Compiling and Verifying Security Protocols

Florent Jacquemard, Michaël Rusinowitch, Laurent Vigneron

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Compiling and Verifying Security Protocols

Florent Jacquemard , Michaël Rusinowitch , Laurent Vigneron

N˚3938

May 30, 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--3938--FR+ENG

Compiling and Verifying Security Protocols

Florent Jacquemard* , Michaël Rusinowitch† , Laurent Vigneron‡

Thème 2 — Génie logiciel
et calcul symbolique
Projet PROTHEO

Rapport de recherche n ° 3938 — May 30, 2000 — 25 pages

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.

Key-words: Security Protocols, Verification, Automated Deduction, Term Rewriting, Narrowing, Resolution

(Résumé : tsvp)

* LORIA and INRIA, projet PROTHEO, 615 rue du Jardin Botanique, B.P. 101, 54602 Villers-les-Nancy Cedex, France. email: Florent.Jacquemard@loria.fr – http://www.loria.fr/∼jacquema

† LORIA and INRIA, projet PROTHEO, 615 rue du Jardin Botanique, B.P. 101, 54602 Villers-les-Nancy Cedex, France. email: Michael.Rusinowitch@loria.fr – http://www.loria.fr/∼rusi

‡ Université Nancy 2, Campus Scientifique, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France. email: Laurent.Vigneron@loria.fr – http://www.loria.fr/∼vigneron

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ

Compilation et vérification de protocoles de sécurité

Résumé : Nous proposons une traduction directe et entièrement automatisée des descriptions standard de protocoles de sécurité en règles de réécriture. Cette compilation définit une sémantique opérationnelle non-ambigüe pour les protocoles et le comportement d'intrus : les règles de réécriture sont exécutées en appliquant une variante de la surréduction modulo associativité commutativité. Les règles de réécriture sont traitées par le démonstrateur automatique DATAC. Des instances multiples d'un même protocole peuvent être exécutées simultanément ainsi qu'un modèle de l'intrus (parmi plusieurs possibles). L'existence de failles de sécurité dans le protocole est détectée par la dérivation d'une incohérence. À l'aide de notre implantation du compilateur CASRUL, et du démonstrateur DATAC, nous avons dérivé un bon nombre d'attaques sur des protocoles cryptographiques classiques.

Mots-clé : Protocoles de sécurité, vérification, déduction automatique, réécriture de termes, surréduction, résolution

Introduction

Many verification methods have been applied to the analysis of some particular cryptographic protocols [17, 3, 5, 18, 22]. Recently, tools have appeared [12, 10, 6] 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 [11].

In a second part of our work, we have processed the obtained rewrite rules by the theorem-prover daTac [21] based on first order resolution 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 [4]) 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 [12]) 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 three systems share this advantage, namely Casper [12], CVS [10] and CAPSL [16]. Therefore, we shall limit our comparison to these three works.

Casper and CVS are compilers from protocol specification to process algebra (CSP for Casper and SPA for CVS). In the case of Casper, the approach is oriented towards finite-state verification by model-checking with FDR [19]. 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. CVS and Casper are similar at the specification level. However CVS has been developed to support the so-called Non-Interference approach [9] to protocol analysis. This approach requires the tedious extra-work of analysing interference traces in order to check whether they are real flaws. Our approach is also based on analysing traces. However it captures automatically the traces corresponding to attacks. Note that a recent interesting work by D.Basin [2] proposes a lazy mechanism for the automated analysis of infinite traces.

CAPSL [16] is a specification language for authentication protocols in the flavour of Casper's input. There exists a compiler [6] from CAPSL to an intermediate formalism CIL which may be converted to an input for automated verification tools such as Maude, PVS, NRL [15]. 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 [8] than other rewrite models are.

As a back-end system, the advantage of daTac over Maude is that ac-unification is built-in. In [5] 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 [14] who was the first to apply narrowing to protocol analysis. Her narrowing rules were however restricted to symbolic encryption equations.

We focus here on the translation technique. The translation is completely formalised, and the algorithm we present here is sufficiently simple and precise to be easily implemented. The interested reader may refer to `http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/` for additional informations.

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 some experiments performed with the theorem prover daTac.

We assume that the reader is familiar with basic notions of cryptography and security protocols (public and symmetric key cryptography, hash functions) [20], and of term rewriting [7].

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 [16] or Casper [12] 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 [13] 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,

$$(i.\ S_i \to R_i\ :\ M_i)_{1 \le i \le n}$$

For each $i \le n$, 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 [23] 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 resp. D and C are used for encryption instead of K.

```
protocol TV; # symmetric key version
identifiers
 C,D  : user;
 Ins  : number;
 K    : symmetric_key;
messages
 1. D → C : ⟨D, {Ins}_K⟩
 2. C → D : ⟨C, D, {Ins}_K⟩
knowledge
 D : C, K;
 C : K;
session_instance :
 [D : tv, C : scard, K : key];
intruder : divert, impersonate;
intruder_knowledge : scard;
goal : correspondence_between scard, tv;
```

```
protocol TV; # public key version
identifiers
 C,D  : user;
 Ins  : number;
 T    : table;
messages
 1. D → C : ⟨D, {Ins}_{T[D]^-1}⟩
 2. C → D : ⟨C, {Ins}_{T[C]^-1}⟩
knowledge
 D : C, T, T[D]^-1;
 C : T, T[C]^-1;
session_instance :
 [D : tv, C : scard, T : key];
intruder : eaves_dropping;
intruder_knowledge : key;
goal : secrecy_of Ins;
```

Figure 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 one 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 and D.

Definition 3 *Identifiers which occur in a* `knowledge` *declaration 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 alternatively 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 [12]) for running the protocol. The different sessions can take place concurrently or sequentially an arbitrary number of times.

Example 4 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 parallel session 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 Users 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

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

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

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

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 [8] 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 $l_0 \to r_0$ is also created. The left member l_i describes the tests performed by $S_i = R_{i-1}$ after receiving the message M_{i-1} (S_i checks that M_{i-1} contains what was expected). The right member r_i describes how S_i composes and send the 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 5 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 Normalized ac-narrowing

Our operational semantics for protocols are based on narrowing [11]. 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 $s \to_S t$, if $s|_p =_{ac} l\sigma$ and $t = s[r\sigma]_p$, for some rule $l \to r$ 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 $s \to_{\mathcal{S}} s_1 \to_{\mathcal{S}} \ldots \to_{\mathcal{S}} t$ 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 $s \leadsto_{\mathcal{R},\mathcal{S}} t$, 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 $l \to r$ in $\mathcal{R}$, position p in s', and substitution σ.

Example 6 Assume $\mathcal{R} = \{a(x).c(x) \to c(x)\}$ and $\mathcal{S} = \{c(x).c(x) \to 0\}$. Then $a(0).b(0).c(x) \leadsto_{\mathcal{R},\mathcal{S}} b(0).c(0)$.

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 $t \in \mathcal{T}(\mathbb{F})$.

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`.

