Compiling and Verifying Security Protocols

Florent Jacquemard[1], Michaël Rusinowitch[1], Laurent Vigneron[2]

[1] LORIA – INRIA Lorraine
Campus Scientifique, B.P. 239
54506 Vandœuvre-lès-Nancy Cedex, France
Florent.Jacquemard@loria.fr, Michael.Rusinowitch@loria.fr
http://www.loria.fr/~jacquema, http://www.loria.fr/~rusi
[2] LORIA – Université Nancy 2
Campus Scientifique, B.P. 239
54506 Vandœuvre-lès-Nancy Cedex, France
Laurent.Vigneron@loria.fr, http://www.loria.fr/~vigneron

Abstract. We propose a direct and fully automated translation from standard security protocol descriptions to rewrite rules. This compilation defines non-ambiguous operational semantics for protocols and intruder behavior: they are rewrite systems executed by applying a variant of ac-narrowing. The rewrite rules are processed by the theorem-prover daTac. Multiple instances of a protocol can be run simultaneously as well as a model of the intruder (among several possible). The existence of flaws in the protocol is revealed by the derivation of an inconsistency. Our implementation of the compiler CASRUL, together with the prover daTac, permitted us to derive security flaws in many classical cryptographic protocols.

Introduction

Many verification methods have been applied to the analysis of some particular cryptographic protocols [22, 5, 8, 24, 34]. Recently, tools have appeared [17, 13, 9] to automatise the tedious and error-prone process of translating protocol descriptions into low-level languages that can be handled by automated verification systems. In this research stream, we propose a concise algorithm for a direct and fully automated translation of any standard description of a protocol, into rewrite rules. For analysis purposes, the description may include security requirements and malicious agent (intruder) abilities. The asset of our compilation is that it defines non-ambiguous operational semantics for protocols (and intruders): they are rewrite rules executed on initial data by applying a variant of narrowing [15].

In a second part of our work, we have processed the obtained rewrite rules by the theorem-prover daTac [33] based on first order deduction modulo associativity and commutativity axioms (AC). Multiple instances of a protocol can be run simultaneously as well as a model of the intruder (among several possible). The existence of flaws in classical protocols (from [7]) has been revealed by the derivation of an inconsistency with our tool CASRUL.

In our semantics, the protocol is modelled by a set of transition rules applied on a multiset of objects representing a global state. The global state contains both sent messages and expected ones, as well as every piece of information collected by the intruder. Counters (incremented by narrowing) are used for dynamic generation of nonces (random numbers) and therefore ensure their freshness. The expected messages are automatically generated from the standard protocol description and describes concisely the actions to be taken by an agent when receiving a message. Hence, there is no need to specify manually these actions with special constructs in the protocol description. The verification that a received message corresponds to what was expected is performed by unification between a sent message and an expected one. When there is a unifier, then a transition rule can be fired: the next message in the protocol is composed and sent, and the next expected one is built too. The message to be sent is composed from the previously received ones by simple projections, decryption, encryption and pairing operations. This is made explicit with our formalism. The information available to an intruder is also floating in the messages pool, and used for constructing faked messages, by ac-narrowing too. The intruder-specific rewrite rules are built by the compiler according to abilities of the intruder (for diverting and sending messages) given with the protocol description.

It is possible to specify several systems (in the sense of [17]) running a protocol concurrently. Our compiler generates then a corresponding initial state. Finally, the existence of a security flaw can be detected by the reachability of a specific critical state. One critical state is defined for each security property given in the protocol description by mean of a pattern independent from the protocol.

We believe that a strong advantage of our method is that it is not ad-hoc: the translation is working without user interaction for a wide class of protocols and therefore does not run the risk to be biased towards the detection of a known flaw. To our knowledge, only two systems share this advantage, namely Casper [17] and CAPSL [21]. Therefore, we shall limit our comparison to these works.

Casper is a compiler from protocol specification to process algebra (CSP). The approach is oriented towards finite-state verification by model-checking with FDR [28]. We use almost the same syntax as Casper for protocols description. However, our verification techniques, based on theorem proving methods, will handle infinite states models. This permits to relax many of the strong assumptions for bounding information (to get a finite number of states) in model checking. Especially, our counters technique based on narrowing ensures directly that all randomly generated nonces are pairwise different. This guarantees the freshness of information over sessions. Our approach is based on analysing *infinite* traces by refutational theorem-proving and it captures automatically the traces corresponding to attacks. Note that a recent interesting work by D.Basin [4] proposes a lazy mechanism for the automated analysis of infinite traces.

CAPSL [21] is a specification language for authentication protocols in the flavour of Casper's input. There exists a compiler [9] from CAPSL to an intermediate formalism CIL which may be converted to an input for automated verification tools such as Maude, PVS, NRL [20]. The rewrite rules produced by our compilation is also an intermediate language, which has the advantage to be an idiom understood by many automatic deduction systems. In our case we have a single rule for every protocol message exchange, as opposite to CIL which has two rules. For this reason, we feel that our model is closer to Dolev and Yao original model of protocols [11] than other rewrite models are.

As a back-end system, the advantage of daTac over Maude is that ac-unification is built-in. In [8] it was necessary to program an ad-hoc narrowing algorithm in Maude in order to find flaws in protocols such as Needham-Schroeder Public Key.

We should also mention the works by C. Meadows [19] who was the first to apply narrowing to protocol analysis. Her narrowing rules were however restricted to symbolic encryption equations.

The paper is organised as follows. In Section 1, we describe the syntax for specifying a protocol $\mathcal{P}$ to be analysed and to give as input to the translator. Section 2 presents the algorithm implemented in the translator to produce, given $\mathcal{P}$, a set of rewrite rules $R(\mathcal{P})$. This set defines the actions performed by users following the protocol. The intruder won't follow the rules of the protocol, but will rather use various skill to abuse other users. His behaviour is defined by a rewrite system $\mathcal{I}$ given in Section 3. The execution of $\mathcal{P}$ in presence of an intruder may be simulated by applying narrowing with the rules of $R(\mathcal{P}) \cup \mathcal{I}$ on some initial term. Therefore, this defines an operational semantics for security protocols (Section 4). In Section 5, we show how flaws of $\mathcal{P}$ can be detected by pattern matching on execution traces, and Section 6 describes the deduction techniques underlying the theorem prover daTac and some experiments performed with this system. For additional informations the interested reader may refer to http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/.

We assume that the reader is familiar with basic notions of cryptography and security protocols (public and symmetric key cryptography, hash functions) [30], and of term rewriting [10].

1 Input Syntax

We present in this section a precise syntax for the description of security protocols. It is very close to the syntax of CAPSL [21] or Casper [17] though it differs on some points – for instance, on those in Casper which concern CSP. The specification of a protocol $\mathcal{P}$ comes in seven parts (see Example 1, Figure 1). Three concern the protocol itself and the others describe an instance of the protocol (for a simulation).

1.1 Identifiers declarations

The identifiers used in the description of a protocol $\mathcal{P}$ have to be declared to belong to one of the following types: `user` (principal name), `public_key`, symmetric_key, table, function, number. The type number is an abstraction for any kind of data (numeric, text or record ...) not belonging to one of the other types (user, key etc). An identifier T of type table is a one entry array, which associates public keys to users names ($T[D]$ is a public key of D). Therefore, public keys may be declared alone or by mean of an association table. An identifier F of type function is a one-way (hash) function. This means that one cannot retrieve X from the digest $F(X)$.

The unary postfix function symbol $\_^{-1}$ is used to represent the private key associated to some public key. For instance, in Figure 1, $T[D]^{-1}$ is the private key of D.

Among users, we shall distinguish an intruder I (it is not declared). It has been shown by G. Lowe [18] that it is equivalent to consider an arbitrary number of intruders which may communicate and one single intruder.

1.2 Messages

The core of the protocol description is a list of lines specifying the rules for sending messages,

$$(i.\ S_i \rightarrow R_i\ :\ M_i)_{1 \leq i \leq n}$$

For each $i \leq n$, the components i (step number), S_i, R_i (users, respectively sender and receiver of the message) and M_i (message) are ground terms over a signature $\mathbb{F}$ defined as follows. The declared identifiers as well as I are nullary function symbols of $\mathbb{F}$. The symbols of $\mathbb{F}$ with arity greater than 0 are $\_^{-1}$, $\_[\_]$ (for tables access), $\_(\_)$ (for one-way functions access), $\langle \_, \_ \rangle$ (pairing), $\{\_\}_\_$ (encryption). We assume that multiple arguments in $\langle \_, \ldots, \_ \rangle$ are right associated. We use the same notation for public key and symmetric key encryption (overloaded operator). Which function is really employed shall be determined unambiguously by the type of the key.

Example 1. Throughout the paper, we illustrate our method on two toy examples of protocols inspired by [36] and presented in Figure 1. These protocols describe messages exchanges in a home cable tv set made of a decoder D and a smartcard C. C is in charge of recording and checking subscription rights to channels of the user. In the first rule of the symmetric key version, the decoder D transmits his name together with an instruction *Ins* to the smartcard C. The instruction *Ins*, summarised in a number, may be of the form "(un)subscribe to channel n" or also "check subscription right for channel n". It is encrypted using a symmetric key K known by C and D. The smartcard C executes the instruction *Ins* and if everything is fine (*e.g.* the subscription rights are paid for channel n), he acknowledges to D, with a message containing C, D and the instruction *Ins* encrypted with K. In the public key version, the privates keys of D and C respectively are used for encryption instead of K.

`protocol TV;` # `symmetric key`	`protocol TV;` # `public key`
`identifiers`	`identifiers`
C, D : `user`;	C, D : `user`;
Ins : `number`;	Ins : `number`;
K : `symmetric_key`;	T : `table`;
`messages`	`messages`
1. $D \to C : \langle D, \{Ins\}_K \rangle$	1. $D \to C : \langle D, \{Ins\}_{T[D]^{-1}} \rangle$
2. $C \to D : \langle C, D, \{Ins\}_K \rangle$	2. $C \to D : \langle C, \{Ins\}_{T[C]^{-1}} \rangle$
`knowledge`	`knowledge`
$D : C, K$;	$D : C, T, T[D]^{-1}$;
$C : K$;	$C : T, T[C]^{-1}$;
`session_instance` :	`session_instance` :
$[D : tv, C : scard, K : key]$;	$[D : tv, C : scard, T : key]$;
`intruder` : `divert, impersonate`;	`intruder` : `eaves_dropping`;
`intruder_knowledge` : $scard$;	`intruder_knowledge` : key;
`goal` : `correspondence_between` $scard, tv$;	`goal` : `secrecy_of` Ins;

Fig. 1. Cable TV toy examples

1.3 Knowledge

At the beginning of a protocol execution, each principal needs some initial knowledge to compose his messages.

