sessions and its complexity"* and *"Constraint solving techniques and enriching the model with equational theories"*), the case of an unbounded number of sessions is undecidable for a reasonable model of protocols [56]. Possible solutions to this problem are relying on user interaction, allowing non-termination, and performing sound approximations (in which case the technique is incomplete: correct security properties cannot always be proved). Theorem proving [84] and logics (Chapter *" Protocol Composition Logic"*) rely on user interaction or on manual proofs. Typing (Chapter *"Using types for security protocol analysis"*) generally relies on lightweight user annotations and is incomplete. Strand spaces (Chapter *"Shapes: Surveying Crypto Protocol Runs"*) and rank functions (Chapter *"Security analysis using rank functions in CSP"*) also provide techniques that can handle an unbounded number of sessions at the cost of incompleteness.

Many methods rely on sound abstractions [50]: they overestimate the possibilities of attacks, most of the time by computing an overapproximation of the attacker knowledge. They make it possible to obtain fully automatic, but incomplete, systems. The Horn clause approach is one such method. It was first introduced by Weidenbach [86]. This chapter presents a variant of his method and extensions that are at the basis of the auto-

[1] Corresponding Author: Bruno Blanchet, École Normale Supérieure, DI, 45 rue d'Ulm, 75005 Paris, France; E-mail: blanchet@di.ens.fr.

matic protocol verifier ProVerif that we developed. In this method, messages are represented by terms M; the fact $\mathsf{attacker}(M)$ means that the attacker may have the message M; Horn clauses (i.e. logic programming rules) give implications between these facts. An efficient resolution algorithm determines whether a fact is derivable from the clauses, which can be used for proving security properties. In particular, when $\mathsf{attacker}(M)$ is not derivable from the clauses, the attacker cannot have M, that is, M is secret. This method is incomplete since it ignores the number of repetitions of each action in the protocol. (Horn clauses can be applied any number of times.) This abstraction is key to avoid bounding the number of runs of the protocol. It is sound, 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 in practice, as experiments demonstrate. Termination is not guaranteed in general, but it is guaranteed on certain subclasses of protocols and can be obtained in all cases by an additional approximation (see Section 2.4).

Without this additional approximation, even if it does not always terminate and is incomplete, this method provides a good balance in practice: it terminates in the vast majority of cases and is very efficient and precise. It can handle a wide variety of cryptographic primitives defined by rewrite rules or by equations, including shared-key and public-key cryptography (encryption and signatures), hash functions, and the Diffie-Hellman key agreement. It can prove various security properties (secrecy, authentication, and process equivalences). We mainly focus on secrecy in this chapter and give references for other properties in Section 3.2.

Other methods rely on abstractions:

- Bolignano [40] was a precursor of abstraction methods for security protocols. He merges keys, nonces, ... so that only a finite set remains and applies a decision procedure.
- Monniaux [80] introduced a verification method based on an abstract representation of the attacker knowledge by tree automata. This method was extended by Goubault-Larrecq [62]. Genet and Klay [59] combine tree automata with rewriting. This method has lead to the implementation of the TA4SP verifier (*Tree-Automata-based Automatic Approximations for the Analysis of Security Protocols*) [39].
  The main drawback of this approach is that, in contrast to Horn clauses, tree automata cannot represent relational information on messages: when a variable appears several times in a message, one forgets that it has the same value at all its occurrences, which limits the precision of the analysis. The Horn clause method can be understood as a generalization of the tree automata technique. (Tree automata can be encoded into Horn clauses.)
- Control-flow analysis [36,38] computes the possible messages at each program point. It is also non-relational, and merges nonces created at the same program point in different sessions. These approximations make it possible to obtain a complexity at most cubic in the size of the protocol. It was first defined for secrecy for shared-key protocols, then extended to message authenticity and public-key protocols [37], with a polynomial complexity.
- Most protocol verifiers compute the knowledge of the attacker. In contrast, Hermès [41] computes the form of messages, for instance encryption under certain keys, that guarantee the preservation of secrecy. The paper handles shared-key

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

Figure 1. Syntax of our protocol representation

and public-key encryption, but the method also applies to signatures and hash functions.

- Backes et al. [15] prove secrecy and authentication by an abstract-interpretation-based analysis. This analysis builds a causal graph that captures the causality between events in the protocol. The security properties are proved by traversing this graph. This analysis always terminates but is incomplete. It assumes that messages are typed, so that names (which represent random numbers) can be distinguished from other messages.

One of the first verification methods for security protocols, the Interrogator [79] is also related to the Horn clause approach: in this system, written in Prolog, the reachability of the state after a sequence of messages is represented by a predicate, and the program uses a backward search in order to determine whether a state is reachable or not. The main problem of this approach is non-termination, and it is partly solved by relying on user interaction to guide the search. In contrast, we provide a fully automatic approach by using a different resolution strategy that provides termination in most cases.

The NRL protocol analyzer [77,57] improves the technique of the Interrogator by using narrowing on rewriting systems. It does not make abstractions, so it is correct and complete but may not terminate.

Overview Section 1 details our protocol representation. Section 2 describes our resolution algorithm, and sketches its proof of correctness. Several extensions of this work are detailed in Section 3. Section 4 presents experimental results and Section 5 concludes.

1. Abstract Representation of Protocols by Horn Clauses

A protocol is represented by a set of Horn clauses; the syntax of these clauses is given in Figure 1. In this figure, x ranges over variables, a over names, f over function symbols, and p over predicate symbols. The terms M represent messages that are exchanged between participants of the protocol. A variable can represent any term. Names represent atomic values, such as keys and nonces (random numbers). Each principal has the ability of creating new names: fresh names are created at each run of the protocol. Here, the created names are considered as functions of the messages previously received by the principal that creates the name. Thus, names are distinguished only when the preceding messages are different. As noticed by Martín Abadi (personal communication), this approximation is in fact similar to the approximation done in some type systems (such as [2]): 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. In particular, by adding as parameter a session identifier that takes a different value in each run of the protocol, one can distinguish all names. This is necessary for proving authentication but not for secrecy, so we omit session identifiers here for simplicity. We refer the reader to [32,58] for additional information. The function applications $f(M_1, \ldots, M_n)$ build terms: examples of functions are encryption and hash functions. A fact $F = p(M_1, \ldots, M_n)$ expresses a property of the messages $M_1, \ldots, M_n$. Several predicates p can be used but, for a first example, we are going to use a single predicate $\mathsf{attacker}$, such that the fact $\mathsf{attacker}(M)$ means "the attacker may have the message M". A clause $R = F_1 \wedge \ldots \wedge F_n \Rightarrow F$ means that, if all facts $F_1, \ldots, F_n$ are true, then F is also true. A clause with no hypothesis $\Rightarrow F$ is written simply F.

We use as a running example the naive handshake protocol introduced in Example 1 of Chapter *"Introduction"*:

$$\begin{array}{ll} \text{Message 1.} & A \to B : \left\{\left|[k]_{sk_A}\right|\right\}^{\mathsf{a}}_{pk_B} \\ \text{Message 2.} & B \to A : \{\!|s|\!\}^{\mathsf{s}}_{k} \end{array}$$

We refer the reader to Chapter *"Introduction"* for an explanation of this protocol. We denote by sk_A the secret key of A, pk_A his public key, sk_B the secret key of B, pk_B his public key.

1.1. 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 that 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 is defined by a set $\mathrm{def}(g)$ of rewrite rules of the form $g(M_1, \ldots, M_n) \to M$ where $M_1, \ldots, M_n, M$ are terms that contain only variables and constructors and the variables of M all occur in $M_1, \ldots, M_n$. For instance, the decryption $\mathsf{pdecrypt}$ is a destructor, defined by $\mathsf{pdecrypt}(\mathsf{pencrypt}(m, \mathsf{pk}(sk)), sk) \to m$. This rewrite rule models that, by decrypting a ciphertext with the corresponding secret key, one obtains the cleartext. Other functions are defined similarly:

- For signatures, we use a constructor sign and write $\mathsf{sign}(m, sk)$ for the message m signed under the secret key sk. A destructor $\mathsf{getmess}$ defined by $\mathsf{getmess}(\mathsf{sign}(m, sk)) \to m$ returns the message without its signature, and $\mathsf{checksign}(\mathsf{sign}(m, sk), \mathsf{pk}(sk)) \to m$ returns the message only if the signature is valid.
- The shared-key encryption is a constructor $\mathsf{sencrypt}$ and the decryption is a destructor $\mathsf{sdecrypt}$, defined by $\mathsf{sdecrypt}(\mathsf{sencrypt}(m, k), k) \to m$.
- A one-way hash function is represented by a constructor h (and no destructor).