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` $\to$ `text`) and encryption functions $\{\_\}_\_$ (`text` $\times$ `public_key` $\to$ `text` or `text` $\times$ `private_key` $\to$ `text` or `text` $\times$ `symmetric_key` $\to$ `text`) are the same as in $\mathbb{F}$ (see Section 1.2), as well as the unary function $\_^{-1}$ (`public_key` $\to$ `private_key` or `private_key` $\to$ `public_key`) for private keys (see Section 1.1), and as the table functions $\_[\_]$ (`table` $\times$ `iuser` $\to$ `public_key`). We use a unary function symbol $nonce(\_)$: `int` $\to$ `number` for the fresh numbers, see Section 2.4. We shall use similar unary functions $K(\_)$ (`int` $\to$ `public_key`) and $SK(\_)$ (`int` $\to$ `symmetric_key`) for respectively public and symmetric fresh keys.

At last, the constant 0 (sort `int`) and unary successor function $s(\_)$ (`int` $\to$ `int`) will be used for integer (time) encoding. Some other constants $1, \ldots, k$ and $\underline{0}$, $\underline{1}$ ... 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 9).

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`$\cup$ `private_key`$\cup$ `symmetric_key`, `user`, and `function`. K, SK and KA will be arbitrary terms of $\mathcal{T}(\mathbb{F})$ of resp. sorts `public_key` $\cup$ `private_key`, `symmetric_key` and `public_key` $\cup$ `private_key` $\cup$ `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:

$$\{\{x\}_{x_s}\}_{x_s} \rightarrow x \quad (1)$$

$$\{\{x\}_{x_{pu}}\}_{x_{pu}^{-1}} \rightarrow x \quad (2)$$

$$\{\{x\}_{x_{pu}^{-1}}\}_{x_{pu}} \rightarrow x \quad (3)$$

$$x^{-1^{-1}} \rightarrow x \quad (4)$$

$$\pi_1(\langle x_1, x_2\rangle) \rightarrow x_1 \quad (5)$$

$$\pi_2(\langle x_1, x_2\rangle) \rightarrow x_2 \quad (6)$$

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 [7], 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 7 *A protocol $\mathcal{P}$ is called normalised if all the message terms in the field* `messages` *are in normal form w.r.t. (1), (2), (3).*

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

Knowledge decomposition. We denote by $know(U, i)$ the information that a user U has recorded after step i of the protocol. It 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 current knowledge (at i) of U, using decryption and projection operators. This term t will be used later for composing new messages.

- $know(U, 0)$ is the closure using deduction rules (7)–(9) of the set $\{T_1 : x_{T_1}, \ldots, T_k : x_{T_k}\}$ where `knowledge` U : $T_1, \ldots, T_k$ is a statement of $\mathcal{P}$.
- For each $i \geq 0$, $know(U, i+1)$ is defined recursively by:
  - if $U \neq R_i$ and $U \neq S_i$ then $know(U, i+1) = know(U, i)$,
  - if $U = R_i$ then $know(U, i+1)$ is the closure using deduction rules (7)–(9) of the set $know(U, i) \cup \{M_{i+1} : x_{M_{i+1}}, S_{i+1} : x_{S_{i+1}}\}$,
  - if $U = S_i$ then $know(U, i+1)$ is the closure using deduction rules (7)–(9) of the set $know(U, i) \cup \{N_1 : x_{N_1}, \ldots, N_k : x_{N_k}\}$, where $N_1, \ldots, N_k = fresh(M_i)$.

The deduction rules applied in the above definition are the following:

$$\text{infer } M : \{t\}_{t'} \quad \text{from} \quad \{M\}_{SK} : t \text{ and } SK : t' \quad (7)$$

$$\text{infer } M : \{t\}_{t'} \quad \text{from} \quad \{M\}_{K^{-1}} : t \text{ and } K : t' \quad (8)$$

$$\text{infer } M_1 : \pi_1(t) \text{ and } M_2 : \pi_2(t) \quad \text{from} \quad \langle M_1, M_2\rangle : t \quad (9)$$

Example 8 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.

Lemma 9 *For all U, $i < n$, $know(U, i) \subseteq know(U, i+1)$.*

Lemma 10 *For all $M : t \in know(U, i)$, we have $M\downarrow_{\mathcal{S}_0} = M$.*

Proof. The proof is by induction on t and using in the base case (t is a variable x_V) the fact that the protocol is normalised. □

In the following lemma, σ is a homomorphism extension of some mapping $\mathbb{F}_0 \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$ ($\mathbb{F}_0$ is the set of constant symbols of $\mathbb{F}$) and θ is a homomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ defined by $x_V\theta = V\sigma$ for all $V \in \mathcal{T}(\mathbb{F})$.

Lemma 11 *For all U, i, for all $V : t \in know(U, i)$, $t\theta\downarrow_{\mathcal{S}_0} = V\sigma$.*

Proof. By structural induction on t. When t is a variable, say $V : t = V : x_V$, we have $x_V\theta = V\sigma$ by definition. By Lemma 10, $V\downarrow_{\mathcal{S}_0} = V$. Assume now that t is a not a variable. For instance $t = \{t_1\}_{t_2}$ where $\{V\}_{SK} : t_1$ and $SK : t_2$ belong to $know(U, i)$ and SK is a symmetric key (the other cases are similar). By induction hypothesis, $t_1\theta\downarrow_{\mathcal{S}_0} = (\{V\}_{SK})\sigma$ and $t_2\theta\downarrow_{\mathcal{S}_0} = SK\sigma$. By homomorphism we also have $\{t_1\}_{t_2}\downarrow_{\mathcal{S}_0} = (\{(\{V\}_{SK})\sigma\}_{SK\sigma})\downarrow_{\mathcal{S}_0} = V\sigma$, since $\{V\}_{SK}$ is irreducible for $\mathcal{S}_0$, by Lemma 10. □

Message composition. We define now an operator *compose*() which returns recipes of $\mathcal{T}(\mathcal{F},\mathcal{X})$ for building messages from the knowledge of a user. In that way, we formalise the basic operations performed by a sender when he builds his message M_{i+1}. In rule (14) below, we assume that M is the k^{th} nonce created in the message M_{i+1}.

$$\begin{aligned}
compose(U,M,i) &= t \quad \text{if } M:t \in know(U,i) && (10)\\
compose\big(U,\langle M_1,M_2\rangle,i\big) &= \big\langle compose(U,M_1,i), compose(U,M_2,i)\big\rangle && (11)\\
compose\big(U,\{M\}_{KA},i\big) &= \big\{compose(U,M,i)\big\}_{compose(U,KA,i)} && (12)\\
compose\big(U,T[A],i\big) &= compose(U,T,i)[compose(U,A,i)] && (13)\\
compose\big(U,M,i\big) &= nonce(s_k(x_{\text{time}})) && (14)\\
compose(U,M,i) &= \textbf{Fail} \quad \text{in every other case} && (15)
\end{aligned}$$

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

Lemma 12 *For all U,p, for all $V \in \mathcal{T}(\mathbb{F})$ irreducible by $\mathcal{S}_0$, $compose(U,V,p)\theta\downarrow_{\mathcal{S}_0} = V\sigma$.*

The σ and θ are defined similarly as in Lemma 11.
Proof. By structural induction on V. □

