# Towards Efficient Automated Verification of Security Protocols

Y. Chevalier and L. Vigneron

LORIA – UHP – UN2
Campus Scientifique, B.P. 239
54506 Vandœuvre-lès-Nancy Cedex, France
{chevalie,vigneron}@loria.fr

## Introduction

The verification of cryptographic protocols has been intensively studied these last years. A lot of methods have been defined for analyzing particular protocols [14, 3, 5, 16, 19, 9]. Some tools (Casper [11], CVS [8], CAPSL [6]) have also been developed for automating one of the most sensitive step: the translation of a protocol specification into a low-level language that can be handled by automated verification systems.

Our work is in this last line. We have designed a protocols compiler Casrul [10] that translates a cryptographic protocol specification to a set of rewrite rules. These generic rules are then turned into rules for the theorem prover daTac. This translation step permits, through static analysis of the protocol, to rule out many errors while being protocol independent. A comparison of Casrul with systems such as CAPSL and Casper can be found in [10].

It is also possible to use Casrul to have a representation of the protocol in various systems:

- As Horn clauses, they can be used by theorem provers in first-order logic, or as a Prolog program.
- As rewrite rules, they can be used by inductive theorem provers, or as an ELAN program.
- As propositional formulas, they can be used by SAT.

In our case, we use the theorem prover daTac for trying to find flaws in protocols. The technique implemented in daTac is narrowing. This unification-based technique permits us to handle infinite states models, and also to guarantee the freshness of the randomly generated nonces or keys [10]. Note that there was a first approach with narrowing by Meadows in [12].

The main objective of this paper is, after giving a general presentation of Casrul in Section 1, to present in Section 2 an innovative model of the intruder behavior, based on the definition of a lazy model. This lazy approach is completely different and much more efficient than the model of the intruder presented in [10]. In Section 3, we show that our method can be successfully applied to many different kinds of protocols. We explain the results obtained for two protocols and we give a summary of flaws found in other protocols.

1

# 1   Input Protocols

We present in this section the syntax used for describing security protocols, illustrated in Figure 1. This syntax has been fully detailed in [10], and is close to one of CAPSL [13] or Casper [11] though it differs on some points – for instance, on those in Casper which concern CSP. All the notions we will use for protocols are classical and can be found in [18]

We also present some algorithms for verifying the correctness and run-ability of the protocol.

These algorithms are implemented in our compiler, Casrul[1], that transforms a protocol given as in Figure 1 into a set of rewrite rules. In [10], we have proved that this compilation defines a non-ambiguous operational semantics for protocols and intruder behavior.

---

**Protocol WLMA;**
**Identifiers**
$Q, P, S$       : *User*;
$Np, Nq$       : *Number*;
$Kpq, Kps, Kqs$ : *Symmetric_key*;
**Knowledge**
$Q$                : $P, S, Kqs$;
$P$                : $Q, S, Kps$;
$S$                : $P, Q, Kps, Kqs$;
**Messages**
1. $P \rightarrow Q$    : $P, Np$
2. $Q \rightarrow P$    : $Q, Nq$
3. $P \rightarrow Q$    : $\{P, Q, Np, Nq\}Kps$
4. $Q \rightarrow S$    : $\{P, Q, Np, Nq\}Kps, \{P, Q, Np, Nq\}Kqs$
5. $S \rightarrow Q$    : $\{Q, Np, Nq, Kpq\}Kps, \{P, Np, Nq, Kpq\}Kqs$
6. $Q \rightarrow P$    : $\{Q, Np, Nq, Kpq\}Kps, \{Np, Nq\}Kpq$
7. $P \rightarrow Q$    : $\{Nq\}Kpq$
**Session_instances**
$[P : a; Q : I; S : se; Kqs : kis; Kps : kas]$
$[P : I; Q : a; S : se; Kps : kis; Kqs : kas]$;
**Intruder**   *Divert, Impersonate*;
**Intruder_knowledge**   $a, se, kis$;
**Goal** *Correspondence_between Q S*;

---

**Fig. 1.** Woo and Lam Mutual Authentification Protocol.

The information given for describing a protocol can be decomposed into two parts: the description of the protocol itself, and the instances and strategies to be used for verifying it.

---

[1] `http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/`

## 1.1 Main Information

The description of a protocol is the composition of three types of information: the identifiers, the messages, and the initial knowledge. Let us present each of these.