- Tuples of arity n are represented by a constructor $(\_, \ldots, \_)$ and n destructors $i\mathsf{th}_n$ defined by $i\mathsf{th}_n((x_1, \ldots, x_n)) \rightarrow x_i$, $i \in \{1, \ldots, n\}$. Tuples can be used to represent various data structures in protocols.

Rewrite rules offer a flexible method for defining many cryptographic primitives. It can be further extended by using equations, as sketched in Section 3.1.

1.2. 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, following the so-called Dolev-Yao model [55]. We first present the encoding of the computation abilities of the attacker. The encoding of the protocol will be detailed in Section 1.3.

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

$$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n)).$$

If g is a destructor, for each rewrite rule $g(M_1, \ldots, M_n) \rightarrow M$ in $\mathrm{def}(g)$, we have the clause:

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

The destructors never appear in the clauses, they are coded by pattern-matching on their parameters (here $M_1, \ldots, M_n$) in the hypothesis of the clause and generating their result in the conclusion. In the particular case of public-key encryption, this yields:

$$\begin{aligned}
&\mathsf{attacker}(m) \wedge \mathsf{attacker}(pk) \Rightarrow \mathsf{attacker}(\mathsf{pencrypt}(m, pk)),\\
&\mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(\mathsf{pk}(sk)),\\
&\mathsf{attacker}(\mathsf{pencrypt}(m, \mathsf{pk}(sk))) \wedge \mathsf{attacker}(sk) \Rightarrow \mathsf{attacker}(m), \qquad (1)
\end{aligned}$$

where the first two clauses correspond to the constructors pencrypt and pk, and the last clause 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 cleartext m. (We assume that the cryptography is perfect, hence the attacker can obtain the cleartext from the encrypted message only if it has the key.)

Clauses for signatures (sign, getmess, checksign) and for shared-key encryption (sencrypt, sdecrypt) are given in Figure 2.

The clauses above describe the computation abilities of the attacker. Moreover, the attacker initially has the public keys of the protocol participants. Therefore, we add the clauses $\mathsf{attacker}(\mathsf{pk}(sk_A[\,]))$ and $\mathsf{attacker}(\mathsf{pk}(sk_B[\,]))$. We also give a name a to the attacker, that will represent all names it can generate: $\mathsf{attacker}(a[\,])$. In particular, $a[\,]$ can represent the secret key of any dishonest participant, his public key being $\mathsf{pk}(a[\,])$, which the attacker can compute by the clause for constructor pk.

1.3. 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 of starting the protocol with the principal it wants, that is, the attacker will send a preliminary message to A, mentioning the public key of the principal with which A should talk. This principal can be B, or another principal represented by the attacker. Hence, if the attacker has some key $\mathsf{pk}(x)$, it can send $\mathsf{pk}(x)$ to A; A replies with his first message, which the attacker can intercept, so the attacker obtains $\mathsf{pencrypt}(\mathsf{sign}(k, sk_A[\,]), \mathsf{pk}(x))$. Therefore, we have a clause of the form

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

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 clause becomes:

$$\mathsf{attacker}(\mathsf{pk}(x)) \Rightarrow \mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(x)], sk_A[\,]), \mathsf{pk}(x))). \qquad (2)$$

When B receives a message, he decrypts it with his secret key sk_B, so B expects a message of the form $\mathsf{pencrypt}(x', \mathsf{pk}(sk_B[\,]))$. Next, B tests whether A has signed x', that is, B evaluates $\mathsf{checksign}(x', pk_A)$, and this succeeds only when $x' = \mathsf{sign}(y, sk_A[\,])$. If so, he assumes that the key y is only known by A, and sends a secret s (a constant that the attacker does not have a priori) encrypted under y. We assume that the attacker relays the message coming from A, and intercepts the message sent by B. Hence the clause:

$$\mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(y, sk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \mathsf{attacker}(\mathsf{sencrypt}(\mathsf{s}, y)).$$

Remark 1 With these clauses, A cannot play the role of B and vice-versa. In order to model a situation in which all principals play both roles, we can replace all occurrences of $sk_B[\,]$ with $sk_A[\,]$ in the clauses above. Then A plays both roles, and is the only honest principal. A single honest principal is sufficient for proving secrecy properties by [48].

More generally, a protocol that contains n messages is encoded by n sets of clauses. If a principal X sends the ith message, the ith set of clauses contains clauses 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 function defined by several rewrite rules, such as the function exp of Section 3.1. In this case, a clause must be generated for each combination of possible patterns. Moreover, notice that the hypotheses of the clauses describe all 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.

Computation abilities of the attacker:

For each constructor f of arity n:

$$\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \Rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n))$$

For each destructor g, for each rewrite rule $g(M_1, \ldots, M_n) \rightarrow M$ in $\mathrm{def}(g)$:

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

that is

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	$\mathsf{attacker}(\mathsf{sign}(m, sk)) \wedge \mathsf{attacker}(\mathsf{pk}(sk)) \Rightarrow \mathsf{attacker}(m)$
sencrypt	$\mathsf{attacker}(m) \wedge \mathsf{attacker}(k) \Rightarrow \mathsf{attacker}(\mathsf{sencrypt}(m, k))$
sdecrypt	$\mathsf{attacker}(\mathsf{sencrypt}(m, k)) \wedge \mathsf{attacker}(k) \Rightarrow \mathsf{attacker}(m)$
Name generation:	$\mathsf{attacker}(a[\,])$

Initial knowledge: $\mathsf{attacker}(\mathsf{pk}(sk_A[\,])), \quad \mathsf{attacker}(\mathsf{pk}(sk_B[\,]))$

The 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:
$$\mathsf{attacker}(\mathsf{pencrypt}(\mathsf{sign}(y, sk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \mathsf{attacker}(\mathsf{sencrypt}(\mathsf{s}, y))$$

Figure 2. Summary of our representation of the protocol of Example 1 of Chapter *"Introduction"*

Remark 2 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: $\mathsf{message}(C, M)$ 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 clauses have to be added to describe the behavior of the attacker. The attacker can listen on all channels it has: $\mathsf{message}(x, y) \wedge \mathsf{attacker}(x) \Rightarrow \mathsf{attacker}(y)$. It can send all messages it has on all channels it has: $\mathsf{attacker}(x) \wedge \mathsf{attacker}(y) \Rightarrow \mathsf{message}(x, y)$.

1.4. Summary

To sum up, a protocol can be represented by three sets of Horn clauses, as detailed in Figure 2 for the protocol of Example 1 of Chapter *"Introduction"*:

- Clauses representing the computation abilities of the attacker: constructors, destructors, and name generation.
- Facts corresponding to the initial knowledge of the attacker. In general, there are facts giving the public keys of the participants and/or their names to the attacker.
- Clauses representing the messages of the protocol itself. There is one set of clauses for each message in the protocol. In the set corresponding to the ith message, sent by principal X, the clauses 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}, ..., 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.

1.5. Approximations

The reader can notice that our representation of protocols is approximate. Specifically, the number of repetitions of each action is ignored, since Horn clauses can be applied any number of times. So 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 (even when future steps have already been completed). For instance, consider the following protocol (communicated by Véronique Cortier)

First step:	A sends $\{\!\|\langle N_1, M\rangle\|\!\}_k^{\mathsf{s}}$, $\{\!\|\langle N_2, M\rangle\|\!\}_k^{\mathsf{s}}$
Second step:	If A receives $\{\!\|\langle x, M\rangle\|\!\}_k^{\mathsf{s}}$, he replies with x
Third step:	If A receives N_1, N_2, he replies with s

where N_1, N_2, and M are nonces. In an exact model, A never sends s, since $\{\!\|\langle N_1, M\rangle\|\!\}_k^{\mathsf{s}}$ or $\{\!\|\langle N_2, M\rangle\|\!\}_k^{\mathsf{s}}$ can be decrypted, but not both. In the Horn clause model, even though the first step is executed once, the second step may be executed twice for the same M (that is, the corresponding clause can be applied twice), so that both $\{\!\|\langle N_1, M\rangle\|\!\}_k^{\mathsf{s}}$ and $\{\!\|\langle N_2, M\rangle\|\!\}_k^{\mathsf{s}}$ can be decrypted, and A may send s. We have a false attack against the secrecy of s.