Expected patterns. The term of $\mathcal{T}(\mathcal{F},\mathcal{X})$ returned by the following variant of *compose*() is a filter used to check received messages by pattern matching.

$$\begin{aligned}
expect(U,M,i) &= t \quad \text{if } M:t \in know(U,i) && (16)\\
expect\big(U,\langle M_1,M_2\rangle,i\big) &= \big\langle expect(U,M_1,i), expect(U,M_2,i)\big\rangle && (17)\\
expect\big(U,\{M\}_K,i\big) &= \big\{expect(U,M,i)\big\}_{compose(U,K^{-1},i)^{-1}} && (18)\\
expect\big(U,\{M\}_{K^{-1}},i\big) &= \big\{expect(U,M,i)\big\}_{compose(U,K,i)^{-1}} && (19)\\
expect\big(U,\{M\}_{SK},i\big) &= \big\{expect(U,M,i)\big\}_{compose(U,SK,i)} && (20)\\
expect\big(U,T[A],i\big) &= expect(U,T,i)[expect(U,A,i)] && (21)\\
expect(U,M,i) &= x_{U,M,i} \quad \text{in every other case} && (22)
\end{aligned}$$

Note that unless *compose*(), the *expect*() function can not fail. If the call to *compose*() in one of the cases (18)–(20), then the case (22) will be applied.

Example 13 The pattern expected by C for message 1 (Figure 1, symmetric key version) is

$$expect(C,\langle D,\{Ins\}_K\rangle,1) = \langle x_{C,D,1},\{x_{C,Ins,1}\}_{x_K}\rangle$$

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).

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

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.

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

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 (14) of the definition of *compose*().

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

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

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.

In every case $(0 \leq i \leq n)$,

if $k_i > i$ then $i' = i$, $i'' = i + 1$, and $c' = xc$,

if $k_i \leq i$ then $i' = i'' = 0$ and $c' = s(xc)$.

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.

Note that the calls of *compose*() may return **Fail**. In this case, the construction of $R(\mathcal{P})$ stops with failure. The presentation *in extenso* of the algorithm construction for the construction of $R(\mathcal{P})$ is given in Figure 2.

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. [3]).

The function $\ell know(U, i)$ associates to a user U and a step number i a term corresponding to a list of variables, describing a subset of $know(U, i-1)$.

$\ell know(U, 0) = \langle x_U, x_{T_1}, \dots, x_{T_n} \rangle$, where `knowledge` $U : T_1, \dots, T_n$ is a statement of $\mathcal{P}$,
$\ell know(U, i+1) =$ if $U \neq R_i$ and $U \neq S_i$ then $\ell know(U, i)$
if $U = R_i$ then $\ell know(U, i) :: x_{M_i} :: x_{S_i}$
if $U = S_i$ then $\ell know(U, i) :: x_{N_1} :: \dots :: x_{N_k}$
where $N_1, \dots, N_k = fresh(M_i)$
and $\ell :: a$ denotes the appending of a at the end of a list ℓ.

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 14 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 15 *k is a bijection from $\{0,\ldots,n\}$ to $\{0,\ldots,n\}$.*

Proof. It is sufficient to show that k is injective. Assume that $k_i = k_{i'}$ with $i < i'$. If $A_i = \{j | j > i \text{ and } R_j = R_i\}$ is not empty, let k_i be its minimum. We have $R_{k_i} = R_i$. If $A_{i'}$ is not empty then $R_{k'_i} = R_{i'}$. In this case $R_i = R_{i'}$ and the situation $i < i' < k_i (= k_{i'})$ is now impossible by the choice of k_i. The case $i < k_i (= k_{i'}) \leq i'$ is also excluded since $R_i = R_{k_{i'}}$ and $i < k_{i'}$. If A_i is empty we have again $R_{k_i} = R_i = R_{k'_i} = R_{i'}$. But there is a contradiction between the fact that A_i is empty and $R_i = R_{i'}$ for an $i' > i$. □

add the rule `[init]`

$$\left\{\begin{array}{l} h(s(x_{\text{time}})). \\ w(0, x_{S_0}, x_{R_0}, x_{M_0}, \ell know(R_0, 0), xc) \end{array}\right\}$$
$$\longrightarrow \left\{\begin{array}{l} h(x_{\text{time}}). \\ m(1, x_{R_0}, x_{R_0}, compose(R_0, R_1, 0), compose(R_0, R_1, 0), xc). \\ w(k_0, compose(R_0, S_{k_0}, 0), x_{R_0}, expect(R_0, M_{k_0}, 0), \ell know(R_0, 1), c') \end{array}\right\}$$

where if $k_0 > 0$ then $c' = xc$
and if $k_0 = 0$ then $c' = s(xc)$

for $i = 1$ to $n-1$ do add the rule `[step]`

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

where if $k_i > i$ then $i' = i$, $i'' = i+1$, $c' = xc$
and if $k_i \leq i$ then $i' = i'' = 0$, $c' = s(xc)$

done;

add the rule `[end]`

$$\left\{\begin{array}{l} h(s(x_{\text{time}})). \\ w(n, x_{S_n}, x_{R_n}, x_{M_n}, \ell know(R_n, n), xc). \\ m(n, x_r, x_{S_n}, x_{R_n}, x_{M_n}, xc') \end{array}\right\}$$
$$\longrightarrow \left\{\begin{array}{l} h(x_{\text{time}}). \\ w(k_n, compose(R_n, S_{k_n}, n), x_{R_n}, expect(R_n, M_{k_n}, n), \ell know(R_n, 0), s(xc)) \end{array}\right\}$$

stop with success.

Figure 2: Rules generation algorithm

Example 16 The translator generates the following $R(\mathcal{P})$ for the symmetric key version of the protocol of Figure 1.

$$\begin{array}{ll} h(s(x_{\text{time}})).w(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_K\rangle, xc) & (\text{tvs}_1) \\ \rightarrow h(x_{\text{time}}).m(1, x_D, x_D, x_C, \langle x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_K}\rangle, xc) & \\ \quad .w(2, x_C, x_D, \langle x_C, x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_K}\rangle, \langle x_D, x_C, x_K, x_{M_0}, x_{S_0}, nonce(s_1(x_{\text{time}}))\rangle, xc) & \\ h(s(x_{\text{time}})).w(1, x_D, x_C, x_{M_1}, \langle x_C, x_K\rangle, xc) & (\text{tvs}_2) \\ \quad .m(1, x_r, x_D, x_C, x_{M_1}, xc') & \\ \rightarrow h(x_{\text{time}}).m(2, x_C, x_C, \pi_1(x_{M_1}), \langle x_C, \pi_1(x_{M_1}), \pi_2(x_{M_1})\rangle, xc') & \\ \quad .w(1, x_D, x_C, \langle x_D, \{x_1\}_{x_K}\rangle, \langle x_C, x_K\rangle, s(xc)) & \\ h(s(x_{\text{time}})).w(2, x_C, x_D, x_{M_2}, \langle x_D, x_C, x_K, x_{M_0}, x_{S_0}, x_{Ins}\rangle, xc) & (\text{tvs}_3) \\ \quad .m(2, x_r, x_C, x_D, x_{M_2}, xc') & \\ \rightarrow h(x_{\text{time}}).w(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_K\rangle, s(xc)) & \end{array}$$