**Identifiers.** This section contains the declaration of all the identifiers used in the protocol messages. This includes principals (users), keys (symmetric, public/private, table), random numbers (also called nonces), hash functions. Some of those identifiers will be used as fresh information, i.e. they will be generated during the execution of the protocol.

Let us give more details about the three kinds of keys:

- A *symmetric key* is a key that is used to decode what it has encoded. For instance, if a message $M$ is encoded by a symmetric key $K$, written $\{M\}_K$, then $\{\{M\}_K\}_K$ is equal to $M$.
- A *public key* is a key known by everybody; to each public key $K$ corresponds a unique *private key* $K^{-1}$ able to decode what $K$ has encoded: $\{\{M\}_K\}_{K^{-1}}$ is equal to $M$. In general, a private key is associated to a principal and can be used as a signature: $\{\{M\}_{K^{-1}}\}_K$ is equal to $M$.
- A *table* $T$ associates a public and a private key to the name of a principal $A$: $T[A]$ and $T[A]^{-1}$. Initially, only the owner of the table knows those keys.

**Initial knowledge.** The last information needed for a complete description of a protocol is the list of initial knowledge of each principal.

An identifier (key or number) that is not in any initial knowledge will be used as a *fresh* information, created at its first use.

**Messages.** They describe the different steps of the protocol with, for each one, its index, the name of its sender, the name of its receiver, and the body of the message itself. The syntax is very classical for encoding: $\{M\}_K$ means the message $M$ encode by the key $K$. We also allow $Xor$ encryption with the notation $(M)\mathsf{xor}(T)$, in which we assume $M$ and $T$ are two expressions of the same size, thus getting rid of bloc properties of *xor* encryption.

All this information brings a precise view of the proposed protocol, and at this point we should be able to run the protocol. However, the model of a principal is not complete: we have to check the protocol is correct and runnable, by verifying the evolution of the knowledge of each principal.

## 1.2 Correctness of the Protocol

The knowledge of the principals in a protocol is always changing. One has to verify that all the messages can be composed and sent to the right person, to guarantee the protocol can be run.

The knowledge of each participant can be decomposed into three parts:

3

– the initial knowledge, declared in the protocol,
– the acquired knowledge, obtained by decomposition of the received messages,
– the generated knowledge, created for composing a message (fresh knowledge).

A protocol is correct and runnable if each principal can compose the messages it is supposed to send. For some messages, principals will use parts of the received messages. So, a principal has to update its knowledge as soon as it receives a message: it has to store the new information, and check if it can be used for decoding old ciphers (i.e. parts of received messages it could not decode because it did not have the key).

The following function describes the composition of a message $M$ by a principal $U$ at step $i$. Its knowledge is therefore the union of its initial knowledge and the information it could get in the received and sent messages, until step $i - 1$ (included). For an easier reuse of this knowledge, a name is assigned to each information.

As the message $M$ will be sent by $U$ at step $i$, any problem will generate a failure in this function.

$$compose(U, M, i) = t \qquad \text{if } M \text{ is known by } U \text{ and named } t$$
$$compose(U, \langle M_1, M_2 \rangle, i) = \langle compose(U, M_1, i), compose(U, M_2, i) \rangle$$
$$compose(U, (M_1)\mathsf{xor}(M_2), i) = (compose(U, M_1, i))\mathsf{xor}(compose(U, M_2, i))$$
$$compose(U, \{M\}_K, i) = \{compose(U, M, i)\}_{compose(U,K,i)}$$
$$compose(U, T[A], i) = compose(U, T, i)[compose(U, A, i)]$$
$$compose(U, T[A]^{-1}, i) = compose(U, T, i)[compose(U, A, i)]^{-1}$$
$$compose(U, M, i) = nonce(x_{time}) \qquad \text{if } M \text{ is a nonce}$$
$$compose(U, M, i) = \texttt{Fail} \qquad \text{else}$$

In addition to being able to compose the messages, a principal has also to be able to verify the information received in messages: if it is supposed to receive an information it already knew, it has to check it is really the same. A principal also knows the shape of the messages it receives. So it has to be able to check that everything it can access in a received message corresponds to what it expected. These verifications are done by the following function, where a principal $U$ tries to compose an expected message $M$ using its knowledge before step $i$, the step when he will receive this message. All the unknown ciphers are replaced by new variables.