The field following `knowledge` associates to each `user` a list of terms of $\mathcal{T}(\mathbb{F})$ describing all the data (names, keys, function *etc*) he knows before the protocol starts. We assume that the own name of every user is always implicitly included in his initial knowledge. The intruder's name I may also figure here. In some cases indeed, the intruder's name is known by other (naïve) principals, who shall start to communicate with him because they ignore his bad intentions.

Example 2. In Example 1, D needs the name of the smartcard C to start communication. In the symmetric key version, both C and D know the shared key K. In the public key version, they both know the `table` T. It means that whenever D knows C's name, he can retrieve and use his public key $T[C]$, and conversely. Note that the `number` Ins is not declared in D's knowledge. This value may indeed vary from one protocol execution to another, because it is created by D at the beginning of a protocol execution. The identifier Ins is therefore called a *fresh* number, or *nonce* (for oNly once), as opposite to persistent identifiers like C, D or K.

Definition 1. *Identifiers which occur in a* `knowledge` *declaration* $U : \ldots$ *(including the user name* U*) are called* persistent. *Other identifiers are called* fresh.

The subset of $\mathbb{F}$ of fresh identifiers is denoted $\mathbb{F}_{fresh}$. The identifier $ID \in \mathbb{F}_{fresh}$ is said to be *fresh in* M_i, if ID occurs in M_i and does not occur in any M_j for $j < i$. We denote $fresh(M_i)$ the list of identifiers fresh in M_i (occurring in this order). We assume that if there is a public key $K \in fresh(M_i)$ then K^{-1} also occurs in $fresh(M_i)$ (right after K). Fresh identifiers are indeed instantiated by a principal in every protocol session, for use in this session only, and disappear at the end of the session. This is typically the case of nonces. Moreover we assume that the same fresh value cannot be created in two different executions of a protocol. Symmetric keys may either be persistent or fresh.

1.4 Session instances

This field proposes some possible values to be assigned to the persistent identifiers (*e.g.* tv for D in Figure 1) and thus describes the different systems (in the sense of Casper [17]) for running the protocol. The different sessions can take place concurrently or sequentially an arbitrary number of times.

Example 3. In Figure 1, the field `session_instance` contains only one trivial declaration, where one value is assigned to each identifier. This means that we want a simulation where only one system is running the protocol (*i.e.* the number of concurrent sessions is one, and the number of sequential sessions is unbounded).

1.5 Intruder

The `intruder` field describes which strategies the intruder can use, among passive `eaves_dropping`, `divert`, `impersonate`. These strategies are described in Section 3. A blank line here means that we want a simulation of the protocol without intruder.

1.6 Intruder knowledge

The `intruder_knowledge` is a set of values introduced in `session_instance`, but not a set of identifiers (like `knowledge` of others principals).

1.7 Goal

This is the kind of flaw we want to detect. There are two families of goals, `correspondence_between` and `secrecy_of` (see Sections 5.4 and 5.3). The secrecy is related to one identifier which must be given in the declaration, and the correspondence is related to two users.

2 Protocol rules

We shall give a formal description of the possible executions of a given protocol in the formalism of normalised ac-narrowing. More precisely, we give an algorithm which translates a protocol description $\mathcal{P}$ in the above syntax into a set of rewrite rules $R(\mathcal{P})$.

We assume given a protocol $\mathcal{P}$, described by all the fields defined in Section 1, such that

$$R_i = S_{i+1} \text{ for } i = 0 \ldots n-1$$

This hypothesis is not restrictive since we can add empty messages. For instance, we can replace

$$\begin{array}{rl} i.\ A \to B : M \\ i+1.\ C \to D : M' \end{array} \quad \text{by} \quad \begin{array}{rl} i.\ A \to B : M \\ i+1.\ B \to C : \emptyset \\ i+2.\ C \to D : M' \end{array}$$

For technical convenience, we let $R_0 = S_1$ and assume that S_0, M_0 are defined and are two arbitrary new constants of $\mathbb{F}$.

As in the model of Dolev and Yao [11] the translation algorithm associates to each step $S_i \to R_i : M_i$ a rewrite rule $l_i \to r_i$. An additional rule $l_{n+1} \to r_{n+1}$ is also created. The left member l_i describes the tests performed by R_{i-1} after receiving the message M_{i-1} – R_{i-1} compares M_{i-1} (by unification) with a *pattern* describing what was expected. The right member r_i describes how $S_i = R_{i-1}$ composes and send the next message M_i, and what is the pattern of the next message expected. This representation makes explicit most of the actions performed during protocol execution (recording information, checking and composing messages), which are generally hidden in protocol description. How to build the message from the pieces has to be carefully (unambiguously) specified. The expected pattern has also to be described precisely.

Example 4. In the symmetric key version of the protocol described in Figure 1, the cipher $\{Ins\}_K$ in last field of message 2 may be composed in two ways: either directly by projection on second field of message 1, or by decryption of this projection (on second field of message 1), and re-encryption of the value *Ins* obtained, with key K. The first (shortest) case is chosen in our procedure.
The pattern expected by C for message 1 is $\langle C, x_1, \{x_2\}_K \rangle$, because C does not know D's name in advance, nor the number *Ins*. The pattern expected by D for message 2 is $\langle C, D, \{Ins\}_K \rangle$, because D wants to check that C has sent the right *Ins*.

2.1 Normalised ac-narrowing

Our operational semantics for protocols are based on narrowing [15]. To be more precise, each step of an execution of the protocol $\mathcal{P}$ is simulated by a narrowing step using $R(\mathcal{P})$. We recall that narrowing unifies the left-hand side of a rewrite rule with a target term and replaces it with the corresponding right-hand side, unlike standard rewriting which relies on *matching* left-hand sides.

Let $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denote the set of terms constructed from a (finite) set $\mathcal{F}$ of function symbols and a (countable) set $\mathcal{X}$ of variables. The set of ground terms $\mathcal{T}(\mathcal{F}, \emptyset)$ is denoted $\mathcal{T}(\mathcal{F})$. In our notations, every variable starts by the letter x. We use $u[t]_p$ to denote a term that has t as a subterm at position p. We use $u[\cdot]$ to denote the context in which t occurs in the term $u[t]_p$. By $u|_p$, we denote the *subterm* of u rooted at *position* p. A rewrite *rule* over a set of terms is an ordered pair (l, r) of terms and is written $l \to r$. A *rewrite system* $\mathcal{S}$ is a finite set of such rules. The rewrite relation $\to_{\mathcal{S}}$ can be extended to rewrite over congruence classes defined by a set of equations AC, rather than terms. These constitute ac-rewrite systems. In the following the set AC will be $\{x.(y.z) = (x.y).z, x.y = y.x\}$ where $\_.\_$ is a special binary function used for representing multisets of messages. The congruence relation generated by the AC axioms will be denoted $=_{ac}$. For instance $e.h.g =_{ac} g.e.h$. A term s *ac-rewrites by* S to another term t, denoted $s \rightarrow_{\mathcal{S}} t$, if $s|_p =_{ac} l\sigma$ and $t = s[r\sigma]_p$, for some rule $l \rightarrow r$ in $\mathcal{S}$, position p in s, and substitution σ. When s cannot be rewritten by S in any way we say it is a *normal form* for S. We note $s \downarrow_{\mathcal{S}} t$, or $t = s\downarrow_{\mathcal{S}}$ if there is a finite sequence of rewritings $s \rightarrow_{\mathcal{S}} s_1 \rightarrow_{\mathcal{S}} \ldots \rightarrow_{\mathcal{S}} t$ and t is a *normal form* for $\mathcal{S}$.

In the following we shall consider two rewrite systems $\mathcal{R}$ and $\mathcal{S}$. The role of the system $\mathcal{S}$ is to keep the messages normalised (by rewriting), while $\mathcal{R}$ is used for narrowing. A term s *ac-narrows by* $\mathcal{R}, \mathcal{S}$ to another term t, denoted $s \rightsquigarrow_{\mathcal{R},\mathcal{S}} t$, if $i)$ s is a normal form for $\mathcal{S}$, and $ii)$ $s|_p\sigma =_{ac} l\sigma$ and $t = (s[r]_p)\sigma\downarrow_{\mathcal{S}}$, for some rule $l \rightarrow r$ in $\mathcal{R}$, position p in s, and substitution σ.

Example 5. Assume $\mathcal{R} = \{a(x).c(x) \rightarrow c(x)\}$ and $\mathcal{S} = \{c(x).c(x) \rightarrow 0\}$. Then $a(0).b(0).c(x) \rightsquigarrow_{\mathcal{R},\mathcal{S}} b(0).c(0)$.

2.2 Messages algebra

We shall use for the rewrite systems $\mathcal{R}$ and $\mathcal{S}$ a sorted signature $\mathcal{F}$ containing (among other symbols) all the non-nullary symbols of $\mathbb{F}$ of Section 1, and a variable set $\mathcal{X}$ which contains one variable x_t for each term $t \in \mathcal{T}(\mathbb{F})$.

Sorts. The sorts for $\mathcal{F}$ are: `user`, `intruder`, `iuser` = `user` $\cup$ `intruder`, `public_key`, `private_key`, `symmetric_key`, `table`, `function`, `number`. Additional sorts are `text`, a super-sort of all the above sorts, and `int`, `message` and `list_of`.

Signature. All the constants occurring in a declaration `session_instance` are constant symbols of $\mathcal{F}$ (with the same sort as the identifier in the declaration). The symbol I is the only constant of sort `intruder` in $\mathcal{F}$. The pairing function $\langle \_, \_ \rangle$ (profile `text` $\times$ `text` $\rightarrow$ `text`) and encryption functions $\{\_\}_\_$ (`text` $\times$ `public_key` $\rightarrow$ `text` or `text` $\times$ `private_key` $\rightarrow$ `text` or `text` $\times$ `symmetric_key` $\rightarrow$ `text`) are the same as in $\mathbb{F}$ (see Section 1.2), as well as the unary function $\_^{-1}$ (`public_key` $\rightarrow$ `private_key` or `private_key` $\rightarrow$ `public_key`) for private keys (see Section 1.1), and as the table functions $\_[\_]$ (`table` $\times$ `iuser` $\rightarrow$ `public_key`). We use a unary function symbol $nonce(\_)$: `int` $\rightarrow$ `number` for the fresh numbers, see Section 2.4. We shall use similar unary functions $K(\_)$ (`int` $\rightarrow$ `public_key`) and $SK(\_)$ (`int` $\rightarrow$ `symmetric_key`) for respectively public and symmetric fresh keys.

At last, the constant 0 (sort `int`) and unary successor function $s(\_)$ (`int` $\rightarrow$ `int`) will be used for integer (time) encoding. Some other constants $1, \ldots, k$ and $\underline{0}$, $\underline{1} \ldots$ and some alternative successor functions $s_1(\_), \ldots, s_k(\_)$ are also used. The number k is fixed according to the protocol $\mathcal{P}$ (see page 10).