However, the important point is that the approximations are sound: if an attack exists in a more precise model, such as the applied pi calculus [6] or multiset rewriting [52], then it also exists in our representation. This is shown for the applied pi calculus in [3] and for multiset rewriting in [29]. In particular, we have shown formally that the only approximation with respect to the multiset rewriting model is that the number of repetitions of actions is ignored. 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 is that false attacks may be found by the verifier: sequences of clause applications that do not correspond to a protocol run, as illustrated above. False attacks appear in particular for protocols with temporary secrets: when some value first needs to be kept secret and is revealed later in the protocol, the Horn clause model considers that this value can be reused in the beginning of the protocol, thus breaking the protocol. 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. Fortunately, our representation is precise enough so that false attacks are rare. (This is demonstrated by our experiments, see Section 4.)

1.6. Secrecy Criterion

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

Our notion of secrecy is similar to that of [2,36,44]: 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 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 [1, Section 6] for further discussion and references.)

In our running example, $\mathsf{attacker}(\mathsf{s})$ is derivable from the clauses. The derivation is as follows. The attacker generates a fresh name $a[\,]$ (considered as a secret key), it computes $\mathsf{pk}(a[\,])$ by the clause for pk, obtains $\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,]), \mathsf{pk}(a[\,]))$ by the clause for the first message. It decrypts this message using the clause for pdecrypt and its knowledge of $a[\,]$, thus obtaining $\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,])$. It reencrypts the signature under $\mathsf{pk}(sk_B[\,])$ by the clause for pencrypt (using its initial knowledge of $\mathsf{pk}(sk_B[\,])$), thus obtaining $\mathsf{pencrypt}(\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,]), \mathsf{pk}(sk_B[\,]))$. By the clause for the second message, it obtains $\mathsf{sencrypt}(\mathsf{s}, k[\mathsf{pk}(a[\,])])$. On the other hand, from $\mathsf{sign}(k[\mathsf{pk}(a[\,])], sk_A[\,])$, it obtains $k[\mathsf{pk}(a[\,])]$ by the clause for getmess, so it can decrypt $\mathsf{sencrypt}(\mathsf{s}, k[\mathsf{pk}(a[\,])])$ by the clause for sdecrypt, thus obtaining s. In other words, the attacker starts a session between A and a dishonest participant of secret key $a[\,]$. It gets the first message $\mathsf{pencrypt}(\mathsf{sign}(k, sk_A[\,]), \mathsf{pk}(a[\,]))$, decrypts it, reencrypts it under $\mathsf{pk}(sk_B[\,])$, and sends it to B. For B, this message looks like the first message of a session between A and B, so B replies with $\mathsf{sencrypt}(\mathsf{s}, k)$, which the attacker can decrypt since it obtains k from the first message. Hence, the obtained derivation corresponds to the known attack against this protocol. In contrast, if we fix the protocol by adding the public key of B in the first message $\left\{\left|[\langle pk_B, k\rangle]_{sk_A}\right|\right\}^{\mathsf{a}}_{pk_B}$, $\mathsf{attacker}(\mathsf{s})$ is not derivable from the clauses, so the fixed protocol preserves the secrecy of s.

Next, we formally define when a given fact can be derived from a given set of clauses. We shall see in the next section how we determine that. Technically, the hypotheses $F_1, \ldots, F_n$ of a clause are considered as a multiset. This means that the order of the hypotheses is irrelevant, but the number of times a hypothesis is repeated is important. (This is not related to multiset rewriting models of protocols: the semantics of a clause does not depend on the number of repetitions of its hypotheses, but considering multisets is necessary in the proof of the resolution algorithm.) We use R for clauses (logic programming *rules*), H for hypothesis, and C for conclusion.

Definition 1 (Subsumption) *We say that $H_1 \Rightarrow C_1$ subsumes $H_2 \Rightarrow C_2$, and we write $(H_1 \Rightarrow C_1) \sqsupseteq (H_2 \Rightarrow C_2)$, if and only if there exists a substitution σ such that $\sigma C_1 = C_2$ and $\sigma H_1 \subseteq H_2$ (multiset inclusion).*

We write $R_1 \sqsupseteq R_2$ 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.

A derivation is defined as follows, as illustrated in Figure 3.

Definition 2 (Derivability) *Let F be a closed fact, that is, a fact without variable. Let $\mathcal{R}$ be a set of clauses. F is derivable from $\mathcal{R}$ if and only if there exists a derivation of F from $\mathcal{R}$, that is, a finite tree defined as follows:*

1. *Its nodes (except the root) are labeled by clauses $R \in \mathcal{R}$;*
2. *Its edges are labeled by closed facts;*
3. *If the tree contains a node labeled by R with one incoming edge labeled by F_0 and n outgoing edges labeled by $F_1, \ldots, F_n$, then $R \sqsupseteq F_1 \wedge \ldots \wedge F_n \Rightarrow F_0$.*
4. *The root has one outgoing edge, labeled by F. The unique son of the root is named the* subroot.

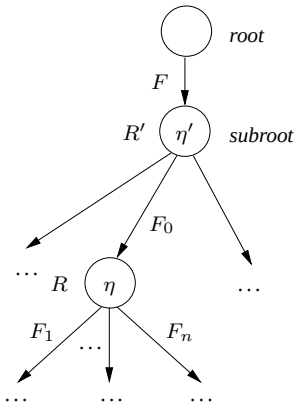

Figure 3. Derivation of F

In a derivation, if there is a node labeled by R with one incoming edge labeled by F_0 and n outgoing edges labeled by $F_1, \ldots, F_n$, then F_0 can be derived from $F_1, \ldots, F_n$ by the clause R. Therefore, there exists a derivation of F from $\mathcal{R}$ if and only if F can be derived from clauses in $\mathcal{R}$ (in classical logic).

2. Resolution Algorithm

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

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

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.

As detailed below, the main idea is to combine pairs of clauses by resolution, and to guide this resolution process by a selection function: our resolution algorithm is resolution with free selection [51,75,14]. This algorithm is similar to ordered resolution with selection, used by [86], but without the ordering constraints.

Notice that, since a term is secret when a fact is *not* derivable from the clauses, soundness in terms of security (if the verifier claims that there is no attack, then there is no attack) corresponds to the completeness of the resolution algorithm in terms of logic programming (if the algorithm claims that a fact is not derivable, then it is not). The resolution algorithm that we use must therefore be complete.

2.1. The Basic Algorithm

Let us first define resolution: when the conclusion of a clause R unifies with a hypothesis of another (or the same) clause R', resolution infers a new clause that corresponds to applying R and R' one after the other. Formally, resolution is defined as follows:

Definition 3 *Let R and R' be two clauses, $R = H \Rightarrow C$, and $R' = H' \Rightarrow C'$. 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 $R \circ_{F_0} R' = \sigma(H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma C'$. The clause $R \circ_{F_0} R'$ is the result of resolving R' with R upon F_0.*

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

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

with the substitution $\sigma = \{sk \mapsto x, m \mapsto \mathsf{sign}(k[\mathsf{pk}(x)], sk_A[\,])\}$.

We guide the resolution by a selection function:

Definition 4 *A* selection function sel *is a function from clauses to sets of facts, such that $sel(H \Rightarrow C) \subseteq H$. If $F \in sel(R)$, we say that F is selected in R. If $sel(R) = \emptyset$, we say that no hypothesis is selected in R, or that the conclusion of R is selected.*

The resolution algorithm is correct (sound and complete) with any selection function, as we show below. However, the choice of the selection function can change dramatically the behavior of the algorithm. The essential idea of the algorithm is to combine clauses by resolution only when the facts unified in the resolution are selected. We will therefore choose the selection function to reduce the number of possible unifications between selected facts. Having several selected facts slows down the algorithm, because it has more choices of resolutions to perform, therefore we will select at most one fact in each clause. In the case of protocols, facts of the form $\mathsf{attacker}(x)$, with x variable, can be unified will all facts of the form $\mathsf{attacker}(M)$. Therefore, we should avoid selecting them. So a basic selection function is a function sel_0 that satisfies the constraint

$$sel_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if } \forall F \in H, \exists x \text{ variable}, F = \mathsf{attacker}(x) \\ \{F_0\} & \text{where } F_0 \in H \text{ and } \forall x \text{ variable}, F_0 \neq \mathsf{attacker}(x) \end{cases} \quad (3)$$

The resolution algorithm works in two phases, described in Figure 4. The first phase transforms the initial set of clauses into a new one that derives the same facts. The second phase uses a depth-first search to determine whether a fact can be derived or not from the clauses.