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, (23). If `divert` is selected in the field `intruder`, the message is removed from the current state (24), but not if `eaves_dropping` is selected (25).

$$m(x_i, x_u, x_u, I, x, xc) \rightarrow i(x).i(x_u) \quad (23)$$
$$m(x_i, x_u, x_u, x'_u, x, xc) \rightarrow i(x).i(x_u).i(x'_u) \quad (24)$$
$$m(x_i, x_u, x_u, x'_u, x, xc) \rightarrow m(x_i, x_u, x_u, x'_u, x, xc).i(x).i(x_u).i(x'_u) \quad (25)$$

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.

$$i(\langle x_1, x_2 \rangle) \rightarrow i(\langle x_1, x_2 \rangle).i(x_1).i(x_2) \quad (26)$$
$$i(\{x_1\}_{x_p}).i(x_p^{-1}) \rightarrow i(\{x_1\}_{x_p}).i(x_p^{-1}).i(x_1) \quad (27)$$
$$i(\{x_1\}_{x_s}).i(x_s) \rightarrow i(\{x_1\}_{x_s}).i(x_s).i(x_1) \quad (28)$$
$$i(\{x_1\}_{x_p^{-1}}).i(x_p) \rightarrow i(\{x_1\}_{x_p^{-1}}).i(x_p).i(x_1) \quad (29)$$

I is then able to reconstruct terms as he wishes.

$$i(x_1).i(x_2) \rightarrow i(x_1).i(x_2).i(\langle x_1, x_2 \rangle) \quad (30)$$
$$i(x_1).i(x_{ps}) \rightarrow i(x_1).i(x_{ps}).i(\{x_1\}_{x_{ps}}) \quad (31)$$
$$i(x_{\mathrm{f}}).i(x) \rightarrow i(x_{\mathrm{f}}).i(x).i(x_{\mathrm{f}}(x)) \quad (32)$$
$$i(x_t).i(x_u) \rightarrow i(x_t).i(x_u).i(x_t[x_u]) \quad (33)$$

I can send arbitrary messages in his own name,

$$i(x).i(x_u) \rightarrow i(x).i(x_u).m(j, I, I, x_u, x, \underline{0}) \quad j \leq n \quad (34)$$

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

$$i(x).i(x_u).i(x'_u) \rightarrow i(x).i(x_u).i(x'_u).m(j, I, x_u, x'_u, x, \underline{0}) \quad j \leq n \quad (35)$$

Note that the above intruder rules are independent from the protocol $\mathcal{P}$ in consideration. The rewrite system of the intruder (23)–(35) 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 containing the patterns ($w()$-terms) of the first messages expected by the principals, and their initial knowledge, for every session instance. We shall use the following notations in the definition of $t_{init}(\mathcal{P})$. Let us admit $\min(\emptyset) = 0$, and let us define for a user U,

$$\mathrm{first}_S(U) = \min\{i \mid S_i = U\}, \quad \mathrm{first}_R(U) = \min\{i \mid R_i = U\}$$

Note that $\min\{i \mid R_i = U\} = \mathrm{first}_S(U) - 1$ since $R_0 = S_1$ and by the alternating message hypothesis. The term $t_{init}(\mathcal{P})$ is constructed according to the possible session instances, declared in the field `session_instance` of $\mathcal{P}$, as described by the algorithm given in Figure 3.
We add to the initial state term a set of initial knowledge for the intruder I. More precisely, we let $t_{init}(\mathcal{P}) := t_{init}(\mathcal{P}).i(v_1) \ldots i(v_n)$ if the field `intruder_knowledge`: $v_1, \ldots, v_n$; is declared in $\mathcal{P}$.

Example 17 The initial state for protocol of Figure 1 (symmetric key version) is:

$$t_{init}(\mathcal{P}) := h(x_{\mathrm{time}}).w(0, x_1, tv, x_2, \langle tv, scard, key \rangle, \underline{1}) \\ .w(1, x_3, scard, \langle x_3, \{x_4\}_{key} \rangle, \langle scard, key \rangle, \underline{1}).i(scard)$$

let $t_{init}(\mathcal{P}) := h(x_{\text{time}})$; let $c = 1$;
for each field `session_instance` : ℓ in $\mathcal{P}$ do
  for each user U do
    let $\vec{v} := \langle v_1, \ldots, v_k \rangle$, with `knowledge` $U : ID_1, \ldots, ID_k$ in $\mathcal{P}$, and $ID_j : v_j$ in ℓ for each $j \leq k$
    let $t_{init}(\mathcal{P}) := t_{init}(\mathcal{P}).w(\text{first}_S(U) - 1, v_s, v_r, e, \vec{v}, \underline{c})$
    where if $U = S_1$, then $e = x_{M_1}$ and $U : v_r$ in ℓ, $v_s = v_r$,
          else $e = expect(U, M_{\text{first}_S(U)-1}, 0)$, and $v_s = expect(U, \text{first}_S(U) - 1, 0)$, $U : v_r \in \ell$ ($\text{first}_S(U) > 1$ in this case)
  c := c+1;

Figure 3: Generation of the initial state t_{init}

4.2 Protocol executions

Definition 18 *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 fair derivations $t_0 \rightsquigarrow_{R,S} t_1 \rightsquigarrow_{R,S} \ldots$*

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 19 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 19 *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 19 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.

Lemma 20 *Each rule of $R(\mathcal{P})$ of the form*

$$h\big(s(x_{\text{time}})\big).w\big(i, x_{S_i}, x_{R_i}, x_{M_i}, \ell know(R_i, i), xc\big).m(i, \ldots) \rightarrow h(x_{\text{time}}).m\big(i+1, x_{R_i}, x_{R_i}, r_{i+1}, c_{i+1}, xc'\big).w(\ldots)$$

is such that $Var(r_{i+1}) \cup Var(c_{i+1}) \subseteq Var(\ell know(R_i, i)) \cup \{x_{M_i}, x_{\text{time}}, x_{S_i}\}$.

Proof. We can show that for all $V \in \mathcal{T}(\mathbb{F})$, there exists $x_V \in \mathcal{X}$ such that $V : x_V \in know(R_i, i)$ iff $x_V \in \ell know(R_i, i)$. Therefore, $x_V \notin Var(\ell know(R_i, i)) \cup \{x_{M_i}, x_{\text{time}}, x_{S_i}\}$ implies that $V : x_V \notin know(R_i, i)$. This makes a contradiction with $x_V \in Var(r_{i+1}) \cup Var(c_{i+1})$ by definition of *compose*() (and *know*()) – recall that, the algorithm in Figure 2, for the generation of $R(\mathcal{P})$, $r_{i+1} = compose(R_i, R_{i+1}, i)$, $c_{i+1} = compose(R_i, M_{i+1}, i)$. □

Proof. (Theorem 19) Let us define the homomorphisms $\sigma_i : \mathbb{F} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$, for all $1 \leq i \leq n$ by:

- for all $ID \in \mathbb{F} \setminus \mathbb{F}_{fresh}$, $ID\sigma_1 = \ldots = ID\sigma_n =$ the name assigned to ID in the (unique) line in field `session_instance` of $\mathcal{P}$
- for all $N \in \mathbb{F}_{fresh}$, if N is the k^{th} nonce created in M_i, then
  - $N\sigma_i = nonce(s_k(x_{\text{time}}))$,
  - $N\sigma_{i+j} = nonce(s_k(\underbrace{s(\ldots s(}_{j} x_{\text{time}}))))$ for all $j \leq n - i$,
  - $N\sigma_j$ is undefined for all $j < i$.