From now on, x_t, x_{pu}, x_p, x_s, x_{ps}, x_u, x_{f} are variables of respective sorts `table`, `public_key`, `public_key` $\cup$ `private_key`, `symmetric_key`, `public_key` $\cup$

$\mathtt{private\_key} \cup \mathtt{symmetric\_key}$, $\mathtt{user}$, and $\mathtt{function}$. K, SK and KA will be arbitrary terms of $\mathcal{T}(\mathbb{F})$ of resp. sorts $\mathtt{public\_key} \cup \mathtt{private\_key}$, $\mathtt{symmetric\_key}$ and $\mathtt{public\_key} \cup \mathtt{private\_key} \cup \mathtt{symmetric\_key}$.

Rewrite system for normalisation. In order to specify the actions performed by the principals, $\mathcal{F}$ contains some destructors. The decryption function applies to a text encrypted with some key, in order to extract its content. It is denoted the same way as the encryption function $\{\_\}_\_$. Compound messages can be broken into parts using projections $\pi_1(\_), \pi_2(\_)$. Hence the relations it introduces in the message algebra are:

$$\left\{\{x\}_{x_s}\right\}_{x_s} \rightarrow x \tag{1}$$

$$\left\{\{x\}_{x_{pu}}\right\}_{x_{pu}^{-1}} \rightarrow x \tag{2}$$

$$\left\{\{x\}_{x_{pu}^{-1}}\right\}_{x_{pu}} \rightarrow x \tag{3}$$

$$x^{-1^{-1}} \rightarrow x \tag{4}$$

$$\pi_1(\langle x_1, x_2\rangle) \rightarrow x_1 \tag{5}$$

$$\pi_2(\langle x_1, x_2\rangle) \rightarrow x_2 \tag{6}$$

The rule (4) does not correspond to a real implementation of the generation of private key from public key. However, it is just a technical convenience. The terminating rewrite system $(1) - (6)$ is called S_0. It can be easily shown that S_o is convergent [10], hence every message t admits a unique normal form $t \downarrow_{S_0}$ for S_0.

We assume from now on that the protocol $\mathcal{P}$ is *normalised*, in the following sense.

Definition 2. *A protocol $\mathcal{P}$ is called normalised if all the message terms in the field* $\mathtt{messages}$ *are in normal form w.r.t.* $\mathcal{S}_0$.

Note that this hypothesis is not restrictive since any protocol $\mathcal{P}$ is equivalent to the normalised protocol $\mathcal{P} \downarrow_{\mathcal{S}_0}$.

2.3 Operators on messages

We define in this section some functions to be called during the construction of the system $\mathcal{R}(\mathcal{P})$ in Section 2.4.

Knowledge decomposition. We denote by $know(U, i)$ the information that a user U has memorised at the end of the step $S_i \rightarrow R_i : M_i$ of the protocol $\mathcal{P}$. This information augments incrementally with i:

- if U is the receiver R_i, then he records the received message M_i as well as the sender's (official) name S_i,
- if U is the sender S_i, then he records the fresh elements (nonces ...) he has created for composing M_i (and may use latter),

– in any other case, the knowledge of U remains unchanged.

The set $know(U, i)$ contains labelled terms $V : t \in \mathcal{T}(\mathbb{F}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$. The label t keeps track of the operations to derive V from the knowledge of U at the end of step i, using decryption and projection operators. This term t will be used later for composing new messages.

The informations are not only memorized but also decomposed with the function $CL^{(7-11)}()$ which is the closure of a set of terms using the following four rules:

$$\begin{aligned}
&\text{infer } M : \{t\}_{t'} \ \text{ from } \ \{M\}_{SK} : t \text{ and } SK : t' && (7)\\
&\text{infer } M : \{t\}_{t'} \ \text{ from } \ \{M\}_{K} : t \text{ and } K^{-1} : t' && (8)\\
&\text{infer } M : \{t\}_{t'} \ \text{ from } \ \{M\}_{K^{-1}} : t \text{ and } K : t' && (9)\\
&\text{infer } M_1 : \pi_1(t) && (10)\\
&\text{and } M_2 : \pi_2(t) \ \text{ from } \ \langle M_1, M_2\rangle : t && (11)
\end{aligned}$$

The function $know()$ is defined by:

$$\begin{aligned}
know(U, 0) &= CL^{(7-11)}\big(\{T_1 : x_{T_1}, \ldots, T_k : x_{T_k}\}\big)\\
&\qquad \text{where } \texttt{knowledge}\ U : T_1, \ldots, T_k \text{ is a statement of } \mathcal{P}.\\
know(U, i+1) &= know(U, i) \quad \text{ if } U \neq S_{i+1} \text{ and } U \neq R_{i+1}\\
know(R_{i+1}, i+1) &= CL^{(7-11)}(know(U, i) \cup \{M_{i+1} : x_{M_{i+1}}, S_{i+1} : x_{S_{i+1}}\})\\
know(S_{i+1}, i+1) &= CL^{(7-11)}(know(U, i) \cup \{N_1 : x_{N_1}, \ldots, N_k : x_{N_k}\})\\
&\qquad \text{where } N_1, \ldots, N_k = fresh(M_{i+1})
\end{aligned}$$

Example 6. In the symmetric-key version of the Cable TV example (Figure 1), we have $Ins : \{\pi_2(x_M)\}_K \in know(C, 1)$ where M is the first message and x_M gets instantiated during the execution of a protocol instance.

Message composition. We define now an operator $compose(U, M, i)$ which returns a receipt of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ for the user U for building M from the knowledge gained at the end of step i (hence, U's knowledge at the begining of step $i+1$). In that way, we formalise the basic operations performed by a sender when he composes the pieces of the message M_{i+1}. In rule (16) below, we assume that M is the k^{th} nonce created in the message M_{i+1}.

$$\begin{aligned}
compose(U, M, i) &= t \quad \text{ if } M : t \in know(U, i) && (12)\\
compose\big(U, \langle M_1, M_2\rangle, i\big) &= \big\langle compose(U, M_1, i), compose(U, M_2, i)\big\rangle && (13)\\
compose\big(U, \{M\}_{KA}, i\big) &= \big\{compose(U, M, i)\big\}_{compose(U, KA, i)} && (14)\\
compose\big(U, T[A], i\big) &= compose(U, T, i)[compose(U, A, i)] && (15)\\
compose\big(U, M, i\big) &= nonce(s_k(x_{\text{time}})) && (16)\\
compose(U, M, i) &= \textbf{Fail} \quad \text{ in every other case} && (17)
\end{aligned}$$

The cases of the $compose()$ definition are tried in the given order. Other orders are possible, and more studies are necessary to evaluate their influence on the behaviour of our system.

The construction in case (16) is similar when M is a fresh public key or a fresh symmetric key, with respective terms $K(s_k(x_{\text{time}}))$, and $SK(s_k(x_{\text{time}}))$.

Expected patterns. The term of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ returned by the following variant of $\mathit{compose}(U, M, i)$ is a filter used to check received messages by pattern matching. More precisely, the function $\mathit{expect}(U, M, i)$ defined below is called right after the message M_{i+1} has been sent by U (hence with $U = S_{i+1} = R_i$).

$$\mathit{expect}(U, M, i) = t \quad \text{if } M : t \in \mathit{know}(U, i) \tag{18}$$
$$\mathit{expect}\big(U, \langle M_1, M_2\rangle, i\big) = \big\langle \mathit{expect}(U, M_1, i), \mathit{expect}(U, M_2, i)\big\rangle \tag{19}$$
$$\mathit{expect}\big(U, \{M\}_K, i\big) = \big\{\mathit{expect}(U, M, i)\big\}_{\mathit{compose}(U, K^{-1}, i)^{-1}} \tag{20}$$
$$\mathit{expect}\big(U, \{M\}_{K^{-1}}, i\big) = \big\{\mathit{expect}(U, M, i)\big\}_{\mathit{compose}(U, K, i)^{-1}} \tag{21}$$
$$\mathit{expect}\big(U, \{M\}_{SK}, i\big) = \big\{\mathit{expect}(U, M, i)\big\}_{\mathit{compose}(U, SK, i)} \tag{22}$$
$$\mathit{expect}\big(U, T[A], i\big) = \mathit{expect}(U, T, i)[\mathit{expect}(U, A, i)] \tag{23}$$
$$\mathit{expect}(U, M, i) = x_{U,M,i} \quad \text{in every other case} \tag{24}$$

Note that unless $\mathit{compose}()$, the $\mathit{expect}()$ function cannot fail. If the call to $\mathit{compose}()$ fails in one of the cases (20)–(22), then the case (24) will be applied.

Example 7. The pattern expected by C for message 1 (Figure 1, symmetric key version) is $\mathit{expect}(C, \big\langle D, \{\mathit{Ins}\}_K\big\rangle, 1) = \langle x_{C,D,1}, \{x_{C,\mathit{Ins},1}\}_{x_K}\rangle$ because C does not know D's name in advance, nor the number Ins, but he knows K.

2.4 Narrowing rules for standard messages exchanges

The global state associated to a step of a protocol instance will be defined as the set of messages $m_1.m_2.\ldots.m_n$ sent and not yet read, union the set of expected messages $w_1.\ldots.w_m$.

A sent message is denoted by $m(i, s', s, r, t, c)$ where i is the protocol step when it is sent, s' is the real sender, s is the official sender, r is the receiver, t is the body of the message and c is a session counter (incremented at the end of each session).

$$m : \texttt{step} \times \texttt{iuser} \times \texttt{iuser} \times \texttt{iuser} \times \texttt{text} \times \texttt{int} \to \texttt{message}$$

Note that s and s' may differ since messages can be impersonated (the receiver r never knows the identity of the real sender s').

A message expected by a principal is signalled by a term $w(i, s, r, t, \ell)$ with similar meaning for the fields i, s, r, t, and c, and where ℓ is a list containing r's knowledge just before step i.

$$w : \texttt{step} \times \texttt{iuser} \times \texttt{user} \times \texttt{text} \times \texttt{list\_of text} \times \texttt{int} \to \texttt{message}$$

Nonces and freshness. We describe now a mechanism for the construction of fresh terms, in particular of nonces. This is an important aspect of our method. Indeed, it ensures freshness of the randomly generated nonces or keys over several executions of a protocol. The idea is the following: nonces admit as argument a counter that is incremented at each transition (this argument is therefore the *age* of the nonce). Hence if two nonces are emitted at different steps in an execution trace, their counters do not match. We introduce another term in the global state for representing the counter, with the new unary head symbol h. Each rewrite rule $l \to r$ is extended to $h(s(x_{\text{time}})).l \to h(x_{\text{time}}).r$ in order to update the counter. Note that the variable x_{time} occurs in the argument of $nonce()$ in case (16) of the definition of $compose()$.