The first phase, $\mathsf{saturate}(\mathcal{R}_0)$, contains 3 steps.

- The first step inserts in $\mathcal{R}$ the initial clauses representing the protocol and the attacker (clauses that are in $\mathcal{R}_0$), after elimination of subsumed clauses by elim: if R' subsumes R, and R and R' are in $\mathcal{R}$, then R is removed by $\mathsf{elim}(\mathcal{R})$.
- The second step is a fixpoint iteration that adds clauses created by resolution. The resolution of clauses R and R' is added only if no hypothesis is selected in R and the hypothesis F_0 of R' that we unify is selected. When a clause is created by resolution, it is added to the set of clauses $\mathcal{R}$. Subsumed clauses are eliminated from $\mathcal{R}$.
- At last, the third step returns the set of clauses of $\mathcal{R}$ with no selected hypothesis.

Basically, saturate preserves derivability (it is both sound and complete):

First phase: saturation

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

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

Second phase: backward depth-first search

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

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

Figure 4. Resolution algorithm

Lemma 1 (Correctness of saturate**)** *Let F be a closed fact. F is derivable from $\mathcal{R}_0$ if and only if it is derivable from* $\mathsf{saturate}(\mathcal{R}_0)$.

This result is proved by transforming a derivation of F from $\mathcal{R}_0$ into a derivation of F from $\mathsf{saturate}(\mathcal{R}_0)$. Basically, when the derivation contains a clause R' with $sel(R') \neq \emptyset$, we replace in this derivation two clauses R, with $sel(R) = \emptyset$, and R' that have been combined by resolution during the execution of saturate with a single clause $R \circ_{F_0} R'$. This replacement decreases the number of clauses in the derivation, so it terminates, and, upon termination, all clauses of the obtained derivation satisfy $sel(R') = \emptyset$ so they are in $\mathsf{saturate}(\mathcal{R}_0)$. A detailed proof is given in Section 2.2.

Usually, resolution with selection is used for proofs by refutation. That is, the negation of the goal F is added to the clauses, under the form of a clause without conclusion: $F \Rightarrow$. The goal F is derivable if and only if the empty clause "$\Rightarrow$" can be derived. Here, we would like to avoid repeating the whole resolution process for each goal, since in general we prove the secrecy of several values for the same protocol. For non-closed goals, we also want to be able to know which instances of the goal can be derived. That is why we prove that the clauses in $\mathsf{saturate}(\mathcal{R}_0)$ derive the same facts as the clauses in $\mathcal{R}_0$. The set of clauses $\mathsf{saturate}(\mathcal{R}_0)$ can then be used to query several goals, using the second phase of the algorithm described next.

The second phase searches the facts that can be derived from $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$. This is simply a backward depth-first search. The call $\mathsf{derivable}(F, \mathcal{R}_1)$ returns a set of clauses $R = H \Rightarrow C$ with no selected hypothesis, such that R can be obtained by resolution from $\mathcal{R}_1$, C is an instance of F, and all instances of F derivable from $\mathcal{R}_1$ can be derived by using as last clause a clause of $\mathsf{derivable}(F, \mathcal{R}_1)$. (Formally, if F' is an instance of F derivable from $\mathcal{R}_1$, then there exist a clause $H \Rightarrow C \in \mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution σ such that $F' = \sigma C$ and σH is derivable from $\mathcal{R}_1$.)

The search itself is performed by $\mathsf{deriv}(R, \mathcal{R}, \mathcal{R}_1)$. The function deriv starts with $R = F \Rightarrow F$ and transforms the hypothesis of R by using a clause R' of $\mathcal{R}_1$ to derive

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

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

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

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

Lemma 2 (Correctness of derivable**)** *Let F' be a closed instance of F. F' is derivable from $\mathcal{R}_1$ if and only if there exist a clause $H \Rightarrow C$ in $\mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution σ such that $\sigma C = F'$ and all elements of σH are derivable from $\mathcal{R}_1$.*

Basically, this result is proved by transforming a derivation of F' from $\mathcal{R}_1$ into a derivation of F' whose last clause (the one that concludes F') is $H \Rightarrow C$ and whose other clauses are still in $\mathcal{R}_1$. The transformation relies on the replacement of clauses combined by resolution during the execution of derivable. A detailed proof is given in Section 2.2.

It is important to apply saturate before derivable, so that all clauses in $\mathcal{R}_1$ have no selected hypothesis. Then the conclusion of these clauses is in general not $\mathsf{attacker}(x)$ (with the optimizations of Section 2.3 and a selection function that satisfies (3), it is never $\mathsf{attacker}(x)$), so that we avoid unifying with $\mathsf{attacker}(x)$.

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 clauses. The first part simply combines Lemmas 1 and 2. The second part mentions two easy and important particular cases.

Theorem 1 (Correctness) *Let F' be a closed instance of F. F' is derivable from $\mathcal{R}_0$ if and only if there exist a clause $H \Rightarrow C$ in $\mathsf{derivable}(F, \mathsf{saturate}(\mathcal{R}_0))$ and a substitution σ such that $\sigma C = F'$ and all elements of σH are derivable from $\mathsf{saturate}(\mathcal{R}_0)$.*

In particular, if $\mathsf{derivable}(F, \mathsf{saturate}(\mathcal{R}_0)) = \emptyset$, then no instance of F is derivable from $\mathsf{saturate}(\mathcal{R}_0)$. If the selection function satisfies (3) *and F is closed, then F is derivable from $\mathcal{R}_0$ if and only if $\mathsf{derivable}(F, \mathsf{saturate}(\mathcal{R}_0)) \neq \emptyset$.*

Proof:
The first part of the theorem is obvious from Lemmas 1 and 2. The first particular case is also an obvious consequence. For the second particular case, if F is derivable from $\mathcal{R}_0$, then $\mathsf{derivable}(F, \mathsf{saturate}(\mathcal{R}_0)) \neq \emptyset$ by the first particular case. For the converse, suppose that $\mathsf{derivable}(F, \mathsf{saturate}(\mathcal{R}_0)) \neq \emptyset$. Then $\mathsf{derivable}(F, \mathsf{saturate}(\mathcal{R}_0))$ contains a clause $H \Rightarrow C$. By definition of derivable, C is an instance of F, so $C = F$, and $sel(H \Rightarrow C) = \emptyset$, so all elements of H are of the form $\mathsf{attacker}(x_i)$ for some variable

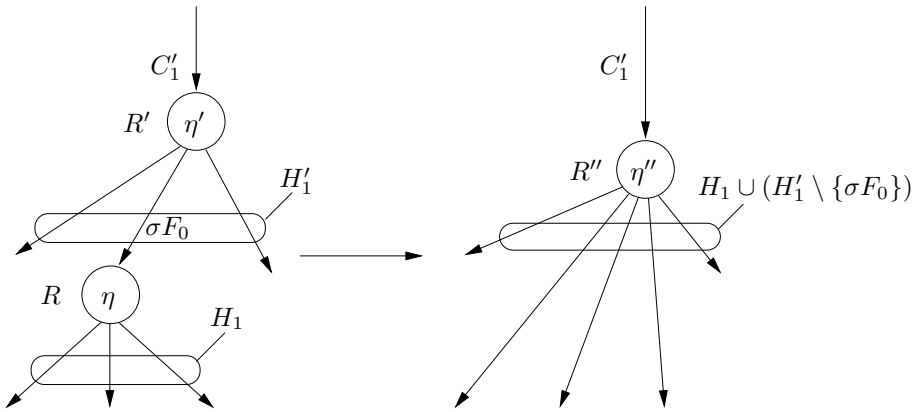

Figure 5. Merging of nodes of Lemma 3

x_i. The attacker has at least one term M, for instance $a[]$, so $\mathsf{attacker}(\sigma x_i)$ is derivable from $\mathcal{R}_0$, where $\sigma x_i = M$. Hence all elements of σH are derivable from $\mathcal{R}_0$, so from $\mathsf{saturate}(\mathcal{R}_0)$, and $\sigma C = F$. Therefore, F is derivable from $\mathcal{R}_0$. □

2.2. Proofs

In this section, we detail the proofs of Lemmas 1 and 2. We first need to prove a few preliminary lemmas. The first one shows that two nodes in a derivation can be replaced by one when combining their clauses by resolution.