$$expect(U, M, i) = compose(U, M, i) \qquad \text{if no Fail}$$
$$expect(U, \langle M_1, M_2 \rangle, i) = \langle expect(U, M_1, i), expect(U, M_2, i) \rangle$$
$$expect(U, (M_1)\mathsf{xor}(M_2), i) = (compose(U, M_1, i))\mathsf{xor}(expect(U, M_2, i)) \quad \text{if no Fail}$$
$$expect(U, (M_1)\mathsf{xor}(M_2), i) = (expect(U, M_1, i))\mathsf{xor}(compose(U, M_2, i)) \quad \text{if no Fail}$$
$$expect(U, \{M\}_K, i) = \{expect(U, M, i)\}_{compose(U,K^{-1},i)^{-1}} \qquad \text{if no Fail}$$
$$expect(U, \{M\}_{K^{-1}}, i) = \{expect(U, M, i)\}_{compose(U,K,i)^{-1}} \qquad \text{if no Fail}$$
$$expect(U, \{M\}_{SK}, i) = \{expect(U, M, i)\}_{compose(U,SK,i)} \qquad \text{if no Fail}$$
$$expect(U, M, i) = x_{U,M,i} \qquad \text{else}$$

4

Note that $K$ stands for a public key, $K^{-1}$ for a private key (possibly through the use of a table), and $SK$ for a symmetric key.

These algorithms (and others) are implemented in Casrul. This compiler can therefore generate rewrite rules that model the behavior of principals: to wait for a message and then to send a new one.

$$expect(U, M_i, i) \Rightarrow compose(U, M_{i+1}, i + 1)$$

This kind of model was already used by Dolev and Yao in [7].

## 1.3 Additional Information

Verifying a protocol consists in trying to simulate what an intruder could do for disturbing the run of the protocol, without some participants knowing. In this purpose, we require more information in the protocol description.

**Session instances.** This field proposes some possible values to be assigned to the persistent identifiers and thus describes the different systems (in the sense of Casper [11]) for running the protocol. The different sessions can take place concurrently or sequentially an arbitrary number of times. Note that we can mention that some sessions are between honest principals, and some other sessions involve the intruder ($I$), playing the role of one of the principals.

**Intruder.** The Intruder field describes which strategies the intruder can use, among passive eaves_dropping, divert and impersonate. If nothing is specified, this means that we want a simulation of the protocol without intruder.
The intruder can record all the messages spread over the net. In addition, if divert is selected, it can remove messages; if eaves_dropping is selected, it cannot.
The intruder is then able to reconstruct terms as it wishes, using all the information it got. It can send arbitrary messages in his own name.
If moreover impersonate is selected, then it can fake others identity in sent messages.
We will focus on the case where the intruder may divert messages and impersonate principals.

**Intruder knowledge.** The Intruder_knowledge is the list of information known from the beginning by the intruder. Contrarily to the initial knowledge of others principals, each element of the intruder knowledge has to have been introduced in Session_instances, as an effective knowledge (and not a formal one used for describing the messages).

**Goal.** This field gives the kind of flaw we want to detect. There are two families of goals, `Correspondence_between` and `Secrecy_of`.

*Secrecy* means that some secret information (e.g. a key or a number) 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.

We do not detail here how those goals are specified by Casrul, and how they are checked. This does not differ from the specifications given in [10].

## 2   Intruder's model

One of the biggest problem in the area of cryptographic protocols verification is the definition of the intruder. The Dolev-Yao's model of an intruder [7] is not scalable, since there are rules for composing messages, and these rules such as building a couple from two terms, do not terminate: given a term, it is possible to build a couple with two copies of this term, and to do it again with that couple.

In some approaches people try to bound the size of the messages, but these bounds are valid when one considers specific kinds of protocols and/or executions. We want to be able to study all the protocols definable within the Casrul syntax, and to get a system that is as independent as possible of the number of sessions. Thus, these bounds are not relevant in our approach, and this led us to bring a new approach.

A proposed approach to deal with this infinite-space problem is to use a lazy model while testing the protocol by model-checking [2]. We rather use this lazy-analysis approach on the intruder. We replace the terms building step of the intruder with a step in which, at the same time, the intruder analyzes its knowledge and tests if it can build a term matching the message awaited by a principal. The pattern of the awaited message being given by the principal, we can describe our intruder's model as a "lazy" one.