Rules. The rules set $R(\mathcal{P})$ generated by our algorithm contains (for $i = 0..n$):

$$\boxed{\begin{array}{l} h(s(x_{\text{time}})). \\ w\big(i, x_{S_i}, x_{R_i}, x_{M_i}, \ell know(R_i, i), xc\big). \\ m(i, x_r, x_{S_i}, x_{R_i}, x_{M_i}, c) \to \\ \quad h(x_{\text{time}}). \\ \quad m\big(i+1, x_{R_i}, x_{R_i}, compose(R_i, R_{i+1}, i), compose(R_i, M_{i+1}, i), c\big). \\ \quad w\big(k_i, compose(R_i, S_{k_i}, i), x_{R_i}, expect(R_i, M_{k_i}, i'), \ell know(R_i, i'), c'\big) \end{array}}$$

where k_i is the next step when R_i expects a message (see definition below), and $\ell know(R_i, i)$, $\ell know(R_i, i')$ are lists of variables described below.

If $i = 0$, the term $m(i, \dots)$ is missing in left member, and $c = xc$.
If $1 \leq i \leq n$, then $c = xc'$ (another variable).
If $i = n$, the term $m(i, \dots)$ is missing in right member.
In every case ($0 \leq i \leq n$),
 if $k_i > i$ then $i' = i + 1$ and $c' = xc$,
 if $k_i \leq i$ then $i' = 0$ and $c' = s(xc)$.

Note that the calls of $compose()$ may return **Fail**. In this case, the construction of $R(\mathcal{P})$ stops with failure.

After receiving message i (of content x_{M_i}) from x_r (apparently from x_{S_i}), x_{R_i} checks whether he received what he was expecting (by unification of the two instances of x_{M_i}), and then composes and sends message $i+1$. The term returned by $compose(R_i, M_{i+1}, i)$ contains some variables in the list $\ell know(R_i, i)$. As soon as he is sending the message $i+1$, x_{R_i} gets into a state where he is waiting for new messages. This will be expressed by deleting the term $w(i, \dots)$ (previously expected message) and generating the term $w(k_i, \dots)$ in the right-hand side (next expected message). Hence sending and receiving messages is not synchronous (see e.g. [5]).

The function $\ell know(U, i)$ associates to a user U and a (step) number $i \in \{0..n\}$ a term corresponding to a list of variables, used to refer to the knowledge of U. Below, $\ell :: a$ denotes the appending of the element a at the end of a list ℓ.

$$
\begin{aligned}
\ell know(U,0) &= \langle x_U, x_{T_1}, \ldots, x_{T_n} \rangle \\
&\qquad \text{where } \texttt{knowledge}\ U : T_1, \ldots, T_n \text{ is a statement of } \mathcal{P}, \\
\ell know(U,i+1) &= \ell know(U,i) \text{ if } U \neq R_i \\
&= \ell know(U,i) :: x_{M_i} :: x_{S_i} :: n_1 :: \ldots :: n_k \text{ if } U = R_i \\
&\qquad \text{where } \mathit{fresh}(M_i) = N_1, \ldots, N_k \\
&\qquad \text{and } n_i = x_{N_i} \text{ if } N_i \text{ is of sort } \texttt{nonce} \text{ or } \texttt{symmetric\_key}, \\
&\qquad \text{and } n_i = x_{N_i} :: x_{N_i^{-1}} \text{ if } N_i \text{ is of sort } \texttt{public\_key},
\end{aligned}
$$

The algorithm also uses the integer k_i which is the next session step when R_i expects a message. If R_i is not supposed to receive another message in the current session then either he is the session initiator S_1 and k_i is reinitialized to 0, otherwise k_i is the first step in the next session where he should receive a message (and then $k_i < i$). Formally, k_i is defined for $i = 0$ to n as follows:

$k_i = \min\{j \mid j > i \text{ and } R_j = R_i\}$ if this set is not empty;
otherwise $k_i = \min\{j \mid j \leq i \text{ and } R_j = R_i\}$ (recall that $R_0 = S_1$ by hypothesis);

Example 8. In both protocols presented in Figure 1, one has $R_0 = D$, $R_1 = C$, $R_2 = D$, and therefore: $k_0 = 2$, $k_1 = 1$, $k_2 = 0$.

Lemma 1. *k is a bijection from $\{0, \ldots, n\}$ to $\{0, \ldots, n\}$.*

Example 9. The translator generates the following $R(\mathcal{P})$ for the symmetric key version of the protocol of Figure 1. For sake of readability, in this example and the following ones, the fresh variables are denoted x_i (where i is an integer) instead of the form of the case (24) in the definition of *expect*().

$$
\begin{aligned}
&h(s(x_{\text{time}})).w\big(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_K\rangle, xc\big) \rightarrow \\
&\quad h(x_{\text{time}}).m\big(1, x_D, x_D, x_C, \langle x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_K}, xc\big). \\
&\qquad\qquad w\big(2, x_C, x_D, \langle x_C, x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_K}\rangle, \\
&\qquad\qquad\quad \langle x_D, x_C, x_K, x_{M_0}, x_{S_0}, nonce(s_1(x_{\text{time}}))\rangle, xc\big) \qquad (\text{tvs}_1)
\end{aligned}
$$

$$
\begin{aligned}
&h(s(x_{\text{time}})).w\big(1, x_D, x_C, x_{M_1}, \langle x_C, x_K\rangle, xc\big). \\
&\qquad\qquad m\big(1, x_r, x_D, x_C, x_{M_1}, xc'\big) \rightarrow \\
&\quad h(x_{\text{time}}).m\big(2, x_C, x_C, \pi_1(x_{M_1}), \langle x_C, \pi_1(x_{M_1}), \pi_2(x_{M_1})\rangle, xc'\big). \\
&\qquad\qquad w\big(1, x_D, x_C, \langle x_D, \{x_1\}_{x_K}\rangle, \langle x_C, x_K\rangle, s(xc)\big) \qquad (\text{tvs}_2)
\end{aligned}
$$

$$
\begin{aligned}
&h(s(x_{\text{time}})).w\big(2, x_C, x_D, x_{M_2}, \langle x_D, x_C, x_K, x_{M_0}, x_{S_0}, x_{Ins}\rangle, xc\big). \\
&\qquad\qquad m\big(2, x_r, x_C, x_D, x_{M_2}, xc'\big) \rightarrow \\
&\quad h(x_{\text{time}}).w\big(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_K\rangle, s(xc)\big) \qquad (\text{tvs}_3)
\end{aligned}
$$

3 Intruder rules

The main difference between the behaviour of a honest principal and the intruder I is that the latter is not forced to follow the protocol, but can send messages

arbitrarily. Therefore, there will be no $w()$ terms for I. In order to build messages, the intruder stores some information in the global state with terms of the form $i()$, where i is a new unary function symbol. The rewriting rules corresponding to the various intruder's techniques are detailed below.
The intruder can record the information aimed at him, (25). If `divert` is selected in the field `intruder`, the message is removed from the current state (26), but not if `eaves_dropping` is selected (27).

$$m(x_i, x_u, x_u, I, x, xc) \to i(x).i(x_u) \tag{25}$$
$$m(x_i, x_u, x_u, x'_u, x, xc) \to i(x).i(x_u).i(x'_u) \tag{26}$$
$$m(x_i, x_u, x_u, x'_u, x, xc) \to m(x_i, x_u, x_u, x'_u, x, xc).i(x).i(x_u).i(x'_u) \tag{27}$$

After collecting information, I can decompose it into smaller $i()$ terms. Note that the information which is decomposed (*e.g.* $\langle x_1, x_2\rangle$) is not lost during the operation.

$$i\big(\langle x_1, x_2\rangle\big) \to i\big(\langle x_1, x_2\rangle\big).i(x_1).i(x_2) \tag{28}$$
$$i\big(\{x_1\}_{x_p}\big).i\big(x_p^{-1}\big) \to i\big(\{x_1\}_{x_p}\big).i\big(x_p^{-1}\big).i(x_1) \tag{29}$$
$$i\big(\{x_1\}_{x_s}\big).i\big(x_s\big) \to i\big(\{x_1\}_{x_s}\big).i\big(x_s\big).i(x_1) \tag{30}$$
$$i\big(\{x_1\}_{x_p^{-1}}\big).i\big(x_p\big) \to i\big(\{x_1\}_{x_p^{-1}}\big).i\big(x_p\big).i(x_1) \tag{31}$$

I is then able to reconstruct terms as he wishes.

$$i(x_1).i(x_2) \to i(x_1).i(x_2).i\big(\langle x_1, x_2\rangle\big) \tag{32}$$
$$i(x_1).i(x_{ps}) \to i(x_1).i(x_{ps}).i\big(\{x_1\}_{x_{ps}}\big) \tag{33}$$
$$i(x_{\mathrm{f}}).i(x) \to i(x_{\mathrm{f}}).i(x).i\big(x_{\mathrm{f}}(x)\big) \tag{34}$$
$$i(x_t).i(x_u) \to i(x_t).i(x_u).i\big(x_t[x_u]\big) \tag{35}$$

I can send arbitrary messages in his own name,

$$i(x).i(x_u) \to i(x).i(x_u).m(j, I, I, x_u, x, \underline{0}) \quad j \leq n \tag{36}$$

If moreover `impersonate` is selected, then I can fake others identity in sent messages.

$$i(x).i(x_u).i(x'_u) \to i(x).i(x_u).i(x'_u).m(j, I, x_u, x'_u, x, \underline{0}) \quad j \leq n \tag{37}$$

Note that the above intruder rules are independent from the protocol $\mathcal{P}$ in consideration. The rewrite system of the intruder (25)–(37) is denoted $\mathcal{I}$.

4 Operational semantics

4.1 Initial state

After the definition of rules of $R(\mathcal{P})$ and $\mathcal{I}$, the presentation of an operational "state/transition" semantics of protocol executions is completed here by the

definition of an initial state $t_{init}(\mathcal{P})$. This state is a term of the form $w(\dots)$ containing the patterns of the first messages expected by the principals, and their initial knowledge, for every session instance.
We add to the initial state term a set of initial knowledge for the intruder I. More precisely, we let $t_{init}(\mathcal{P}) := t_{init}(\mathcal{P}).i(v_1)\dots i(v_n)$ if the field `intruder_knowledge` : $v_1, \dots, v_n$; is declared in $\mathcal{P}$.