Lemma 3 *Consider a derivation containing a node η', labeled R'. Let F_0 be a hypothesis of R'. Then there exists a son η of η', labeled R, such that the edge $\eta' \to \eta$ is labeled by an instance of F_0, $R \circ_{F_0} R'$ is defined, and one obtains a derivation of the same fact by replacing the nodes η and η' with a node η'' labeled $R'' = R \circ_{F_0} R'$.*

Proof:
This proof is illustrated in Figure 5. Let $R' = H' \Rightarrow C'$, H'_1 be the multiset of the labels of the outgoing edges of η', and C'_1 the label of its incoming edge. We have $R' \sqsupseteq (H'_1 \Rightarrow C'_1)$, so there exists a substitution σ such that $\sigma H' \subseteq H'_1$ and $\sigma C' = C'_1$. Since $F_0 \in H'$, $\sigma F_0 \in H'_1$, so there is an outgoing edge of η' labeled σF_0. Let η be the node at the end of this edge, let $R = H \Rightarrow C$ be the label of η. We rename the variables of R so that they are distinct from the variables of R'. Let H_1 be the multiset of the labels of the outgoing edges of η. So $R \sqsupseteq (H_1 \Rightarrow \sigma F_0)$. By the above choice of distinct variables, we can then extend σ so that $\sigma H \subseteq H_1$ and $\sigma C = \sigma F_0$.

The edge $\eta' \to \eta$ is labeled σF_0, instance of F_0. Since $\sigma C = \sigma F_0$, the facts C and F_0 are unifiable, so $R \circ_{F_0} R'$ is defined. Let σ' be the most general unifier of C and F_0, and σ'' such that $\sigma = \sigma''\sigma'$. We have $R \circ_{F_0} R' = \sigma'(H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma' C'$. Moreover, $\sigma''\sigma'(H \cup (H' \setminus \{F_0\})) \subseteq H_1 \cup (H'_1 \setminus \{\sigma F_0\})$ and $\sigma''\sigma' C' = \sigma C' = C'_1$. Hence $R'' = R \circ_{F_0} R' \sqsupseteq (H_1 \cup (H'_1 \setminus \{\sigma F_0\})) \Rightarrow C'_1$. The multiset of labels of outgoing edges of η'' is precisely $H_1 \cup (H'_1 \setminus \{\sigma F_0\})$ and the label of its incoming edge is C'_1, therefore we have obtained a correct derivation by replacing η and η' with η''. □

Lemma 4 *If a node η of a derivation D is labeled by R, then one obtains a derivation D' of the same fact as D by relabeling η with a clause R' such that $R' \sqsupseteq R$.*

Proof:
Let H be the multiset of labels of outgoing edges of the considered node η, and C be the label of its incoming edge. We have $R \sqsupseteq H \Rightarrow C$. By transitivity of $\sqsupseteq$, $R' \sqsupseteq H \Rightarrow C$. So we can relabel η with R'. □

Lemma 5 *At the end of* saturate, *$\mathcal{R}$ satisfies the following properties:*

1. *For all $R \in \mathcal{R}_0$, R is subsumed by a clause in $\mathcal{R}$;*
2. *Let $R \in \mathcal{R}$ and $R' \in \mathcal{R}$. Assume that $sel(R) = \emptyset$ and there exists $F_0 \in sel(R')$ such that $R \circ_{F_0} R'$ is defined. In this case, $R \circ_{F_0} R'$ is subsumed by a clause in $\mathcal{R}$.*

Proof:
To prove the first property, let $R \in \mathcal{R}_0$. We show that, after the addition of R to $\mathcal{R}$, R is subsumed by a clause in $\mathcal{R}$.

In the first step of saturate, we execute the instruction $\mathcal{R} \leftarrow \mathsf{elim}(\{R\} \cup \mathcal{R})$. After execution of this instruction, R is subsumed by a clause in $\mathcal{R}$.

Assume that we execute $\mathcal{R} \leftarrow \mathsf{elim}(\{R''\} \cup \mathcal{R})$ for some clause R'' and that, before this execution, R is subsumed by a clause in $\mathcal{R}$, say R'. If R' is removed by this instruction, there exists a clause R'_1 in $\mathcal{R}$ that subsumes R', so by transitivity of subsumption, R'_1 subsumes R, hence R is subsumed by the clause $R'_1 \in \mathcal{R}$ after this instruction. If R' is not removed by this instruction, then R is subsumed by the clause $R' \in \mathcal{R}$ after this instruction.

Hence, at the end of saturate, R is subsumed by a clause in $\mathcal{R}$, which proves the first property.

In order to prove the second property, we just need to notice that the fixpoint is reached at the end of saturate, so $\mathcal{R} = \mathsf{elim}(\{R \circ_{F_0} R'\} \cup \mathcal{R})$. Hence, $R \circ_{F_0} R'$ is eliminated by elim, so it is subsumed by some clause in $\mathcal{R}$. □

Proof of Lemma 1:
Assume that F is derivable from $\mathcal{R}_0$ and consider a derivation of F from $\mathcal{R}_0$. We show that F is derivable from $\mathsf{saturate}(\mathcal{R}_0)$.

We consider the value of the set of clauses $\mathcal{R}$ at the end of saturate. For each clause R in $\mathcal{R}_0$, R is subsumed by a clause in $\mathcal{R}$ (Lemma 5, Property 1). So, by Lemma 4, we can replace all clauses R in the considered derivation with a clause in $\mathcal{R}$. Therefore, we obtain a derivation D of F from $\mathcal{R}$.

Next, we build a derivation of F from $\mathcal{R}_1$, where $\mathcal{R}_1 = \mathsf{saturate}(\mathcal{R}_0)$. If D contains a node labeled by a clause not in $\mathcal{R}_1$, we can transform D as follows. Let η' be a lowest node of D labeled by a clause not in $\mathcal{R}_1$. So all sons of η' are labeled by elements of $\mathcal{R}_1$. Let R' be the clause labeling η'. Since $R' \notin \mathcal{R}_1$, $sel(R') \neq \emptyset$. Take $F_0 \in sel(R')$. By Lemma 3, there exists a son of η of η' labeled by R, such that $R \circ_{F_0} R'$ is defined, and we can replace η and η' with a node η'' labeled by $R \circ_{F_0} R'$. Since all sons of η' are labeled by elements of $\mathcal{R}_1$, $R \in \mathcal{R}_1$. Hence $sel(R) = \emptyset$. So, by Lemma 5, Property 2, $R \circ_{F_0} R'$ is subsumed by a clause R'' in $\mathcal{R}$. By Lemma 4, we can relabel η'' with R''. The total number of nodes strictly decreases since η and η' are replaced with a single node η''.

So we obtain a derivation D' of F from $\mathcal{R}$, such that the total number of nodes strictly decreases. Hence, this replacement process terminates. Upon termination, all clauses are in $\mathcal{R}_1$. So we obtain a derivation of F from $\mathcal{R}_1$, which is the expected result.

For the converse implication, notice that, if a fact is derivable from $\mathcal{R}_1$, then it is derivable from $\mathcal{R}$, and that all clauses added to $\mathcal{R}$ do not create new derivable facts: if a fact is derivable by applying the clause $R \circ_{F_0} R'$, then it is also derivable by applying R and R'. □

Proof of Lemma 2:
Let us prove the direct implication. We show that, if F' is derivable from $\mathcal{R}_1$, then there exist a clause $H \Rightarrow C$ in $\mathsf{derivable}(F, \mathcal{R}_1)$ and a substitution σ such that $\sigma C = F'$ and all elements of σH are derivable from $\mathcal{R}_1$.

Let $\mathcal{D}$ be the set of derivations D' of F' such that, for some $\mathcal{R}$, the clause R' at the subroot of D' satisfies $\mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{derivable}(F, \mathcal{R}_1)$ and $\forall R'' \in \mathcal{R}, R'' \not\sqsupseteq R'$, and the other clauses of D' are in $\mathcal{R}_1$.