First, we briefly present the system testing if terms can be built. Then, we define a system for decomposing the intruder's knowledge, relying on the testing system. It is remarkable that the knowledge decomposition using this system now allows decomposition of ciphers with composed key (see the Otway-Rees example in Section 3.2) and even the *xor*-encryption, whereas other similar models such as [1] only allow atomic symmetric keys.

For the next two sections, we have to give the meaning of the terms in the rewrite rules generated by Casrul.

- Atomic terms are those constants described in the `Session_instances` field;
- Some unary operators are used to type those constants,such as MR to describe a principal; we also use F for representing any of those operators;
- The C operator stands for coupling;
- CRYPT, SCRYPT and XCRYPT operators stand respectively for public or private key encryption, symmetric key encryption and *xor*-encryption.

We also use other operators whose meaning should be clear from the name, e.g. the *Comp* operator, except maybe for the "." operator, which is not a list constructor, but an AC operator.

## 2.1 Test of the Composition of a Term

The heart of our intruder's model is to *test* if a term matching a term $t$ can be composed from a knowledge set $C$. The rewriting system described in Figure 2 tries to reduce $Comp(t)$ from $C$ ; $Id$, building a substitution $\tau$.

$$Comp(t).T \text{ from } t.C \; ; \; \tau \; \rightarrow \; T \text{ from } t.C \; ; \; \tau \qquad (1)$$

$$Comp(r).T \text{ from } s.C \; ; \; \tau \overset{r\sigma = s\sigma}{\rightarrow} T\sigma \text{ from } s.C\sigma \; ; \; \tau\sigma \qquad (2)$$

$$Comp(\text{C}(t_1, t_2)).T \text{ from } C \; ; \; \tau \; \rightarrow \; Comp(t_1).Comp(t_2).T \text{ from } C \; ; \; \tau \quad (3)$$

$$Comp(\text{CRYPT}(t_1, t_2)).T \text{ from } C \; ; \; \tau \; \rightarrow \; Comp(t_1).Comp(t_2).T \text{ from } C \; ; \; \tau \quad (4)$$

$$Comp(\text{SCRYPT}(t_1, t_2)).T \text{ from } C \; ; \; \tau \; \rightarrow \; Comp(t_1).Comp(t_2).T \text{ from } C \; ; \; \tau \quad (5)$$

$$Comp(\text{XCRYPT}(t_1, t_2)).T \text{ from } C \; ; \; \tau \; \rightarrow \; Comp(t_1).Comp(t_2).T \text{ from } C \; ; \; \tau \quad (6)$$

**Fig. 2.** System to test if a term may be composed from some knowledge.

This system, being complete in the sense that it can find all the ways of composing a term cannot be confluent since two different ways will lead to two different normal forms. It also heavily relies on the fact that we *do not* use the rule (2) when the term $t$ is a variable, thereby reducing the test of the composability of a term to the test of the composability of some of its variables, which can then be instantiated later.

For example, from the Intruder's knowledge $\text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na))$, we may test if a term matching $\text{CRYPT}(\text{C}(\text{MR}(a), x_1), \text{SCRYPT}(sk(k_a), x_2))$ can be built:

$$Comp(\text{CRYPT}(\text{C}(\text{MR}(a), x_1), \text{SCRYPT}(sk(k_a), x_2)))$$
$$\text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \; ; \; Id$$
$$\rightarrow^{(4)} Comp(\text{C}(\text{MR}(a), x_1)).Comp(\text{SCRYPT}(sk(k_a), x_2))$$
$$\text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \; ; \; Id$$
$$\rightarrow^{(3)} Comp(\text{MR}(a)).Comp(x_1).Comp(\text{SCRYPT}(sk(k_a), x_2))$$
$$\text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \; ; \; Id$$
$$\rightarrow^{(1)} Comp(x_1).Comp(\text{SCRYPT}(sk(k_a), x_2))$$
$$\text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \; ; \; Id$$
$$\rightarrow^{(2)} Comp(x_1)$$
$$\text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \; ; \; \sigma$$

The test is successful, generating the substitution $\sigma : x_2 \leftarrow \text{NONCE}(Na)$ in the last step. This is the only solution. In general, we have to explore all the possible solutions. Note that we stop, accepting the composition, as soon as there are only variables left in the *Comp* terms.