Example 10. The initial state for the protocol of Figure 1 (symmetric key version) is: $t_{init}(\mathcal{P}) := h(x_{\text{time}}).w(0, x_1, tv, x_2, \langle tv, scard, key\rangle, \underline{1})$
$.w(1, x_3, scard, \langle x_3, \{x_4\}_{key}\rangle, \langle scard, key\rangle, \underline{1}).i(scard)$

4.2 Protocol executions

Definition 3. *Given a ground term t_0 and rewrite systems R, S the set of executions $EXEC(t_0, R, S)$ is the set of maximal derivations $t_0 \rightsquigarrow_{R,S} t_1 \rightsquigarrow_{R,S} \dots$*

Maximality is understood w.r.t. the prefix ordering on sequences. The *normal executions* of protocol $\mathcal{P}$ are the elements of the set

$$EXEC_n(\mathcal{P}) := EXEC\big(t_{\text{init}}(\mathcal{P}), R(\mathcal{P}), \mathcal{S}_0\big)$$

Executions in the presence of an intruder are the ones in

$$EXEC_i(\mathcal{P}) := EXEC\big(t_{\text{init}}(\mathcal{P}), R(\mathcal{P}) \cup \mathcal{I}, S_0\big)$$

4.3 Executability

The following Theorem 1 states that if the construction of $R(\mathcal{P})$ does not fail, then normal executions will not fail (the protocol can always run and restart without deadlock).

Theorem 1. *If $\mathcal{P}$ is normalised, the field* `session_instance` *of $\mathcal{P}$ contains only one declaration, and the construction of $R(\mathcal{P})$ does not fail on $\mathcal{P}$, then every derivation in $EXEC_n(\mathcal{P})$ is infinite.*

Theorem 1 is not true if the field `session_instance` of $\mathcal{P}$ contains at least two declarations, as explained in the next section. Concurrent executions may interfere and enter a deadlock state.

4.4 Approximations for intruder rules

Due to the intruder rules of Section 3 the search space is too large. In particular, the application of rules (32)–(33) is obviously non-terminating. In our experiences, we have used restricted intruder rules for message generation.

Intruder rules guided by expected messages. The first idea is to change rules (36)–(37) so that I sends a faked message $m(i, I, x_u, x'_u, x)$ only if there exists a term of the form $w(i, x_u, x'_u, x, x_\ell, xc)$ in the global state. More precisely, we replace (36), (37) in $\mathcal{I}$ by, respectively,

$$i(x).i(x_u).w(j, I, x_u, x, x_\ell, xc) \rightarrow$$
$$i(x).i(x_u).w(j, I, x_u, x, x_\ell, xc).m(j, I, I, x_u, x, \underline{0}) \quad \text{where } j \leq n \tag{36$'$}$$

$$i(x).i(x_u).i(x'_u).w(j, x_u, x'_u, x, x_\ell, xc) \rightarrow$$
$$i(x).i(x_u).i(x'_u).w(j, x_u, x'_u, x, x_\ell, xc).m(j, I, x_u, x'_u, x, \underline{0}) \text{ where } j \leq n \tag{37$'$}$$

The obtained rewrite system is called $\mathcal{I}_w$.

This approximation is complete: every attack in $EXEC_i(\mathcal{P})$ exists also in the trace generated by the modified system, indeed, the messages in a trace of $EXEC_i(\mathcal{P})$ and not in $EXEC(t_{\text{init}}(\mathcal{P}), R(\mathcal{P}) \cup \mathcal{I}_w, S_0)$ would be rejected by the receiver as non-expected or ill-formed messages. Similar observations are reported independantly in [32]. Therefore, there is no limitation for detecting attacks with this simplification (this strategy prunes only useless branches) but it is still inefficient.

Rules guided approximation. The above strategy is improved by deleting rules (32)–(35) and replacing each rules of (36'), (37') new rules (several for each protocol message), such that a sent message has the form $m(i, I, x_u, x'_u, t, \underline{0})$, where, roughly speaking, t follows the pattern M_i where missing parts are filled with some knowledge of I. Formally, we define a non-deterministic unary operator $* : \mathcal{T}(\mathbb{F}) \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$.

$$\langle M_1, M_2 \rangle^* = \langle M_1^*, M_2^* \rangle \tag{38}$$
$$\{M\}_K^* = \{M^*\}_{K^*} \quad \big|\ x_{\{M\}_K} \tag{39}$$
$$F(M)^* = x_F(M^*) \quad \big|\ x_{F(M)} \tag{40}$$
$$T[A]^* = x_T[x_A] \quad \big|\ x_{T[A]} \tag{41}$$
$$ID^* = x_{ID} \qquad \text{if } ID \text{ is a nullary function symbol of } \mathbb{F} \tag{42}$$

Given $T \in \mathcal{T}(\mathbb{F})$ we denote $\mathit{skel}(T)$ the set of possible terms for T^*. Then, we replace (36$'$), (37$'$) in $\mathcal{I}$ by,
for each $j \in 1..n$, for each $t \in \mathit{skel}(M_j)$, for each distinct identifier A of sort `user`, let $\{x_1, \ldots, x_m\} = \mathit{Var}(t) \cup \{x_A, x_{S_i}, x_{R_i}\}$ (no variable occurrence more than once in the sequence $x_1, \ldots, x_m$):

$$i(x_1).\ldots.i(x_m).w(i, x_{S_i}, x_{R_i}, x, x_\ell, xc) \rightarrow$$
$$i(x_1).\ldots.i(x_m).w(i, x_{S_i}, x_{R_i}, x, x_\ell, xc).m(i, I, I, x_A, t, \underline{0}) \tag{36$''$}$$

and, if `impersonate` is selected in the field `intruder` of $\mathcal{P}$, by: for each $i \in 1..n$, for each $t \in \mathit{skel}(M_i)$, for each distinct identifiers A, B of sort `user`, let $\{x_1, \ldots, x_m\} = \mathit{Var}(t) \cup \{x_A, x_B, x_{S_i}, x_{R_i}\}$:

$$i(x_1).\ldots.i(x_m).w(i, x_{S_i}, x_{R_i}, x, x_\ell, xc) \rightarrow$$
$$i(x_1).\ldots.i(x_m).w(i, x_{S_i}, x_{R_i}, x, x_\ell, xc).m(i, I, x_A, x_B, t, \underline{0}) \tag{37$''$}$$

Because of deletion of rules (32)-(35), one rule for public key decryption with tables needs to be added:

$$i(\{x_1\}_{x_t[x_u]^{-1}}).i(x_t).i(x_u) \to i(\{x_1\}_{x_t[x_u]^{-1}}).i(x_p).i(x_u).i(x_1) \qquad (43)$$

The obtained system depends on $\mathcal{P}$. Note that this approximation is not complete. However, it seems to give reasonable results in practice.

5 Flaws

In our state/transition model, a flaw will be detected when the protocol execution reaches some critical state. We define a critical state as a pattern $t_{\mathrm{goal}}(\mathcal{P}) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, which is constructed automatically from the protocol $\mathcal{P}$. The existence of a flaw is reducible to the following reachability problem, where a can be either i or n:

$$\exists t_0, \dots, t_{\mathrm{goal}}(\mathcal{P})\sigma \in EXEC_a(\mathcal{P}) \text{ for some substitution } \sigma$$

5.1 Design flaws

It may happen that the protocol fails to reach its goals even without intruder, *i.e.* only in presence of honest agents following the protocol carefully. In particular, it may be the case that there is an interference between several concurrent runs of the same protocol: confusion between a message $m(i, \dots)$ from the first run and another $m(i, \dots)$ from the second one. An example of this situation is given in Appendix A. The critical state in this case is: (recall that xc and xc' correspond to session counters)

$$t_{\mathrm{goal}}(\mathcal{P}) := w(i, x_s, x_r, x_m, x_l, xc).m(i, x_{s'}, x_s, x_r, x_m, xc').[xc \neq xc']$$

where $[xc \neq xc']$ is a constraint that can be checked either by extra rewrite rules or by an internal mechanism as in daTac.

5.2 Attacks, generalities

Following the classification of Woo and Lam [36], we consider two basic security properties for authentication protocols: secrecy and correspondence. *Secrecy* means that some secret information (*e.g.* a key) exchanged during the protocol is kept secret. *Correspondence* means that every principal was really involved in the protocol execution, *i.e.* that mutual authentication is ensured. The failure of one of these properties in presence of an intruder is called a flaw.

Example 11. The following scenario is a *correspondence attack* for the symmetric key version of the cable tv toy example in Figure 1:

$$\begin{array}{ll} 1. \quad D \to I(C) & : \langle D, \{Ins\}_K \rangle \\ 2. \; I(C) \to \quad D & : \langle C, D, \{Ins\}_K \rangle \end{array}$$

Following the traditional notation, the $I(C)$ in step 1 means that I did `divert` the first message of D to C. Note that this ability is selected in Figure 1. It may be performed in real world by interposing a computer between the decoder and the smartcard, with some serial interface and a smartcard reader. The sender $I(C)$ in the second message means that C did `impersonate` C for sending this message. Note that I is able to reconstruct the message of step 2 from the message he diverted at step 1, with a projection π_1 to obtain the name of D and projection π_2 to obtain the cipher $\{Ins\}_K$ and his initial knowledge (the name of the smartcard). Note that the smartcard C did not participate at all to this protocol execution. Such an attack may be used if the intruder wants to watch some channel x which is not registered in his smartcard. See [1] for the description of some real-world hacks on pay TV.

A *secrecy attack* can be performed on the public key version of the protocol in Figure 1. By listening to the message sent by the decoder at step 1, the intruder (with `eaves_dropping` ability) can decode the cipher $\{Ins\}_{T[D]^{-1}}$ since he knows the public key $T[D]$, and thus he will learn the secret instruction Ins. Note that there was no correspondence flaw in this scenario.

5.3 Secrecy attack

Definition 4. *We say that a principal U of $\mathcal{P}$* shares *a (secret) identifier N if there exists j and t such that $N : t \in know(U, j)$.*

In the construction of $R(\mathcal{P})$, we say that the term $t = compose(U, M, j)$ is *bound* to M.

Definition 5. *An execution $t_0, \dots \in EXEC_i(\mathcal{P})$ satisfies the secrecy property iff for each j, t_j does not contain an instance of $i(t)$ as a subterm, where t is bound to a term N declared in a field* `goal : secrecy of` *N of $\mathcal{P}$.*

To define a critical state corresponding to a secrecy violation in our semantics, we add a binary function symbol $secret(\_,\_)$ to $\mathcal{F}$, which is used to store a term t (nonce or session key) that is bound to some data N declared as secret in $\mathcal{P}$, by `secrecy_of` N. If this term t appears as an argument of $i(\_)$, and I was not supposed to share t, then it means that its secrecy has been corrupted by the intruder I.

We must formalise the condition that "I was not supposed to share t". For this purpose, we add a second argument to $secret(\_,\_)$ which is a term of $\mathcal{T}(\{s, \underline{1}, \dots, \underline{k}\})$, corresponding to the the value of a session counter, where k is the number of fields `session_instance :` ℓ in $\mathcal{P}$. Let $C = \{\underline{1}, \dots, \underline{k}\}$. To each field `session_instance` in $\mathcal{P}$ is associated a unique constant in C by the protcedure described in Section 4.1. Let $\mathcal{J} \subseteq C$ be the set of session instances where I has not the role of a principal that shares N.