Let D_0 be a derivation of F' from $\mathcal{R}_1$. Let D'_0 be obtained from D_0 by adding a node labeled by $R' = F \Rightarrow F$ at the subroot of D_0. By definition of $\mathsf{derivable}$, $\mathsf{deriv}(R', \emptyset, \mathcal{R}_1) \subseteq \mathsf{derivable}(F, \mathcal{R}_1)$, and $\forall R'' \in \emptyset, R'' \not\sqsupseteq R'$. Hence D'_0 is a derivation of F' in $\mathcal{D}$, so $\mathcal{D}$ is non-empty.

Now, consider a derivation D_1 in $\mathcal{D}$ with the smallest number of nodes. The clause R' labeling the subroot η' of D_1 satisfies $\mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{derivable}(F, \mathcal{R}_1)$, and $\forall R'' \in \mathcal{R}, R'' \not\sqsupseteq R'$. In order to obtain a contradiction, we assume that $sel(R') \neq \emptyset$. Let $F_0 \in sel(R')$. By Lemma 3, there exists a son η of η', labeled by R, such that $R \circ_{F_0} R'$ is defined and we can replace η and η' with a node η'' labeled by $R_0 = R \circ_{F_0} R'$, obtaining a derivation D_2 of F' with fewer nodes than D_1. The subroot of D_2 is the node η'' labeled by R_0.

By hypothesis on the derivation D_1, $R \in \mathcal{R}_1$, so $\mathsf{deriv}(R_0, \{R'\} \cup \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1) \subseteq \mathsf{derivable}(F, \mathcal{R}_1)$ (third case of the definition of $\mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1)$).

- If $\forall R_1 \in \{R'\} \cup \mathcal{R}, R_1 \not\sqsupseteq R_0$, D_2 is a derivation of F' in $\mathcal{D}$, with fewer nodes than D_1, which is a contradiction.
- Otherwise, $\exists R_1 \in \{R'\} \cup \mathcal{R}, R_1 \sqsupseteq R_0$. Therefore, by Lemma 4, we can build a derivation D_3 by relabeling η'' with R_1 in D_2. There is an older call to deriv, of the form $\mathsf{deriv}(R_1, \mathcal{R}', \mathcal{R}_1)$, such that $\mathsf{deriv}(R_1, \mathcal{R}', \mathcal{R}_1) \subseteq \mathsf{derivable}(F, \mathcal{R}_1)$. Moreover, R_1 has been added to $\mathcal{R}'$ in this call, since R_1 appears in $\{R'\} \cup \mathcal{R}$. Therefore the third case of the definition of $\mathsf{deriv}(R_1, \mathcal{R}', \mathcal{R}_1)$ has been applied, and not the first case. So $\forall R_2 \in \mathcal{R}', R_2 \not\sqsupseteq R_1$, so the derivation D_3 is in $\mathcal{D}$ and has fewer nodes than D_1, which is a contradiction.

In all cases, we could find a derivation in $\mathcal{D}$ that has fewer nodes than D_1. This is a contradiction, so $sel(R') = \emptyset$, hence $\mathsf{deriv}(R', \mathcal{R}, \mathcal{R}_1) = \{R'\}$ (second case of the definition of deriv), so $R' \in \mathsf{derivable}(F, \mathcal{R}_1)$. The other clauses of this derivation are in $\mathcal{R}_1$. By definition of a derivation, $R' \sqsupseteq H' \Rightarrow F$ where H' is the multiset of labels of the outgoing edges of the subroot of the derivation. Taking $R' = H \Rightarrow C$, there exists σ such that $\sigma C = F$ and $\sigma H \subseteq H'$, so all elements of σH are derivable from $\mathcal{R}_1$.

The proof of the converse implication is left to the reader. (Basically, if a fact is derivable by applying the clause $R \circ_{F_0} R'$, then it is also derivable by applying R and R'.) □

2.3. Optimizations

The resolution algorithm uses several optimizations, in order to speed up resolution. The first two are standard, while the last three are specific to protocols.

Elimination of duplicate hypotheses If a clause contains several times the same hypotheses, the duplicate hypotheses are removed, so that at most one occurrence of each hypothesis remains.

Elimination of tautologies If a clause has a conclusion that is already in the hypotheses, this clause is a tautology: it does not derive new facts. Such clauses are removed.

Elimination of hypotheses $\mathsf{attacker}(x)$ If a clause $H \Rightarrow C$ contains in its hypotheses $\mathsf{attacker}(x)$, where x is a variable that does not appear elsewhere in the clause, then the hypothesis $\mathsf{attacker}(x)$ is removed. Indeed, the attacker always has at least one message, so $\mathsf{attacker}(x)$ is always satisfied for some value of x.

Decomposition of data constructors A data constructor is a constructor f of arity n that comes with associated destructors g_i for $i \in \{1, \ldots, n\}$ defined by $g_i(f(x_1, \ldots, x_n)) \rightarrow x_i$. Data constructors are typically used for representing data structures. Tuples are examples of data constructors. For each data constructor f, the following clauses are generated:

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

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

Therefore, $\mathsf{attacker}(f(p_1, \ldots, p_n))$ is derivable if and only if $\forall i \in \{1, \ldots, n\}$, $\mathsf{attacker}(p_i)$ is derivable. When a fact of the form $\mathsf{attacker}(f(p_1, \ldots, p_n))$ is met, it is replaced with $\mathsf{attacker}(p_1) \wedge \ldots \wedge \mathsf{attacker}(p_n)$. If this replacement is done in the conclusion of a clause $H \Rightarrow \mathsf{attacker}(f(p_1, \ldots, p_n))$, n clauses are created: $H \Rightarrow \mathsf{attacker}(p_i)$ for each $i \in \{1, \ldots, n\}$. This replacement is of course done recursively: if p_i itself is a data constructor application, it is replaced again. The clauses (Rf) and (Rg) for data constructors are left unchanged. (When $\mathsf{attacker}(x)$ cannot be selected, the clauses (Rf) and (Rg) for data constructors are in fact not necessary, because they generate only tautologies during resolution. However, when $\mathsf{attacker}(x)$ can be selected, which cannot be excluded with certain extensions, these clauses may become necessary for soundness.)

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 clauses which have this fact in their hypotheses. At the end of the computation, the tool checks that the fact is indeed underivable from the obtained clauses. 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 clauses. This speeds up the resolution algorithm. In most cases, the secret keys of the principals cannot be known by the attacker. So, examples of underivable facts are $\mathsf{attacker}(sk_A[])$, $\mathsf{attacker}(sk_B[])$, ...

For simplicity, the proofs given in Section 2.2 do not take into account these optimizations. For a full proof, we refer the reader to [30, Appendix C].

2.4. Termination

In general, the resolution algorithm may not terminate. (The derivability problem is undecidable.) In practice, however, it terminates in most examples.

We have shown with Podelski that it always terminates on a large and interesting class of protocols, the *tagged protocols* [35]. We consider protocols that use as cryptographic primitives only public-key encryption and signatures with atomic keys, shared-key encryption, message authentication codes, and hash functions. Basically, a protocol is tagged when each application of a cryptographic primitive is marked with a distinct constant tag. It is easy to transform a protocol into a tagged protocol by adding tags. For instance, our example of protocol can be transformed into a tagged protocol, by adding the tags c_0, c_1, c_2 to distinguish the encryptions and signature:

$$\begin{array}{ll} \text{Message 1.} & A \to B : \left\{\!\left|\langle c_1, [\langle c_0, k\rangle]_{sk_A}\rangle\right|\!\right\}^{\mathsf{a}}_{pk_B} \\ \text{Message 2.} & B \to A : \{\!|\langle c_2, s\rangle|\!\}^{\mathsf{s}}_{k} \end{array}$$

Adding tags preserves the expected behavior of the protocol, that is, the attack-free executions are unchanged. In the presence of attacks, the tagged protocol may be more secure. Hence, tagging is a feature of good protocol design, as explained e.g. in [9]: the tags are checked when the messages are received; they facilitate the decoding of the received messages and prevent confusions between messages. More formally, tagging prevents type-flaw attacks [66], which occur when a message is taken for another message. However, the tagged protocol is potentially more secure than its untagged version, so, in other words, a proof of security for the tagged protocol does not imply the security of its untagged version.