We also define an homomorphism $\theta_i : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$, for all $1 \leq i \leq n$, by

$$x_V\theta_i = V\sigma_i \text{ for all } V \in \mathcal{T}(\mathbb{F}), \qquad x_{\text{time}}\theta_i = x_{\text{time}}$$

We show that there exists a narrowing sequence

$$t_{init}(\mathcal{P}) = h(x_{\text{time}}).t_0 \rightsquigarrow_{\rho_1} h(x_{\text{time}}).t_1 \rightsquigarrow_{\rho_2} \dots h(x_{\text{time}}).t_{n-1} \rightsquigarrow_{\rho_n} h(x_{\text{time}}).t'_0$$

where ρ_i is the i^{th} rewrite rule of $R(\mathcal{P})$ created by running algorithm 2 on $\mathcal{P}$, and t'_0 is obtained from t_0 by replacing each symbol $\underline{1}$ (session counter) with $s(\underline{1})$. We show first that, denoting $\trianglerighteq$ the superterm relation, we have for each $0 < i < n$,

$$t_i \trianglerighteq u_i := w(i, S_i\sigma_i, R_i\sigma_i, w_i, \ell know(R_i, i)\theta_i, \underline{1}).m(i, S_i\sigma_i, S_i\sigma_i, R_i\sigma_i, M_i\sigma_i, \underline{1}) \quad (\star)$$

and w_i unifies with $M_i\sigma_i$.

By construction of $t_{init}(\mathcal{P})$ and ρ_1 (case `[init]` in Figure 2), $(\star)$ is true for $i = 1$.

We assume that $(\star)$ is true for all $j \in \{1, \dots, i\}$, with $1 \leq i < n-1$, and show that then $(\star)$ is true for $i+1$.

The $m()$ subterm. We show now that t_{i+1} contains a subterm:

$$m(i+1, S_{i+1}\sigma_{i+1}, S_{i+1}\sigma_{i+1}, R_{i+1}\sigma_{i+1}, M_{i+1}\sigma_{i+1}, \underline{1})$$

By hypothesis, the rule ρ_{i+1} may be applied to t_i. The rule ρ_{i+1} has indeed the following shape by construction (Figure 2, case [`step`]),

$$\left\{ \begin{array}{l} h\big(s(x_{\text{time}})\big).w(i, x_{S_i}, x_{R_i}, x_{M_i}, \ell know(R_i, i), xc). \\ m(i, x_r, x_{S_i}, x_{R_i}, x_{M_i}, xc') \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} h(x_{\text{time}}).m(i+1, x_{R_i}, x_{R_i}, r_{i+1}, c_{i+1}, xc). \\ w(k_{i+1}, d_{i+1}, x_{R_i}, e_{i+1}, \ell know(R_i, i''), c) \end{array} \right\}$$

Let τ_i be the mgu between the left-hand side of ρ_{i+1} and $h(x_{\text{time}}).u_i$. The application of ρ_{i+1} gives a term $m(i+1, x_{R_i}\tau_i, x_{R_i}\tau_i, r_{i+1}\tau_i, c_{i+1}\tau_i, xc\tau_i)$. We have immediately by induction hypothesis that $x_{R_i}\tau_i = R_i\sigma_i = S_{i+1}\sigma_i$ by the alternating message hypothesis. We show that $r_{i+1}\tau_i \downarrow_{\mathcal{S}_0} = compose(R_i, R_{i+1}, i)\tau_i \downarrow_{\mathcal{S}_0} = R_{i+1}\sigma_{i+1}$ and $c_{i+1}\tau_i \downarrow_{\mathcal{S}_0} = compose(R_i, M_{i+1}, i)\tau_i \downarrow_{\mathcal{S}_0} = M_{i+1}\sigma_{i+1}$. By Lemma 20, $Var(r_{i+1}) \cup Var(c_{i+1}) \subseteq Var(\ell know(R_i, i)) \cup \{x_{M_i}, x_{\text{time}}, x_{S_i}\}$. By hypothesis, the 5th argument of the $w()$ subterm of u_i is $\ell know(R_i, i)\theta_i$, therefore $\tau_i|_{Var(\ell know(R_i,i))} = \theta_i|_{Var(\ell know(R_i,i))}$. Moreover, by induction hypothesis, $x_{M_i}\tau_i = M_i\sigma_i$ and $x_{S_i}\tau_i = S_i\sigma_i$. Thus τ_i coincides with θ_i on the variables of the left hand side of ρ_{i+1}, and $r_{i+1}\tau_i = r_{i+1}\theta_i$, $c_{i+1}\tau_i = c_{i+1}\theta_i$.

By hypothesis, $compose(R_i, R_{i+1}, i) \neq \textbf{Fail}$, thus, by definition of $compose()$, Equation (10), we have that $R_{i+1} : r_{i+1} \in know(R_i, i)$. By Lemma 11, $r_{i+1}\tau_i \downarrow_{\mathcal{S}_0} = r_{i+1}\theta_i \downarrow_{\mathcal{S}_0} = R_{i+1}\sigma_i$.

The proof that $c_{i+1}\tau_i \downarrow_{\mathcal{S}_0} = M_{i+1}\sigma_i$ uses the same arguments but one can be in any of the cases (10)–(14) of $compose()$. We can apply Lemma 12 to M_{i+1} because by hypothesis, the protocol $\mathcal{P}$ is normalised.

The $w()$ subterm. We show now that t_{i+1} contains a subterm:

$$w(i+1, S_{i+1}\sigma_{i+1}, R_{i+1}\sigma_{i+1}, w_{i+1}, \ell know(R_{i+1}, i+1)\theta_{i+1}, \underline{1})$$

Since k is a bijection on $\{0, \dots, n\}$, there exists j such that $k_j = i+1$.

Case $0 \leq j \leq i$. By induction hypothesis:

$$t_j \trianglerighteq u_j := w(j, S_j\sigma_j, R_j\sigma_j, w_j, \ell know(R_j, j)\sigma_j)\theta_j, \underline{1}).m(j, S_j\sigma_j, S_j\sigma_j, R_j\sigma_j, M_j\sigma_j, \underline{1}) \quad (\star\star)$$

and conditions for applying ρ_{j+1} are satisfied by t_j. This rule application has generated the following subterm of t_{j+1} : $w(i+1, compose(R_j, S_{i+1}, j)\tau_j \downarrow_{\mathcal{S}_0}, x_{R_i}\tau_j, expect(R_j, M_{i+1}, j)\tau_j \downarrow_{\mathcal{S}_0}, \underline{1})$ (τ_j is the between the left-hand side of ρ_{j+1} and $h(x_{\text{time}}).u_j$).