The critical state $t_{\mathrm{goal}}(\mathcal{P})$ is any of the terms of the set:

$$\{i(x).secret(x, f(\underline{c}))\}_{\underline{c} \in \mathcal{J}}$$

The auxiliary unary function symbol $f(\_)$ scrapes off the $s(\dots)$ context in the values of session counters, using the following rewrite rule (added to $\mathcal{S}_0$):

$$f(s(x)) \to f(x) \tag{44}$$

The storage in $secret(\_,\_)$ is performed in the rewrite rule for constructing the message M_{i+1} where N appears for the first time. More precisely, there is a special construction in the rewrite rule for building M_{i+1}.The binding to the secret N is a side effect of the recursive call of the form $compose(U, N, i)$. The i^{th} rule constructed by our algorithm (page 12) will be in this case:

$$\left\{\begin{array}{l} h(s(x_{\text{time}})). \\ w(i, x_{S_i}, x_{R_i}, x_{M_i}, \ell know(R_i, i), xc). \\ m(i, x_r, x_{S_i}, x_{R_i}, x_{M_i}, xc') \end{array}\right\} \to$$
$$\left\{\begin{array}{l} h(x_{\text{time}}). \\ m(i+1, x_{R_i}, x_{R_i}, compose(R_i, R_{i+1}, i), compose(R_i, M_{i+1}, i), xc'). \\ w(k_i, compose(R_i, S_{k_i}, i), x_{R_i}, expect(R_i, M_{k_i}, i'), \ell_i', c'). \\ secret(t, f(xc')) \end{array}\right\}$$

Example 12. The rules generated for the protocol of Figure 1, public key version, are:

$$\begin{array}{l} h(s(x_{\text{time}})).w\big(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_T, x_{T[D]^{-1}}\rangle, xc\big) \to \\ \quad h(x_{\text{time}}).m\big(1, x_D, x_D, x_C, \langle x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_{T[D]^{-1}}}\rangle, xc\big). \\ \qquad w\big(2, x_C, x_D, \langle x_C, x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_1}\rangle, \\ \qquad\quad \langle x_D, x_C, x_T, x_{T[D]^{-1}}, x_{M_0}, x_{S_0}, nonce(s_1(x_{\text{time}}))\rangle, xc\big) \\ \qquad secret(nonce(s_1(x_{\text{time}})), f(xc)) \end{array} \tag{tvp$_1$}$$

$$\begin{array}{l} h(s(x_{\text{time}})).w\big(1, x_D, x_C, x_{M_1}, \langle x_C, x_T, x_{T[C]^{-1}}\rangle, xc\big). \\ \qquad m\big(1, x_r, x_D, x_C, x_{M_1}, xc'\big) \to \\ \quad h(x_{\text{time}}).m\big(2, x_C, x_D, \pi_1(x_{M_1}), \langle x_C, \pi_1(x_{M_1}), \pi_2(x_{M_1})\rangle, xc'\big). \\ \qquad w\big(1, x_1, x_{U_1}, \langle x_1, \{x_2\}_{x_K}\rangle, \langle x_C, x_T, x_{T[C]^{-1}}\rangle, s(xc)\big) \end{array} \tag{tvp$_2$}$$

$$\begin{array}{l} h(s(x_{\text{time}})).w\big(2, x_C, x_D, x_{M_2}, \langle x_D, x_C, x_T, x_{T[D]^{-1}}, x_{M_0}, x_{S_0}, x_{Ins}\rangle, xc\big). \\ \qquad m\big(2, x_r, x_C, x_D, x_{M_2}, xc'\big) \to \\ \quad h(x_{\text{time}}).w\big(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_T, x_{T[D]^{-1}}\rangle, s(xc)\big) \end{array} \tag{tvp$_3$}$$

Note the term $secret(nonce(s_1(x_{\text{time}})), xc)$ in rule (tvp$_1$). As described in Example 11, it is easy to see that this protocol has a secrecy flaw. A subterm $secret(nonce(x), \underline{1}).i(nonce(x))$ is obtained in 4 steps, see appendix C.

5.4 Correspondence attack

The correspondence property between two users U and V means that when U terminates its part of a session c of the protocol (and starts next session $s(c)$), then V must have started his own part, and reciprocally. In Definition 6, we use the notation $\text{first}_S(U) = \min\{i \mid S_i = U\}$, assuming $\min(\emptyset) = 0$.

Definition 6. *An execution $t_0, \ldots \in EXEC_i(\mathcal{P})$ satisfies the correspondence property between the (distinct) users U and V iff for each j, t_j does not contain a subterm matching:*

$$w(\mathrm{first}_S(U) - 1, x_s, u, x_t, x_\ell, s(x_c)).w(\mathrm{first}_S(V) - 1, x'_s, x'_r, x'_t, x'_\ell, x_c)$$
$$\text{or } w(\mathrm{first}_S(V) - 1, x_s, v, x_t, x_\ell, s(x_c)).w(\mathrm{first}_S(U) - 1, x'_s, x'_r, x'_t, x'_\ell, x_c),$$

where $U : u$ and $V : v$ occur in the same line of the field `session_instance`.

The critical state $t_{\mathrm{goal}}(\mathcal{P})$ is therefore any of the two above terms in Definition 6. Again, these terms are independent from $\mathcal{P}$.

Example 13. A critical state for the protocol in Figure 1, symmetric key version, is: $t_{\mathrm{goal}}(\mathcal{P}) := w(0, x_1, tv, x_{M_1}, x_{l_1}, xc).w(1, x_2, scard, x_{M_2}, x_{l_2}, s(xc))$

5.5 Key compromising attack

A classical goal of cryptographic protocols is the exchange between two users A and B of new keys – symmetric or public keys. In such a scenario, A may propose to B a new shared symmetric key K or B may ask a trusted server for A's public key K, see Section 5.6 below for this particular second case. In this setting, a technique of attack for the intruder is to introduce a compromised key K': I has built some key K' and he let B think that K' is the key proposed by A or that this is A's public key for instance (see Example 14 for a key compromising attack). The compromising of K may be obtained by exploiting for instance a type flaw as described below. Such an attack is not properly speaking a secrecy attack. However, it can of course be exploited if later on B wants to exchange some secret with A using K (actually the compromised K').

Therefore, a key compromising attack is defined as a secrecy attack for an extended protocol $\mathcal{P}'$ obtained from a protocol $\mathcal{P}$ of the above category as follows:

1. declare a new `identifier` X : `number`;
2. add a rule: $n + 1.\ B \rightarrow A\ :\ \{X\}_K$ where n is the number of messages in $\mathcal{P}$ and K is the key to compromise,
3. add the declaration `goal` : `secrecy_of` X;

5.6 Binding attack

This is a particular case of key compromising attack, and therefore a particular case of secrecy attack, see Section 5.5. It can occur in protocols where the public keys are distributed by a trusted server (who knows a table K of public keys) because the principals do not know in advance the public keys of others. In some case, the intruder I can do appropriate diverting in order to let some principal learn a fake binding name – public key. For instance, I makes some principal B believe that I's public key $K[I]$ is the public key of a third party A (binding A–$K[I]$). This is what can happen with the protocol SLICE/AS, see [7].

5.7 Type flaw

This flaw occurs when a principal can accept a term of the wrong type. For instance, he may accept a pair of numbers instead of a new symmetric key, when numbers, pair of numbers and symmetric keys are assumed to have the same type. Therefore, a type flaw refers more to implementation hypotheses than to the protocol itself. Such a flaw may be the cause of one of the above attack, but its detection requires a modification of the sort system of $\mathcal{F}$. The idea it to collapse some sorts, by introducing new sorts equalities. For instance, one may have the equality $\mathtt{symmetric\_key} = \mathtt{text} = \mathtt{number}$. By definition of profiles of $\{\_\}_\_$ and $\langle \_, \_ \rangle$, ciphers and pairs are in this case `numbers`, and be accepted as `symmetric_key`.

Example 14. A known key compromising attack on Otway-Rees protocol, see [7], exploits a type flaw of this protocol. We present here the extended version of Otway-Rees, see Section 5.5.

$$
\begin{array}{l}
\mathtt{protocol\ Ottway\ Rees} \\
\mathtt{identifiers} \\
A, B, S \qquad\quad : \mathtt{user}; \\
K_{as}, K_{bs}, K_{ab} : \mathtt{symmetric\_key}; \\
M, N_a, N_b, X \ : \mathtt{number}; \\
\mathtt{messages} \\
1.\ A \to B : \langle M, A, B, \{N_a, M, A, B\}_{K_{as}} \rangle \\
2.\ B \to S \ : \langle M, A, B, \{N_a, M, A, B\}_{K_{as}}, N_b, M, A, B_{K_{bs}} \rangle \\
3.\ S \to B : \langle M, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}} \rangle \\
4.\ B \to A \ : \langle M, \{N_a, K_{ab}\}_{K_{as}} \rangle \\
5.\ A \to B : \{X\}_{Kab} \\
\mathtt{knowledge} \\
A : B, S, K_{as}; \\
B : S, K_{bs}; \\
S \ : A, B, K_{as}, K_{bs}; \\
\mathtt{session\_instance} \quad [A : a, B : b, S : s, k_{as} : k_{as}, K_{bs} : k_{bs}]; \\
\mathtt{intruder} : \mathtt{divert,\ impersonate}; \\
\mathtt{intruder\_knowledge} : ; \\
\mathtt{goal} : \mathtt{secrecy\_of}\ \ X;
\end{array}
$$

The symmetric keys K_{as} and K_{bs} are supposed to be only known by A and S, resp. B and S. The identifiers M, N_a, and N_b re nonces. The new symmetric K_{ab} is generated by the trusted server S and transmitted to B and indirectly to A, by mean of the cipher $\{N_a, K_{ab}\}_{K_{as}}$.

If the sorts `numbers`, `text`, and `symmetric_key` are assumed to collapse, then we have the following scenario:

$$
\begin{array}{ll}
1. & \quad A \to I(B) : \langle M, A, B, \{N_a, M, A, B\} K_{as} \rangle \\
4. & I(B) \to \quad A \quad : \langle M, \{N_a, M, A, B\} K_{as} \rangle \\
5. & \quad A \to I(B) : \{X\}_{\langle M, A, B \rangle}
\end{array}
$$

In rule 1, I diverts (and memorises) A's message. In next step 4, I impersonates B and makes him think that the triple $\langle M, A, B\rangle$ is the new shared symmetric key K_{ab}. We recall that $\langle \_, \_\rangle$ is right associative and thereafter $\langle N_a, M, A, B\rangle$ can be considered as identical to $\langle N_a, \langle M, A, B\rangle\rangle$

6 Verification: deduction techniques and experiments

We have implemented the construction of $R(\mathcal{P})$ in OCaml® and performed experiments using the theorem prover daTac [33] with paramodulation modulo AC. Each rule $l \to r \in R(\mathcal{P})$ is represented as an oriented equation $l = r$, the initial state is represented as a unit positive clause $P\big(t_{init}(\mathcal{P})\big)$ and the critical state as a unit negative clause $\neg P\big(t_{\text{goal}}(\mathcal{P})\big)$.
As for multiset rewriting [8], an ac-operator will take care of concurrency. On the other hand unification will take care of communication in an elegant way. The deduction system combines paramodulation steps with equational rewriting by $\mathcal{S}_0$.