Other authors have proved related results: Ramanujan and Suresh [85] have shown that secrecy is decidable for tagged protocols. However, their tagging scheme is stronger since it forbids blind copies. A blind copy happens when a protocol participant sends back part of a message he received without looking at what is contained inside this part. On the other hand, they obtain a decidability result, while we obtain a termination result for an algorithm which is sound, efficient in practice, but approximate. Arapinis and Duflot [13] extend this result but still forbid blind copies. Comon-Lundh and Cortier [47] show that an algorithm using ordered binary resolution, ordered factorization and splitting terminates on protocols that blindly copy at most one term in each message. In contrast, our result puts no limit on the number of blind copies, but requires tagging.

For protocols that are not tagged, we have also designed some heuristics to adapt the selection function in order to obtain termination more often. We refer the reader to [32, Section 8.2] for more details.

It is also possible to obtain termination in all cases at the cost of additional abstractions. For instance, Goubault-Larrecq shows that one can abstract the clauses into clauses in the decidable class $\mathcal{H}_1$ [63], by losing some relational information on the messages.

3. Extensions

3.1. Treatment of Equations

Up to now, we have defined cryptographic primitives by associating rewrite rules to destructors. Another way of defining primitives is by equational theories, as in the applied

pi calculus [6]. This allows us to model, for instance, variants of encryption for which the failure of decryption cannot be detected or more complex primitives such as Diffie-Hellman key agreements. The Diffie-Hellman key agreement [54] enables two principals to build a shared secret. It is used as an elementary step in more complex protocols, such as Skeme [69], SSH, SSL, and IPsec.

The Horn clause verification approach can be extended to handle some equational theories. For example, the Diffie-Hellman key agreement can be modeled by using a constant g and a function exp that satisfy the equation

$$\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) = \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x). \tag{4}$$

In practice, the function is $\mathsf{exp}(x, y) = x^y \mod p$, where p is prime and g is a generator of $\mathbb{Z}_p^*$. The equation $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) = (\mathsf{g}^x)^y \mod p = (\mathsf{g}^y)^x \mod p = \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x)$ is satisfied. In ProVerif, following the ideas used in the applied pi calculus [6], we do not consider the underlying number theory; we work abstractly with the equation (4). The Diffie-Hellman key agreement involves two principals A and B. A chooses a random name x_0, and sends $\mathsf{exp}(\mathsf{g}, x_0)$ to B. Similarly, B chooses a random name x_1, and sends $\mathsf{exp}(\mathsf{g}, x_1)$ to A. Then A computes $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x_1), x_0)$ and B computes $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x_0), x_1)$. Both values are equal by (4), and they are secret: assuming that the attacker cannot have x_0 or x_1, it can compute neither $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x_1), x_0)$ nor $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x_0), x_1)$.

In ProVerif, the equation (4) is translated into the rewrite rules

$$\mathsf{exp}(\mathsf{exp}(\mathsf{g}, x), y) \rightarrow \mathsf{exp}(\mathsf{exp}(\mathsf{g}, y), x) \quad \mathsf{exp}(x, y) \rightarrow \mathsf{exp}(x, y).$$

Notice that this definition of exp is non-deterministic: a term such as $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, a), b)$ can be reduced to $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, b), a)$ and $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, a), b)$, so that $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, a), b)$ reduces to its two forms modulo the equational theory. The rewrite rules in the definition of function symbols are applied exactly once when the function is applied. So the rewrite rule $\mathsf{exp}(x, y) \rightarrow \mathsf{exp}(x, y)$ is necessary to make sure that exp never fails, even when the first rewrite rule cannot be applied, and these rewrite rules do not loop because they are applied only once at each application of exp. More details on the treatment of equations in ProVerif and, in particular, a proof that these rewrite rules correctly model the equation (4) can be found in [33, Section 5].

This treatment of equations has the advantage that resolution can still use syntactic unification, so it remains efficient. However, it also has limitations; for example, it cannot handle associative functions, such as XOR, because it would generate an infinite number of rewrite rules for the destructors. Recently, other treatments of equations that can handle XOR and Diffie-Hellman key agreements with more detailed algebraic relations (including equations of the multiplicative group modulo p) within the Horn clause approach have been proposed by Küsters and Truderung: they handle XOR provided one of its two arguments is a constant in the clauses that model the protocol [71] and Diffie-Hellman key agreements provided the exponents are constants in the clauses that model the protocol [72]; they proceed by transforming the initial clauses into richer clauses on which the standard resolution algorithm is applied. We refer the reader to Chapter *"Verifying a bounded number of sessions and its complexity"* for the treatment of equations for a bounded number of sessions, to [49,46] for treatments of XOR for a bounded number

of sessions, and to [76,45,65,78] for other treatments of Diffie-Hellman key agreements, using unification modulo [76,65] or for a bounded number of sessions [45,78].

3.2. Translation from the Applied Pi Calculus

ProVerif does not require the user to manually enter the Horn clauses described previously. These clauses can be generated automatically from a specification of the protocol in the applied pi calculus [6]. (Chapter *"Applied Pi Calculus"* presents cryptographic pi calculi, and the applied pi calculus in particular.) On such specifications, ProVerif can verify various security properties, by using an adequate translation into Horn clauses:

- secrecy, as described above. The translation from the applied pi calculus to Horn clauses is given in [3].
- correspondences, which are properties of the form "if an event has been executed, then other events have been executed" [32]. They can in particular be used for formalizing authentication.
- some process equivalences, which mean intuitively that the attacker cannot distinguish two processes (i.e. protocols). Process equivalences can be used for formalizing various security properties, in particular by expressing that the attacker cannot distinguish a process from its specification. ProVerif can prove particular cases of observational equivalences. It can prove strong secrecy [28], which means that the attacker cannot see when the value of the secret changes. This is a stronger notion of secrecy than the one mentioned previously. It can be used, for instance, for expressing the secrecy of values taken among a set of known constants, such as bits: one shows that the attacker cannot distinguish whether the bit is 0 or 1. More generally, ProVerif can also prove equivalences between processes that differ by the terms they contain, but have otherwise the same structure [33]. In particular, these equivalences can express that a password-based protocol is resistant to guessing attacks: even if the attacker guesses the password, it cannot verify that its guess is correct.

As for secrecy, when no derivation from the clauses is found, the desired security property is proved. When a derivation is found, there may be attack. ProVerif then tries to reconstruct a trace in the applied pi calculus semantics that corresponds to this derivation [11]. (Trace reconstruction may fail, in particular when the derivation corresponds to a false attack; in this case, one does not know whether there is an attack or not.)

4. Application to Examples of Protocols

The automatic protocol verifier ProVerif is available at `http://www.proverif.ens.fr/`. It was successfully applied to many protocols of the literature, to prove secrecy and authentication properties: flawed and corrected versions of the Needham-Schroeder public-key [81,73] and shared-key [81,42,82], Woo-Lam public-key [87,88] and shared-key [87,12,9,88,61], Denning-Sacco [53,9], Yahalom [42], Otway-Rees [83, 9,84], and Skeme [69] protocols. No false attack occurred in these tests and the only non-termination cases were some flawed versions of the Woo-Lam shared-key protocol. The other protocols were verified in less than one second each on a Pentium M 1.8 GHz [30].

ProVerif was also used for proving strong secrecy in the corrected version of the Needham-Schroeder public-key protocol [73] and in the Otway-Rees [83], Yahalom [42], and Skeme [69] protocols, the resistance to guessing attacks for the password-based protocols EKE [18] and Augmented EKE [19], and authentication in the Wide-Mouth-Frog protocol [8] (version with one session). The runtime went from less than one second to 15 s on these tests, on a Pentium M 1.8 GHz [28,33].

Moreover, ProVerif was also used in more substantial case studies:

- With Abadi [4], we applied it to the verification of a certified email protocol [7]. We use correspondence properties to prove that the receiver receives the message if and only if the sender has a receipt for the message. (We use simple manual arguments to take into account that the reception of sent messages is guaranteed.) One of the tested versions includes the SSH transport layer in order to establish a secure channel. (Total runtime: 6 min on a Pentium M 1.8 GHz.)
- With Abadi and Fournet [5], we studied the JFK protocol (*Just Fast Keying*) [10], which was one of the candidates to the replacement of IKE as key exchange protocol in IPSec. We combined manual proofs and ProVerif to prove correspondences and equivalences. (Total runtime: 3 min on a Pentium M 1.8 GHz.)
- With Chaudhuri [34], we studied the secure filesystem Plutus [67] with ProVerif, which allowed us to discover and fix weaknesses of the initial system.