## 2.2 Decomposition of the Intruder's Knowledge

In Dolev-Yao's model, all the messages sent by the principals acting in the protocol are sent to the intruder. The intruder has then the possibility to decompose the terms he knows, including the last message, and build a new one, faked so as it appears it has been sent by another principal (chosen by the intruder). We define a system that keeps in the predicate *UFO* the data that are not already treated by the intruder, and moves the non-decomposed knowledge out of *UFO*. For the decryption of a cipher (but this should also apply to hash functions), we use a predicate and a conditional rewrite rule. The resulting system described in Figure 3 only deals with *decomposing* the knowledge of the intruder, where we are always using, together with the fourth rule, the equality $t^{\text{-}1\text{-}1} = t$.

$$C.UFO(\text{F}(t).C') \to C.t.UFO(C') \tag{1}$$

$$C.UFO(\text{C}(t_1, t_2).C') \to C.UFO(t_1.t_2.C') \tag{2}$$

$$C.UFO(\text{CRYPT}(t_1, t_2).C') \to C.\text{CRYPT}(t_1, t_2).UFO(\text{TEST}(\text{CRYPT}(t_1, t_2)).C') \tag{3}$$

$$A(t_1^{\text{-}1}, C, \sigma) : C.UFO(\text{TEST}(\text{CRYPT}(t_1, t_2)).C') \to C.UFO(t_2.C')\sigma \tag{4}$$

$$C.UFO(\text{SCRYPT}(t_1, t_2).C') \to C.\text{SCRYPT}(t_1, t_2).UFO(\text{TEST}(\text{SCRYPT}(t_1, t_2)).C') \tag{5}$$

$$A(t_1, C, \sigma) : C.UFO(\text{TEST}(\text{SCRYPT}(t_1, t_2)).C') \to C.UFO(t_2.C')\sigma \tag{6}$$

$$C.UFO(\text{XCRYPT}(t_1, t_2).C') \to C.\text{XCRYPT}(t_1, t_2).UFO(\text{TEST}(\text{XCRYPT}(t_1, t_2)).C') \tag{7}$$

$$A(t_1, C, \sigma) : C.UFO(\text{TEST}(\text{XCRYPT}(t_1, t_2)).C') \to C.UFO(t_2.C')\sigma \tag{8}$$

$$A(t_2, C, \sigma) : C.UFO(\text{TEST}(\text{XCRYPT}(t_1, t_2)).C') \to C.UFO(t_1.C')\sigma \tag{9}$$

**Fig. 3.** Knowledge Simplifications System.

$A(t, C, \sigma)$ is a predicate that is true whenever the term $t$ can be build from the knowledge $C$ using a substitution $\sigma$. The system of Figure 2 shows that this predicate can be implemented with rewrite rules similar to those that are used to test if a principal can compose a message that matches the pattern of an awaited message.

## 2.3 Use this Model for Flaws Detection

We can decompose the sequence of steps the intruder uses to send a message:

1. First, it chooses a principal, which gives a pattern $m$ that the intruder's message should match. At the same time, it can give the pattern of the message $t$ that the intruder will receive if it succeeds in sending a message;
2. Second, the intruder analyzes its knowledge and tests if it can compose a message matching this pattern;
3. If it can send a message matching the pattern $m$, it goes back to step 1.

The only thing to add is that, in our model, the intruder has to keep track of all the previously sent messages. Thus, we maintain a list of previously sent messages with the knowledge at the time the messages were sent:

$$l \stackrel{\text{def}}{=} (T_1 \; from \; C_1) : \ldots : (T_n \; from \; C_n)$$

This is used, for instance in the example of Section 2.1, to prove it is sound to substitute NONCE($Na$) for $x_2$.

We also maintain a set of knowledge $C$ representing the intruder's knowledge evolution whenever it succeeds in sending an appropriate message. We model a protocol step with the rule:

$$(C, l) \rightarrow (C.t, l : (m \text{ from } C))$$

Comparing this model to an execution model where an Oracle tells a message (ground term) that is accepted by the principal, and the intruder has to verify it can send this message, this exhaustive exploration system turns out to be both sound and complete as long as we consider only a bounded number of sessions. The variables here are untyped, thus allowing the discovery of type flaws and messages of unbounded size.