6.1 Deduction techniques. Generalities

The main deduction technique consists in replacing a term by an equal one in a clause: given a clause $l = r \vee C'$ and clause $C[l']$, the clause $(C' \vee C[r])\sigma$ is deduced, where σ is a unifier of l and l', that is a mapping from variables to terms such that $l\sigma$ is equal to $l'\sigma$.

This deduction rule is called *paramodulation*. It has been introduced by Robinson and Wos [27]. Paramodulation (together with resolution and factoring) was proved refutationally complete by Brand [6] who also shown that applying a replacement in a variable position is useless.
For reducing the number of potential deduction steps, the paramodulation rule has been restricted by an ordering, to guarantee it replaces big terms by smaller ones. This notion of ordered paramodulation has been applied to the Knuth-Bendix completion procedure [16] for avoiding failure in some situations (see [14] and [2]). A lot of work has been devoted to putting more restrictions on paramodulation in order to limit combinatorial explosion [23].

In particular paramodulation is often inefficient with axioms such as associativity and commutativity since these axioms allow for many successful unifications between their subterms and subterms in other clauses. Typically word problems in finitely presented commutative semigroups cannot be decided by standard paramodulation. This gets possible by building the associativity and commutativity in the paramodulation rule using the so-called paramodulation modulo AC and rewriting modulo AC rules.

The integration of associativity and commutativity axioms within theorem-proving systems has been first investigated by Plotkin [26] and Slagle [31]. Rusinowitch and Vigneron [29] have built-in this theory in a way that is compatible with the ordered paramodulation strategy and rewriting and preserves refutational completeness. These techniques are implemented in the daTac system [33].

Another approach has been followed by Wertz [35] and Bachmair and Ganzinger [3], consisting of using the idea of extended clauses developed for the equational case by Peterson and Stickel [25].
In all the approaches, the standard unification calculus has to be replaced by unification modulo associativity and commutativity. This may be very costly since some unification problems have doubly exponentially many minimal solutions [12].

6.2 Deduction rules for protocol verification

We present here the version of paramodulation we have applied for simulating and verifying protocols. States are built with the specific ac-operator "." for representing the multiset of information components: sent and expected messages, and the knowledge of the intruder.

The definition of our instance of the paramodulation rule is the following.

Definition 7 (Paramodulation). $\dfrac{l = r \qquad\qquad P(l')}{P(r.z)\sigma}$ *if σ is an ac-unifier of $l.z$ and l', and z is a new variable.*

This rule is much simpler than the general one in [29]. We only need to apply replacements at the top of the term. In addition the equations are such that the left-hand side is greater than the right-hand side and each clause is unit. So we do not need any strategy for orienting the equations or selecting a literal in a clause.

In the verification of protocols, we encounter only simple unification problems. They reduce to unifying multisets of standard terms, where one of the multisets has no variable as argument of ".". Only one argument of the other multiset is a variable. Hence for handling these problems we have designed a unification algorithm which is more efficient than the standard ac-unification algorithm of daTac.

Let us illustrate this with an example.

Example 15. For performing a paramodulation step from $f(x_1).g(a) = c$ into $P(a.g(x_2).f(b).h(x_3))$, trying to unify $f(x_1).g(a)$ and $a.g(x_2).f(b).h(x_3)$ will not succeed. We have to add a new variable in the left-hand side of the equation for capturing the additional arguments of the ac-operator. The unification problem we have to solve is $f(x_1).g(a).z =^?_{ac} a.g(x_2).f(b).h(x_3)$. Its unique solution σ is $\{x_1 \mapsto b, x_2 \mapsto a, z \mapsto a.h(x_3)\}$. The deduced clause is $P(c.z)\sigma$, that is $P(c.a.h(x_3))$.

The paramodulation rule is used for generating new clauses. We need a rule for detecting a contradiction with the clause representing the goal.

Definition 8 (Contradiction). $\dfrac{P(t) \qquad\qquad \neg P(t')}{\Box}$ *if σ is an ac-unifier of t and t'.*

In addition to these two deduction rules, we need to simplify clauses by term rewriting, using equations of S_0 (rewrite rules (1)–(6)). For this step we have to compute a match σ of a term l into l', that is a substitution such that $l\sigma = l'$.

Definition 9 (Simplification). $\dfrac{l = r \qquad\qquad P(t[l'])}{P(t[r\sigma])}$ *if σ is a match of l into l'.*

Applying this rule consists in replacing the initial clause by the simplified one.

6.3 Deduction strategy

We basically apply a breadth first search strategy. The compilation of the protocol generates four sets of clauses:

(0) the rewrite rules of S_0;
(1) the clauses representing transitions rules (including intruder's rules);
(2) the clause representing the initial state, $P\big(t_{init}(\mathcal{P})\big)$;
(3) the critical state $(\neg P\big(t_{\mathrm{goal}}(\mathcal{P})\big))$;

The deduction strategy used by daTac is the following:

Repeat:
  Select a clause C in (2), C contains only a positive literal
  Repeat:
    Select a clause D in (1), D is an equation $l = r$
    Apply Paramodulation from D into C:
      Compute all the most general ac-unifiers
      For each solution σ,
        Generate the resulting clause $C'\sigma$
    Simplify the generated clauses:
      For each generated clause $C'\sigma$,
        Select a rewrite rule $l \rightarrow r$ in (0)
        For each subterm s in $C'\sigma$,
          If s is an instance $l\phi$ of l
          Then Replace s by $r\phi$ in $C'\sigma$
    Add the simplified generated clauses into (2)
    Try Contradiction between the critical state and each new clause:
      If it applies, Exit with message "*contradiction found*".
  Until no more clause to select in (1)
Until no more clause to select in (2)

Note that any derivation of a contradiction $\Box$ with this strategy is a linear derivation from the initial state to the goal and it can be directly interpreted as a scenario for a flaw or an attack.

6.4 Results

The approach has been experimented with several protocols described in [7]. We have been able to find the known flaws with this uniform method in several protocols, in less than 1 minute (including compilation) in every case, see Figure 2.

Protocol	Description	Flaw	Intruder abilities
Encrypted Key Exchange	Key distribution	Correspondence attack	divert impersonate
Needham Shroeder Public Key	Key distribution with authentication	Secrecy attack	divert impersonate
Otway Rees	Key distribution with trusted server	Key compromising = secrecy attack type flaw	divert impersonate
Shamir Rivest Adelman	Transmission of secret information	Secrecy attack	divert impersonate
Tatebayashi Matsuzaki Newman	Key distribution	Key compromising = secrecy attack	eaves_dropping impersonate
Woo and Lam Π	Authentication	Correspondence attack	divert impersonate

Fig. 2. Experiments

See http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/ for more details.

7 Conclusion

We have presented a complete, compliant translator from security protocols to rewrite rules and how it is used for the detection of flaws. The advantages of our system are that the automatic translation covers a large class of protocols and that the narrowing execution mechanism permits to handle several aspects like timeliness. A drawback of our approach is that the produced rewrite system can be complex and therefore flaw detection gets time-consuming. However, simplifications should be possible to shorten derivations. For instance, composition and reduction with rules $\mathcal{S}_0$ may be performed in one step.

The translation can be directly extended for handling key systems satisfying algebraic laws such as commutativity (cf. RSA). It can be extended to other kinds of flaws: binding, typing... We plan to analyse E-commerce protocols where our management of freshness should prove to be very useful since fresh data are ubiquitous in electronic forms (order and payment e.g.). We plan to develop a generic daTac proof strategy for reducing the exploration space when searching for flaws. We also conjecture it is possible to modify our approach in order to prove the absence of flaws under some assumptions.

References

1. R. Anderson. Programming Satan's computer. volume 1000 of *Lecture Notes in Computer Science*. Springer-Verlag.
2. L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without Failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 1–30. Academic Press inc., 1989.
3. L. Bachmair and H. Ganzinger. Associative-Commutative Superposition. In N. Dershowitz and N. Lindenstrauss, editors, *Proc. 4th CTRS Workshop, Jerusalem (Israel)*, volume 968 of *LNCS*, pages 1–14. Springer-Verlag, 1995.
4. D. Basin. Lazy infinite-state analysis of security protocols. In *Secure Networking — CQRE [Secure] '99*, LNCS 1740, pages 30–42. Springer-Verlag, Berlin, 1999.
5. D. Bolignano. Towards the formal verification of electronic commerce protocols. In *IEEE Computer Security Foundations Workshop*, pages 133–146. IEEE Computer Society, 1997.
6. D. Brand. Proving Theorems with the Modification Method. *SIAM J. of Computing*, 4:412–430, 1975.
7. J. Clark and J. Jacob. A survey of authentication protocol literature. `http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps`, 1997.
8. G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Formal Methods and Security Protocols*, 1998. LICS '98 Workshop.
9. G. Denker and J. Millen. Capsl intermediate language. In *Formal Methods and Security Protocols*, 1999. FLOC '99 Workshop.
10. N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. North-Holland, 1990.
11. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29:198–208, 1983. Also STAN-CS-81-854, May 1981, Stanford U.
12. E. Domenjoud. A technical note on AC-unification. the number of minimal unifiers of the equation $\alpha x_1 + \cdots + \alpha x_p \doteq_{AC} \beta y_1 + \cdots + \beta y_q$. *JAR*, 8:39–44, 1992.
13. R. Focardi and R. Gorrieri. Cvs: A compiler for the analysis of cryptographic protocols. In *12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1999.
14. J. Hsiang and M. Rusinowitch. Proving Refutational Completeness of Theorem-Proving Strategies : the Transfinite Semantic Tree Method. *JACM*, 38(3):559–587, July 1991.
15. J.-M. Hullot. Canonical forms and unification. In *5th International Conference on Automated Deduction*, volume 87, pages 318–334. Springer-Verlag, LNCS, july 1980.
16. D. E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.
17. G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

18. G. Lowe. Towards a completeness result for model checking of security protocols. In *11th IEEE Computer Security Foundations Workshop*, pages 96–105. IEEE Computer Society, 1998.
19. C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.
20. C. Meadows. The NRL protocol analyzer: an overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
21. J. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.
22. J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, pages 141–154. IEEE Computer Society, 1997.
23. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 2000.
24. L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.
25. G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *JACM*, 28:233–264, 1981.
26. G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.
27. G. A. Robinson and L. T. Wos. Paramodulation and First-Order Theorem Proving. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, 1969.
28. A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Society, 1995.
29. M. Rusinowitch and L. Vigneron. Automated Deduction with Associative-Commutative Operators. *Applicable Algebra in Engineering, Communication and Computation*, 6(1):23–56, January 1995.
30. B. Schneier. *Applied Cryptography*. John Wiley, 1996.
31. J. R. Slagle. Automated Theorem-Proving for theories with Simplifiers, Commutativity and Associativity. *JACM*, 21(4):622–642, 1974.
32. P. Syverson, C. Meadows, and I. Cervesato. Dolev-Yao is no better than Machiavelli. In *WITS'00. Workshop on Issues in the Theory of Security*, 2000.
33. L. Vigneron. Positive deduction modulo regular theories. In *Proceedings of Computer Science Logic, Paderborn (Germany)*, pages 468–485. LNCS 1092, Springer-Verlag, 1995.
34. C. Weidenbach. Towards an automatic analysis of security protocols. In *Proceedings of the 16th International Conference on Automated Deduction*, pages 378–382. LNCS 1632, Springer-Verlag, 1999.
35. U. Wertz. First-Order Theorem Proving Modulo Equations. Technical Report MPI-I-92-216, MPI Informatik, April 1992.
36. T. Woo and S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Research in Security and Privacy*, pages 178–194. IEEE Computer Society, 1993.