Other authors also use ProVerif for verifying protocols or for building other tools:

- Bhargavan et al. [26,22,20] use it to build the Web services verification tool TulaFale: Web services are protocols that send XML messages; TulaFale translates them into the input format of ProVerif and uses ProVerif to prove the desired security properties.
- Bhargavan et al. [25,23,24] use ProVerif for verifying implementations of protocols in F# (a functional language of the Microsoft .NET environment): a subset of F# large enough for expressing security protocols is translated into the input format of ProVerif. The TLS protocol, in particular, was studied using this technique [21].
- Canetti and Herzog [43] use ProVerif for verifying protocols in the computational model: they show that, for a restricted class of protocols that use only public-key encryption, a proof in the Dolev-Yao model implies security in the computational model, in the universal composability framework. Authentication is verified using correspondences, while secrecy of keys corresponds to strong secrecy.
- ProVerif was also used for verifying a certified email web service [74], a certified mailing-list protocol [68], e-voting protocols [70,16], the ad-hoc routing protocol ARAN (*Authenticated Routing for Adhoc Networks*) [60], and zero-knowledge protocols [17].

Finally, Goubault-Larrecq and Parrennes [64] also use the Horn clause method for analyzing implementations of protocols written in C. However, they translate protocols into clauses of the $\mathcal{H}_1$ class and use the $\mathcal{H}_1$ prover by Goubault-Larrecq [63] rather than ProVerif to prove secrecy properties of the protocol.

5. Conclusion

A strong aspect of the Horn clause approach is that it can prove security properties of protocols for an unbounded number of sessions, in a fully automatic way. This is essential for the certification of protocols. It also supports a wide variety of security primitives and can prove a wide variety of security properties.

On the other hand, the verification problem is undecidable for an unbounded number of sessions, so the approach is not complete: it does not always terminate and it performs approximations, so there exist secure protocols that it cannot prove, even if it is very precise and efficient in practice.

Acknowledgments This work owes much to discussions with Martín Abadi. I am very grateful to him for what he taught me. We thank Mark Ryan and Ben Smyth for comments on a draft of this chapter. This work was partly done at Bell Labs Research, Lucent Technologies, Palo Alto and at Max-Planck-Institut für Informatik, Saarbrücken. This chapter borrows material from [27,31,32].

References

[1] M. Abadi. Security protocols and their properties. In *Foundations of Secure Computation*, NATO Science Series, pages 39–60. IOS Press, 2000.

[2] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, volume 2030 of *LNCS*, pages 25–41. Springer, Apr. 2001.

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

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

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

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

[7] M. Abadi, N. Glew, B. Horne, and B. Pinkas. Certified email with a light on-line trusted third party: Design and implementation. In *11th International World Wide Web Conference*, pages 387–395. ACM Press, May 2002.

[8] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.

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

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

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

[12] R. Anderson and R. Needham. Programming Satan's computer. In *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 426–440. Springer, 1995.

[13] M. Arapinis and M. Duflot. Bounding messages for free in security protocols. In *27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, volume 4855 of *LNCS*, pages 376–387. Springer, Dec. 2007.

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

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

[16] M. Backes, C. Hritcu, and M. Maffei. Automated verification of electronic voting protocols in the applied pi-calculus. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 195–209. IEEE Computer Society, June 2008.

[17] M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *29th IEEE Symposium on Security and Privacy*, pages 202–215. IEEE, May 2008.

[18] S. M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, May 1992.

[19] S. M. Bellovin and M. Merritt. Augmented Encrypted Key Exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 244–250, Nov. 1993.

[20] K. Bhargavan, R. Corin, C. Fournet, and A. Gordon. Secure sessions for web services. In *ACM Workshop on Secure Web Services (SWS'04)*, Oct. 2004.

[21] K. Bhargavan, R. Corin, C. Fournet, and E. Zălinescu. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 459–468. ACM, Oct. 2008.

[22] K. Bhargavan, C. Fournet, and A. Gordon. Verifying policy-based security for web services. In *ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277. ACM, Oct. 2004.

[23] K. Bhargavan, C. Fournet, and A. Gordon. Verified reference implementations of WS-Security protocols. In *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *LNCS*, pages 88–106. Springer, Sept. 2006.

[24] K. Bhargavan, C. Fournet, A. Gordon, and N. Swamy. Verified implementations of the information card federated identity-management protocol. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 123–135. ACM, Mar. 2008.

[25] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152. IEEE Computer Society, July 2006.

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

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

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

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

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

[31] B. Blanchet. *Vérification automatique de protocoles cryptographiques : modèle formel et modèle calculatoire*. Mémoire d'habilitation à diriger des recherches, Université Paris-Dauphine, Nov. 2008.

[32] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

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

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

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

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

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

[38] 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 *LNCS*, pages 84–98.

Springer, Sept. 1998.

[39] Y. Boichut, N. Kosmatov, and L. Vigneron. Validation of prouvé protocols using the automatic tool TA4SP. In *Proceedings of the Third Taiwanese-French Conference on Information Technology (TFIT 2006)*, pages 467–480, Mar. 2006.

[40] D. Bolignano. Towards a mechanization of cryptographic protocol verification. In *9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 131–142. Springer, 1997.

[41] L. Bozga, Y. Lakhnech, and M. Périn. Pattern-based abstraction for verifying secrecy in protocols. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(1):57–76, Feb. 2006.

[42] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[43] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *Proceedings, Theory of Cryptography Conference (TCC'06)*, volume 3876 of *LNCS*, pages 380–403. Springer, Mar. 2006.

[44] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *CONCUR 2000: Concurrency Theory*, volume 1877 of *LNCS*, pages 365–379. Springer, Aug. 2000.

[45] Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. Deciding the security of protocols with Diffie-Hellman exponentiation and products in exponents. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference*, volume 2914 of *LNCS*, pages 124–135. Springer, Dec. 2003.

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

[47] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *LNCS*, pages 148–164. Springer, June 2003.

[48] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. In *Programming Languages and Systems: 12th European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 99–113. Springer, Apr. 2003.

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

[50] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, 29-31 Jan. 1979.

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

[52] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Workshop on Formal Methods and Security Protocols*, 25 June 1998.

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

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

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

[56] N. Durgin, P. Lincoln, J. C. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

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

[58] G. Filé and R. Vigo. Expressive power of definite clauses for verifying authenticity. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 251–265. IEEE, July 2009.

[59] T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *LNCS*, pages 271–290. Springer, June 2000.

[60] J. C. Godskesen. Formal verification of the ARAN protocol using the applied pi-calculus. In *6th International IFIP WG 1.7 Workshop on Issues in the Theory of Security (WITS'06)*, pages 99–113, Mar. 2006.

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

[62] 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)*, volume 1800 of *LNCS*, pages 977–984. Springer, May 2000.

[63] J. Goubault-Larrecq. Deciding $\mathcal{H}_1$ by resolution. *Information Processing Letters*, 95(3):401–408, Aug. 2005.

[64] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 363–379. Springer, Jan. 2005.

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

[66] 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.

[67] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *2nd Conference on File and Storage Technologies (FAST'03)*, pages 29–42. Usenix, Apr. 2003.

[68] H. Khurana and H.-S. Hahm. Certified mailing lists. In *Proceedings of the ACM Symposium on Communication, Information, Computer and Communication Security (ASIACCS'06)*, pages 46–58. ACM, Mar. 2006.

[69] H. Krawczyk. SKEME: A versatile secure key exchange mechanism for Internet. In *Internet Society Symposium on Network and Distributed Systems Security*, Feb. 1996.

[70] S. Kremer and M. D. Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *LNCS*, pages 186–200. Springer, Apr. 2005.

[71] R. Küsters and T. Truderung. Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*, pages 129–138. ACM, Oct. 2008.

[72] R. Küsters and T. Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 157–171. IEEE, July 2009.

[73] 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.

[74] K. D. Lux, M. J. May, N. L. Bhattad, and C. A. Gunter. WSEmail: Secure internet messaging based on web services. In *International Conference on Web Services (ICWS'05)*, pages 75–82. IEEE Computer Society, July 2005.

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

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

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

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

[79] 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.

[80] D. Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, 2003.

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

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

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

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

[85] R. Ramanujam and S. Suresh. Tagging makes secrecy decidable with unbounded nonces as well. In

FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, volume 2914 of *LNCS*, pages 363–374. Springer, Dec. 2003.

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

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

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