Let $v = w(i+1, compose(R_j, S_{i+1}, j) \downarrow_{\mathcal{S}_0}, x_{R_i}, expect(R_j, M_{i+1}, j) \downarrow_{\mathcal{S}_0}, \underline{1})$. We can prove that for all $j+1 < m \leq i+1$, $v\theta_m$ is a subterm of t_{m+1}, *i.e.* that $v\theta_m$ is not modified in the narrowing sequence $h(x_{\text{time}}).t_{j+1} \rightsquigarrow_{\rho_{j+2}} \dots \rightsquigarrow_{\rho_{i+1}} h(x_{\text{time}}).t_{i+1}$.

On one hand no subterm of the left-hand side of ρ_{m+1} matches $v\theta_m$ due to mismatch on the first argument ($i+1$ and m).

On the other hand, the variables of $Var(v\theta_m) \setminus \{x_{\text{time}}\}$ are not instantiated when a rule ρ_m is applied. Indeed, the term $expect(R_j, M_{i+1}, j)$ contains only variables linear which do not occur elsewhere in t_m (except x_{time}). Hence they will be never instantiated by narrowing before $m = i+1$. Only every x_{time} is replaced by $s(x_{\text{time}})$. Hence $v\theta_m$ is replaced by $v\theta_{m+1}$.

Hence $v\theta_i$ is a subterm of t_{i+1}. Since τ_i coincide with θ_i on the variables of the left hand side of ρ_{i+1}, $v\tau_i$ is a subterm of t_{i+1}. We can prove as in the $m()$ case above that $compose(R_j, S_{i+1}, j)\tau_j \downarrow_{\mathcal{S}_0} = S_{i+1}\sigma_{i+1}$. We also have immediately $x_{R_i}\tau_i = R_i\sigma_i$ by definition of θ_i. It remains to show that $expect(R_j, M_{i+1}, j)\theta_i \downarrow_{\mathcal{S}_0}$ unifies with $M_{i+1}\sigma_{i+1}$. This can be proved by case inspection on M_{i+1} and by using the hypothesis that $\mathcal{P}$ is normalised.

Case $j > i$. Then by definition of k it means that t_0 contains the subterm

$$w(i+1, compose(R_j, S_{i+1}, j)\tau_j \downarrow_{\mathcal{S}_0}, x_{R_i}\tau_j, expect(R_j, M_{i+1}, j)\tau_j \downarrow_{\mathcal{S}_0}, \underline{1})$$

and the proof follows the same lines as above.

We have proved that $(\star)$ is true for every $1 \leq i \leq n-1$. It implies that $h(x_{\text{time}}).t_i \rightsquigarrow_{\rho_{i+1}} h(x_{\text{time}}).t_{i+1}$ for every for every $1 \leq i \leq n-1$. We prove similarly that since $(\star)$ is true for $i = n-1$, then $h(x_{\text{time}}).t'_0$ has the required format, by construction of ρ_n (case `[end]` in Figure 2). The difference is that ρ_n has no subterm of the form $m()$ in his right member.

The above narrowing sequence can be turned into a loop and gives an infinite sequence $\vec{s}$ of $EXEC_n(\mathcal{P})$. □

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 $t_{\text{goal}}(\mathcal{P}) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, 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:

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

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 $m(i, \ldots)$ from the first run and another $m(i, \ldots)$ from the second one. An example of this situation is given in appendix. The critical state in this case is: (recall that xc and xc' correspond to session counters)

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

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 [23], 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 21 The following scenario is a *correspondence attack* for the symmetric key version of the cable tv toy example in Figure 1:

$$\begin{array}{llcll} 1. & D & \rightarrow & I(C) & : \langle D, \{Ins\}_K \rangle \\ 2. & I(C) & \rightarrow & D & : \langle C, D, \{Ins\}_K \rangle \end{array}$$

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

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

Definition 22 *An execution $t_0, \ldots \in EXEC_i(\mathcal{P})$ 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 unary 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 secret in $\mathcal{P}$, by `secrecy_of` N. If this term t appears as an argument of i, it means that its secrecy has been corrupted by the intruder I. The critical state is then simply:

$$t_{\text{goal}}(\mathcal{P}) := i(x).secret(x)$$

Note that this term is actually independent of $\mathcal{P}$.

If N is persistent, then $secret(v)$ is added in $t_{init}(\mathcal{P})$ for each declaration $N : v$ in a line of `session_instance`. If N is fresh, the storage in $secret(\_)$ is performed in the rewrite rule for constructing the message where N appears for the first time. More precisely, there is a special construction in the rewrite rule for building the message M_{i+1} where N appears for the first time. The binding to N is ensured with the help of a side effect in function $compose()$. We assume that in the case of a `secrecy` goal, the function is redefined such that a recursive call $compose(U, N, i)$ returns $secret(t)$ as a side effect, where $t = compose(U, N, i)$. The i^{th} rule constructed by the algorithm 2 will be in this case:

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

where the term $secret(t)$ is returned by side effect during the call of $compose(R_i, M_{i+1}, i)$.

Example 23 The rules generated for the protocol of Figure 1, public key version, are:

$$\begin{array}{ll} h(s(x_{\text{time}})).w\big(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_T, x_{T[D]^{-1}}\rangle, xc\big) & (\text{tvp}_1) \\ \rightarrow h(x_{\text{time}}).m\big(1, x_D, x_D, x_C, \langle x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_{T[D]^{-1}}}\rangle, xc\big) & \\ \quad .w\big(2, x_C, x_D, \langle x_C, x_D, \{nonce(s_1(x_{\text{time}}))\}_{x_1}\rangle, & \\ \quad\quad \langle x_D, x_C, x_T, x_{T[D]^{-1}}, x_{M_0}, x_{S_0}, nonce(s_1(x_{\text{time}}))\rangle, xc\big) & \\ \quad .secret(nonce(s_1(x_{\text{time}}))) & \\ h(s(x_{\text{time}})).w\big(1, x_D, x_C, x_{M_1}, \langle x_C, x_T, x_{T[C]^{-1}}\rangle, xc\big) & (\text{tvp}_2) \\ \quad .m\big(1, x_r, x_D, x_C, x_{M_1}, xc'\big) & \\ \rightarrow h(x_{\text{time}}).m\big(2, x_C, x_D, \pi_1(x_{M_1}), \langle x_C, \pi_1(x_{M_1}), \pi_2(x_{M_1})\rangle, xc'\big) & \\ \quad .w\big(1, x_1, x_{U_1}, \langle x_1, \{x_2\}_{x_K}\rangle, \langle x_C, x_T, x_{T[C]^{-1}}\rangle, s(xc)\big) & \\ h(s(x_{\text{time}})).w\big(2, x_C, x_D, x_{M_2}, \langle x_D, x_C, x_T, x_{T[D]^{-1}}, x_{M_0}, x_{S_0}, x_{Ins}\rangle, xc\big) & (\text{tvp}_3) \\ \quad .m\big(2, x_r, x_C, x_D, x_{M_2}, xc'\big) & \\ \rightarrow h(x_{\text{time}}).w\big(0, x_{S_0}, x_D, x_{M_0}, \langle x_D, x_C, x_T, x_{T[D]^{-1}}\rangle, s(xc)\big) & \end{array}$$

Note the term $secret(nonce(s_1(x_{\text{time}})))$ in rule (tvp_1). As described in Example 21, it is easy to see that this protocol has a secrecy flaw. A subterm $secret(nonce(x)).i(nonce(x))$ 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.