## 3 Experimentations

We give a few hints on how to use our system through two examples of protocol analysis taken from the literature. First, we study the Otway-Rees un-amended protocol, which has a type flaw leading to a secrecy flaw. The EKE protocol shows how we deal with parallel sessions.

But first, let us give a short presentation of the prover used.

### 3.1 The Prover daTac

For studying the protocols, we have used the theorem prover daTac[2], specialized for automated deduction in first-order logic, with equality and associative-commutative operators. This last property is important, since we use an AC operator for representing the list of messages at a given state. Hence, asking for one message in this list consists in trying all the possible solutions.

The deduction techniques used by daTac are Resolution and Paramodulation. They are combined with efficient simplification techniques for eliminating redundant information. The interested reader may refer to [17].

For connecting Casrul and daTac, we have designed a tiny tool, Casdat, running Casrul and translating its output into a daTac input file.

### 3.2 The Otway-Rees Protocol

The Casrul specification of this well-known protocol is given in Figure 4. To study this protocol, we only have to compile this specification to daTac rules and to apply the theorem prover daTac on the generated file, leaving the result in the Otway-Rees.exe file:

---

[2] http://www.loria.fr/equipes/protheo/SOFTWARES/DATAC/

```
Protocol Otway_Rees;
Identifiers
A, B, S        : User;
Kas, Kbs, Kab : Symmetric_key;
M, Na, Nb, X   : Number;
Knowledge
A              : B, S, Kas;
B              : S, Kbs;
S              : A, B, Kas, Kbs;
Messages
1. A  → B      : M, A, B, {Na, M, A, B}Kas
2. B  → S      : M, A, B, {Na, M, A, B}Kas, {Nb, M, A, B}Kbs
3. S  → B      : M, {Na, Kab}Kas, {Nb, Kab}Kbs
4. B  → A      : M, {Na, Kab}Kas
5. A  → B      : {X}Kab
Session_instances
[A : a; B : b; S : se; Kas : kas; Kbs : kbs];
Intruder   Divert, Impersonate;
Intruder_knowledge  a;
Goal Secrecy_Of X;
```

**Fig. 4.** Otway-Rees Protocol.

```
dendron% casdat Otway-Rees.cas
dendron% rdatac -i Otway-Rees.dat -r o Otway-Rees.exe
```

The trace of execution is quite hard to analyze if you are not familiar with the techniques implemented in daTac, but hopefully, the result is the sequence of derivations leading to the discovering of the flaw *(in 1.5s)*:

```
> Inference steps to generate the empty clause:
    60 = Resol (1,56)        60 = Simpl (11,60)        . . .
    60 = Simpl (34,60)       63 = Resol (5,60)         63 = Simpl (11,63)
    . . .                    63 = Simpl (30,63)        66 = Resol (44,63)
    66 = Simpl (14,66)       . . .                     66 = Simpl (52,66)
    66 = Clausal Simpl({45},66)
```

Now, we just have to look at the given trace to figure out the scenario that leads to the secrecy flaw. We simply have to look at the clauses after each resolution, and when all the simplifications are done.

The first one is pretty simple, since it is nothing but the first principal sending its first message. All the simplifications following correspond to the decomposition of this message to Intruder's knowledge. We thus have:

```
a  →  _  : M, a, b, {Na, M, a, b}kas
```

The second resolution $(63 = Resol(5, 60))$ is much more exotic, since it is the reception of the message labelled 4 in the protocol by principal $a$. Using the protocol's specification, it is first read as:

```
a  →  _  : M, a, b, {Na, M, a, b}kas
_  →  a  : M, {Na, x5}kas
a  →  _  : {X}x5
```

At this point, we can only say that the intruder has tried to send to the principal $a$ a message *matching* $M, \{Na, x5\}Kas$. It has no choice but to unify ($66 = Resol(44, 63)$) the term yielded after the first message with the required pattern. Now, the sequence of messages becomes:

```
a  →  _  : M, a, b, {Na, M, a, b}kas
_  →  a  : M, {Na, M, a, b}kas
a  →  _  : {X}(M, a, b)
```

The intruder has proved that it can send a term matching the pattern of awaited message, so we can go on to the next step ($66 = Simpl(52, 66)$). But after that, decomposing what it knows, the intruder finds himself knowing $X$, that should have remained secret. The last move (Clausal Simplification) stamps this contradiction out, thus ending the study of this protocol.