Appendix A: design flaws

Example 16.

$$\begin{array}{ll}
\texttt{identifiers} & M, B, C : \texttt{user};\\
& O, N \quad : \texttt{number};\\
& K_b, K_c \;\, : \texttt{public_key};\\
& hash \quad : \texttt{function};\\
\texttt{messages} & 1.\ M \rightarrow C : \{O\}_{K_c}\\
& 2.\ C \rightarrow M : \langle B, \{N\}_{K_b}, hash(N)\rangle\\
& 3.\ M \rightarrow B : \{N\}_{K_b}, hash(O)\\
& 4.\ B \rightarrow M : \Big\{hash\big(hash(N), hash(O)\big)\Big\}_{K_b^{-1}}\\
\texttt{knowledge} & C \;: B, K_b, hash;\\
& M : C, O, K_c, K_b, hash;\\
& B \;: K_b, K_b^{-1}, hash;\\
\texttt{session_instance} & [M : Merchant,\ B : Bank,\ C : Customer,\\
& \quad O : car,\ K_b : k_b,\ K_c : k_c]\\
& [M : Merchant,\ B : Bank,\ C : Customer,\\
& \quad O : peanut,\ K_b : k_b,\ K_c : k_c]
\end{array}$$

This is a flawed e-commerce protocol. While browsing an online commerce site, the customer C is offered an object O (together with an order form, price information *etc*) by merchant M. Then, C transmits M a payment form N with his bank account information and the price of O, in order for M to ask directly to C's bank B for the payment. For confidentiality reasons, M must never read the contents of N, and B must not learn O. Therefore, O is encrypted in message 1 with the public key K_c of C. Also, in message 2, N is transmitted by C to M in encrypted form with the bank's public key K_b and in the form of a digest computed with the *hash* one-way function. Then M relays the cipher $\{N\}_{K_b}$ to B together with a digest of O. The bank B makes the verification for the payment and when it is possible, gives his certificate to M in the form of a dual signature.

The problem is that in message 2, there is no occurrence of O, so there may be some interference between two executions of the protocol. Imagine that C is performing simultaneously two transactions with the same merchant M. In the two concurrent execution of the protocol, M sends 1. $M \rightarrow C :\ \{car\}_{K_c}$ and 1. $M \rightarrow C :\ \{peanut\}_{K_c}$. C will reply with two distinct corresponding payment forms (the price field will vary) 2. $C \rightarrow M :\ \langle B, \{N_{car}\}_{K_b}, hash(N_{car})\rangle$ and 2. $C \rightarrow M :\ \langle B, \{N_{peanut}\}_{K_b}, hash(N_{peanut})\rangle$. But after receiving these two messages, M may be confused about which payment form is for which offer (recall that M can not read N_{car} and N_{peanut}), and send the wrong requests to B: 3. $M \rightarrow B :\ \{N_{car}\}_{K_b}, hash(peanut)$ and 3. $M \rightarrow B :\ \{N_{peanut}\}_{K_b}, hash(car)$. If the bank refuses the payment of N_{car} but authorises the one of N_{peanut}, it will give a certificate for buying a car and paying peanuts! Fortunately for M, the check of dual signature (by M) will fail and transaction will by aborted, but there is nevertheless a serious interference flaw in this protocol, that can occur even only between two honest agents (without an intruder).

Appendix B: a correspondence attack

Trace obtained by daTac of a correspondence attack for the symmetric key TV protocol (Figure 1).

$$
\begin{aligned}
&t_{init}(\mathcal{P}) = \\
&h(x_1).w(0, x_2, tv, x_3, \langle tv, scard, key\rangle, \underline{1}) \\
&\qquad .w(1, x_4, scard, \langle x_4, \{x_5\}_{key}\rangle, \langle scard, key\rangle, \underline{1}) \\
&\qquad .i(scard) \\
&\leadsto_{(\mathrm{tvs}_1)} \\
&h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x_1)\}_{key}\rangle, \underline{1}) \\
&\qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key}\rangle, \langle tv, scard, key, x_2, nonce(x_1)\rangle, \underline{1}) \\
&\qquad .w(1, x_3, scard, \langle x_3, \{x_4\}_{key}\rangle, \langle scard, key\rangle, \underline{1}) \\
&\qquad .i(scard) \\
&\leadsto_{(26)} \\
&h(x_1).w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key}\rangle, \langle tv, scard, key, x_2, nonce(x_1)\rangle, \underline{1}) \\
&\qquad .w(1, x_3, scard, \langle x_3, \{x_4\}_{key}\rangle, \langle scard, key\rangle, \underline{1}) \\
&\qquad .i(tv).i(scard).i(\langle tv, \{nonce(x_1)\}_{key}\rangle) \\
&\leadsto_{(28)} \\
&h(x_1).w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key}\rangle, \langle tv, scard, key, x_2, nonce(x_1)\rangle, \underline{1}) \\
&\qquad .w(1, x_3, scard, \langle x_3, \{x_4\}_{key}\rangle, \langle scard, key\rangle, \underline{1}) \\
&\qquad .i(tv).i(scard).i(\{nonce(x_1)\}_{key}) \\
&\leadsto_{(37)} \\
&h(x_1).m(2, I, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key}\rangle, \underline{0}) \\
&\qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key}\rangle, \langle tv, scard, key, x_2, nonce(x_1)\rangle, \underline{1}) \\
&\qquad .w(1, x_3, scard, \langle x_3, \{x_4\}_{key}\rangle, \langle scard, key\rangle, \underline{1}) \\
&\qquad .i(scard).i(tv).i(\{nonce(x_1)\}_{key}) \\
&\leadsto_{(\mathrm{tvs}_3)} \\
&h(x_1).w(0, x_2, tv, x_3, \langle tv, scard, key\rangle, s(\underline{1})) \\
&\qquad .w(1, x_3, scard, \langle x_3, \{x_4\}_{key}\rangle, \langle scard, key\rangle, \underline{1}) \\
&\qquad .i(scard).i(tv).i(\{nonce(s(x_1))\}_{key})
\end{aligned}
$$

One subterm (of the last term) matches the pattern $t_{\mathrm{goal}}(\mathcal{P})$.

Appendix C: a secrecy attack

Trace obtained by daTac of a secrecy attack for the public key TV protocol (Figure 1).

$$
\begin{aligned}
& t_{init}(\mathcal{P}) = \\
& h(x_1).w(0, x_2, tv, x_3, \langle tv, scard, key, key[tv]^{-1}\rangle, \underline{1}) \\
& \qquad .w(1, x_4, scard, \langle x_4, x_5\rangle, \langle scard, key, key[scard]^{-1}\rangle, \underline{1}) \\
& \qquad .i(key) \\
& \rightsquigarrow_{(\mathrm{tvp}_1)} \\
& h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \underline{1}) \\
& \qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \\
& \qquad\quad \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x_1)\rangle, \underline{1}) \\
& \qquad .w(1, x_4, scard, \langle x_4, x_5\rangle, \langle scard, key, key[scard]^{-1}\rangle, \underline{1}) \\
& \qquad .secret(nonce(x_1), f(\underline{1})) \\
& \qquad .i(key) \\
& \rightsquigarrow_{(27)} \\
& h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \underline{1}) \\
& \qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \\
& \qquad\quad \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x_1)\rangle, \underline{1}) \\
& \qquad .w(1, x_4, scard, \langle x_4, x_5\rangle, \langle scard, key, key[scard]^{-1}\rangle, \underline{1}) \\
& \qquad .secret(nonce(x_1), f(\underline{1})) \\
& \qquad .i(key).i(tv).i(scard).i(\langle tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle) \\
& \rightsquigarrow_{(28)} \\
& h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \underline{1}) \\
& \qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \\
& \qquad\quad \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x_1)\rangle, \underline{1}) \\
& \qquad .w(1, x_4, scard, \langle x_4, x_5\rangle, \langle scard, key, key[scard]^{-1}\rangle, \underline{1}) \\
& \qquad .secret(nonce(x_1), f(\underline{1})) \\
& \qquad .i(key).i(tv).i(scard).i(\{nonce(x_1)\}_{key[tv]^{-1}}) \\
& \rightsquigarrow_{(35)} \\
& h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \underline{1}) \\
& \qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \\
& \qquad\quad \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x_1)\rangle, \underline{1}) \\
& \qquad .w(1, x_4, scard, \langle x_4, x_5\rangle, \langle scard, key, key[scard]^{-1}\rangle, \underline{1}) \\
& \qquad .secret(nonce(x_1), f(\underline{1})) \\
& \qquad .i(key).i(tv).i(scard).i(\{nonce(x_1)\}_{key[tv]^{-1}}).i(key[tv]) \\
& \rightsquigarrow_{(31)} \\
& h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \underline{1}) \\
& \qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x_1)\}_{key[tv]^{-1}}\rangle, \\
& \qquad\quad \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x_1)\rangle, \underline{1}) \\
& \qquad .w(1, x_4, scard, \langle x_4, x_5\rangle, \langle scard, key, key[scard]^{-1}\rangle, \underline{1}) \\
& \qquad .secret(nonce(x_1), f(\underline{1})) \\
& \qquad .i(key).i(tv).i(scard).i(\{nonce(x_1)\}_{key[tv]^{-1}}).i(key[tv]).i(nonce(x_1))
\end{aligned}
$$

The subterm $secret(nonce(x_1), f(\underline{1})).i(nonce(x_1))$ matches the pattern $t_{\mathrm{goal}}(\mathcal{P})$.