Definition 24 *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:*

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

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

The critical state $t_{\text{goal}}(\mathcal{P})$ is therefore on of the two above terms in Definition 24. Again, these terms are independent from $\mathcal{P}$.

Example 25 A critical state for the protocol in Figure 1, symmetric key version, is:

$$t_{\text{goal}}(\mathcal{P}) := \quad w(0, x_1, tv, x_{M_1}, x_{l_1}, xc).w(1, x_2, scard, x_{M_2}, x_{l_2}, s(xc))$$

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 26 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: $n+1.\ B \to A : \{X\}_K$ 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 [4].

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 `symmetric_key` = `text` = `number`. By definition of profiles of $\{\_\}_\_$ and $\langle \_, \_ \rangle$, ciphers and pairs are in this case `numbers`, and be accepted as `symmetric_key`.

Example 26 A known key compromising attack on Otway-Rees protocol, see [4], exploits a type flaw of this protocol. We present here the extended version of Otway-Rees, see Section 5.5.

```
protocol Ottway Rees
identifiers
 A, B, S          :  user;
 K_as, K_bs, K_ab :  symmetric_key;
 M, N_a, N_b, X   :  number;
messages
 1.  A  →  B  :  ⟨M, A, B, {N_a, M, A, B}_{K_as}⟩
 2.  B  →  S  :  ⟨M, A, B, {N_a, M, A, B}_{K_as}, N_b, M, A, B_{K_bs}⟩
 3.  S  →  B  :  ⟨M, {N_a, K_ab}_{K_as}, {N_b, K_ab}_{K_bs}⟩
 4.  B  →  A  :  ⟨M, {N_a, K_ab}_{K_as}⟩
 5.  A  →  B  :  {X}_{K ab}
knowledge
 A  :  B, S, K_as;
 B  :  S, K_bs;
 S  :  A, B, K_as, K_bs;
session_instance  [A : a, B : b, S : s, k_as : k_as, K_bs : k_bs];
intruder : divert, impersonate;
intruder_knowledge : ;
goal : secrecy_of X;
```

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:

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

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 Implementation and experiments

We have implemented the construction of $R(\mathcal{P})$ in OCaml® and performed experiences using the theorem prover daTac [21] with resolution modulo AC. Each rule $l \rightarrow r \in R(\mathcal{P})$ is classically transformed into a Horn clause $P(l) \Rightarrow P(r)$, the initial state becomes $\Rightarrow P\big(t_{init}(\mathcal{P})\big)$ and the critical state $P\big(t_{\text{goal}}(\mathcal{P})\big) \Rightarrow$. As for multiset rewriting [5], the 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 resolution steps with equational rewriting for $\mathcal{S}_0$.

6.1 Resolution strategy

We have used a breadth first search strategy. The compilation of the protocol generates two sets of clauses:

(1) clauses representing transitions rules (including intruder's rules), the critical state (goal), and the rewrite rules of S_0;

(2) clauses representing the initial state.

The resolution strategy used by daTac is the following:

Repeat: Select a clause C in (2), C is a positive literal
  Repeat: **Select** a clause D in (1), D is of the form "$D_1 \Rightarrow D_2$"
    Apply resolution with C and D:

Compute most general ac-unifiers of C and D_1
For each solution σ,
Generate the corresponding instance of the clause $D_2\sigma$
Simplify the generated clauses:
For each generated clause $D_2\sigma$,
Select a rewrite rule $l \to r$ in (1)
For each subterm s in $D_2\sigma$,
If s is an instance of l Then Replace s by the corresponding instance of r in $D_2\sigma$
Add the simplified generated clauses in (2)
Until contradiction Or no more clause to select in (1)
Until contradiction Or no more clause to select in (2)

Note that with this strategy, a refutation tree is always linear. Such a tree is a scenario for a flaw or an attack.

6.2 Approximations for intruder rules

Due to the intruder rules of Section 3 the search space is too large. In particular, the application of rules (30)–(31) 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 (34)–(35) 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 in $\mathcal{I}$ (34), (35) by, respectively,

$$i(x).i(x_u).w(j, I, x_u, x, x_\ell, xc) \to i(x).i(x_u).w(j, I, x_u, x, x_\ell, xc).m(j, I, I, x_u, x, \underline{0}) \quad \text{(34b)}$$
where $j \leq n$

$$i(x).i(x_u).i(x'_u).w(j, x_u, x'_u, x, x_\ell, xc) \to i(x).i(x_u).i(x'_u).w(j, x_u, x'_u, x, x_\ell, xc).m(j, I, x_u, x'_u, x, \underline{0}) \quad \text{(35b)}$$
where $j \leq n$

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.

Lemma 27 $EXEC_i(\mathcal{P}) \subseteq EXEC(t_{\text{init}}(\mathcal{P}), R(\mathcal{P}) \cup \mathcal{I}_w, S_0)$

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 (30)–(33) and replacing each rules of (34b), (35b) by 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 replace (34b), (35b) in $\mathcal{I}$ by, respectively,

$$i(x_1).\dots.i(x_m).i(x_u).w(j, I, x_u, x, x_\ell, xc) \to i(x_1).\dots.i(x_m).i(x_u).w(j, I, x_u, x, x_\ell, xc).m(j, I, I, x_u, v[x_1, \dots, x_m], \underline{0})$$
$$j \leq n,\ v[x_1, \dots, x_m] \in skel(M_i)$$

$$i(x_1).\dots.i(x_m).i(x_u).i(x'_u).w(j, x_u, x'_u, x, x_\ell, xc) \to i(x_1).\dots.i(x_m).i(x_u).i(x'_u).w(j, x_u, x'_u, x, x_\ell, xc).m(j, I, x_u, x'_u, v[x_1, \dots, x_m], \underline{0})$$
$$j \leq n,\ v[x_1, \dots, x_m] \in skel(M_i)$$

$v[x_1, \dots, x_m]$ denotes a term of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ whose variables are $x_1, \dots, x_m$, and the set $skel(M_i)$ of skeletons of M_i is recursively defined by:

$$\begin{aligned}
skel(\langle M_1, M_2 \rangle) &= \{t_1, t_2 \mid t_1 \in skel(M_1), t_2 \in skel(M_2)\} \\
skel(\{M\}_{KA}) &= \{\{t\}_{x_1} \mid t \in skel(M),\ x_1 \text{ is a fresh variable}\} \cup \{x_2 \text{ fresh var}\} \\
skel(T[A]) &= \{x_1[x_2] \mid x_1, x_2 \text{ are fresh variables}\} \cup \{x_3 \text{ fresh var}\} \\
skel(F(M)) &= \{x_1(t) \mid x_1 \text{ is a fresh variable},\ t \in skel(M)\} \cup \{x_2 \text{ fresh var}\}
\end{aligned}$$

Because of deletion of rules (30)-(33), one rule for public key decryption with tables needs to be added:

$$i(\{x_1\}_{x_t[x_u]^{-1}}).i(x_t).i(x_u) \rightarrow i(\{x_1\}_{x_t[x_u]^{-1}}).i(x_p).i(x_u).i(x_1) \quad (36)$$

The obtained system depends on $\mathcal{P}$. Lets call it $\mathcal{I}'_w(\mathcal{P})$.

Lemma 28 $EXEC(t_{\text{init}}(\mathcal{P}), R(\mathcal{P}) \cup \mathcal{I}'_w(\mathcal{P}, S_0) \subset EXEC(t_{\text{init}}(\mathcal{P}), R(\mathcal{P}) \cup \mathcal{I}_w, S_0)$

Note that there is no equality here, *i.e.* this approximation is not complete. However, it seems to give reasonable results in practice.

6.3 Results

The approach has been experimented with several protocols described in [4]. 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 4.

Protocol	Description	Flaw	Intruder abilities
Encrypted Key Exchange	Key distribution	Correspondence attack	divert impersonate
Needham Shroeder Public Key	Key distribution with authentification	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	Authentification	Correspondence attack	divert impersonate

Figure 4: Experiments

For more details, see `http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/`.

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, that the narrowing permits to handle several aspect 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] David Basin. Lazy infinite-state analysis of security protocols. In *Secure Networking — CQRE [Secure] '99*, LNCS 1740, pages 30–42. Springer-Verlag, Berlin, 1999.

[3] D. Bolignano. Towards the formal verification of electronic commerce protocols. In *IEEE Computer Security Foundations Workshop*, pages 133–146. IEEE Computer Society, 1997.

[4] J. Clark and J. Jacob. A survey of authentication protocol literature. `http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps`, 1997.

[5] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *Formal Methods and Security Protocols*, 1998. LICS '98 Workshop.

[6] G. Denker and J. Millen. Capsl intermediate language. In *Formal Methods and Security Protocols*, 1999. FLOC '99 Workshop.

[7] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. North-Holland, 1990.

[8] 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.

[9] R. Focardi, A. Ghelli, and R. Gorrieri. Using non interference for the analysis of security protocols. In *DIMACS Workshop on Design and Verification of Security Protocols*. Rutgers U., 1997.

[10] 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.

[11] J.-M. Hullot. Canonical forms and unification. In *5th International Conference on Automated Deduction*, volume 87, pages 318–334. Springer-Verlag, LNCS, july 1980.

[12] G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[13] 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.

[14] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.

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

[16] J. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[17] 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.

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

[19] 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.

[20] B. Schneier. *Applied Cryptography*. John Wiley, 1996.

[21] L. Vigneron. Positive deduction modulo regular theories. In *Proceedings of Computer Science Logic, Paderborn (Germany)*, pages 468–485. LNCS 1092, Springer-Verlag, 1995.

[22] 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.

[23] 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 29

`identifiers`	M, B, C	:	`user`;			
	O, N	:	`number`;			
	K_b, K_c	:	`public_key`;			
	$hash$	:	`function`;			
`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	:	$\Big\{hash\big(hash(N), hash(O)\big)\Big\}_{K_b^{-1}}$	
`knowledge`	C	:	$B, K_b, hash$;			
	M	:	$C, O, K_c, K_b, hash$;			
	B	:	$K_b, K_b^{-1}, hash$;			
`session_instance`	$[M : Merchant,\ B : Bank,\ C : Customer,\ O : car,\ K_b : k_b,\ K_c : k_c]$					
	$[M : Merchant,\ B : Bank,\ C : Customer,\ O : peanut,\ K_b : k_b,\ K_c : k_c]$					

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) \\
& \rightsquigarrow_{(\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) \\
& \rightsquigarrow_{(24)} \\
& 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) \\
& \rightsquigarrow_{(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(\{nonce(x_1)\}_{key}) \\
& \rightsquigarrow_{(35)} \\
& 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}) \\
& \rightsquigarrow_{(\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, x3, \langle tv, scard, key, key[tv]^{-1}\rangle, 1) \\
&\qquad .w(1, x4, scard, \langle x4, x5\rangle, \langle scard, key, key[scard]^{-1}\rangle, 1) \\
&\qquad .i(key) \\
&\rightsquigarrow^{(\mathrm{tvp}_1)} \\
&h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, 1) \\
&\qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x1)\rangle, 1) \\
&\qquad .w(1, x4, scard, \langle x4, x5\rangle, \langle scard, key, key[scard]^{-1}\rangle, 1) \\
&\qquad .secret(nonce(x1)) \\
&\qquad .i(key) \\
&\rightsquigarrow^{(25)} \\
&h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, 1) \\
&\qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x1)\rangle, 1) \\
&\qquad .w(1, x4, scard, \langle x4, x5\rangle, \langle scard, key, key[scard]^{-1}\rangle, 1) \\
&\qquad .secret(nonce(x1)) \\
&\qquad .i(key).i(tv).i(scard).i(\langle tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle) \\
&\rightsquigarrow^{(26)} \\
&h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, 1) \\
&\qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x1)\rangle, 1) \\
&\qquad .w(1, x4, scard, \langle x4, x5\rangle, \langle scard, key, key[scard]^{-1}\rangle, 1) \\
&\qquad .secret(nonce(x1)) \\
&\qquad .i(key).i(tv).i(scard).i(\{nonce(x1)\}_{key[tv]^{-1}}) \\
&\rightsquigarrow^{(33)} \\
&h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, 1) \\
&\qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x1)\rangle, 1) \\
&\qquad .w(1, x4, scard, \langle x4, x5\rangle, \langle scard, key, key[scard]^{-1}\rangle, 1) \\
&\qquad .secret(nonce(x1)) \\
&\qquad .i(key).i(tv).i(scard).i(\{nonce(x1)\}_{key[tv]^{-1}}).i(key[tv]) \\
&\rightsquigarrow^{(29)} \\
&h(x_1).m(1, tv, tv, scard, \langle tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, 1) \\
&\qquad .w(2, scard, tv, \langle scard, tv, \{nonce(x1)\}_{key[tv]^{-1}}\rangle, \langle tv, scard, key, key[tv]^{-1}, x_2, x_3, nonce(x1)\rangle, 1) \\
&\qquad .w(1, x4, scard, \langle x4, x5\rangle, \langle scard, key, key[scard]^{-1}\rangle, 1) \\
&\qquad .secret(nonce(x1)) \\
&\qquad .i(key).i(tv).i(scard).i(\{nonce(x1)\}_{key[tv]^{-1}}).i(key[tv]).i(nonce(x_1))
\end{aligned}
$$

The subterm $secret(nonce(x1)).i(nonce(x_1))$ matches the pattern $t_{\mathrm{goal}}(\mathcal{P})$.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399