### 3.3   The Encrypted Key Exchange *(EKE)* Protocol

We shall now study the EKE protocol, known to have a parallel correspondence-between-principals attack. The Casrul specification of this protocol is given in Figure 5.

```
Protocol EKE;
Identifiers
A, B          : User;
Na, Nb        : Number;
Ka            : Public_key;
P, R          : Symmetric_key;
Knowledge
A             : B, P;
B             : P;
Messages
1. A  →  B   : {Ka}P
2. B  →  A   : {{R}Ka}P
3. A  →  B   : {Na}R
4. B  →  A   : {Na, Nb}R
5. A  →  B   : {Nb}R
Session_instances
[A : a; B : b; P : p]
[A : b; B : a; P : p];
Intruder   Divert, Impersonate;
Intruder_knowledge  ;
Goal Correspondence_between A B;
```

**Fig. 5.** Encrypted Key Exchange Protocol.

The trace of execution is now a bit longer, but it nonetheless leads to a flaw of correspondence between principals:

```
> Inference steps to generate the empty clause:
87 = Resol (1,82)        87 = Simpl (12,87)        . . .
92 = Resol (2,87)        92 = Simpl (12,92)        . . .
92 = Simpl (31,92)       98 = Resol (28,92)        98 = Simpl (78,98)
105 = Resol (3,98)       105 = Simpl (12,105)      . . .
105 = Simpl (31,105)     118 = Resol (28,105)      118 = Simpl (49,118)
141 = Resol (4,118)      141 = Simpl (12,141)      . . .
141 = Simpl (31,141)     172 = Resol (28,141)      172 = Simpl (74,172)
220 = Resol (5,172)      220 = Simpl (12,220)      . . .
220 = Simpl (31,220)     274 = Resol (28,220)      274 = Simpl (72,274)
274 = Clausal Simpl ({59},274)
```

We can study in deeper details this trace in order to find the scenario of the attack. Since the principals appear in two sessions, we'll give, right after the name of the principal, the number (1 or 2) of the protocol session the message belongs to. First of all, the first principal of the first session starts with sending its first message:

$$a(1) \ \rightarrow \ \_ \ : \{Ka\}P$$

Then, the intruder tries to send a message to the second principal of the second session ($a$ again). To find this, one has to look closer to the generated clauses and find in which session the message was sent to:

$$
\begin{aligned}
a(1) \ &\rightarrow \ \_ \ : \{Ka\}P \\
\_ \ &\rightarrow \ a(2) \ : \{x_1\}P \\
a(2) \ &\rightarrow \ \_ \ : \{\{R\}x_1\}P
\end{aligned}
$$

The intruder now has to prove it could send the message $\{x_1\}P$, which is easily done through unification which the knowledge gained from the first message ($98 = Resol(28, 92)$). The messages sent are now:

$$
\begin{aligned}
a(1) \ &\rightarrow \ \_ \ : \{Ka\}P \\
\_ \ &\rightarrow \ a(2) \ : \{Ka\}P \\
a(2) \ &\rightarrow \ \_ \ : \{\{R\}Ka\}P
\end{aligned}
$$

and the intruder can go on to the next message, sending it to the principal $a$ in the first session ($105 = Resol(3, 98)$):

$$
\begin{aligned}
a(1) \ &\rightarrow \ \_ \ : \{Ka\}P \\
\_ \ &\rightarrow \ a(2) \ : \{Ka\}P \\
a(2) \ &\rightarrow \ \_ \ : \{\{R\}Ka\}P \\
\_ \ &\rightarrow \ a(1) \ : \{\{x_1\}Ka\}P \\
a(1) \ &\rightarrow \ \_ \ : \{Na\}x_1
\end{aligned}
$$

Then again, one can use unification ($118 = Resol(28, 105)$) to prove the intruder could send a matching message, thus yielding:

$$
\begin{array}{lll}
a(1) \ \rightarrow \ \_ & : \{Ka\}P \\
\_ \ \rightarrow \ a(2) & : \{Ka\}P \\
a(2) \ \rightarrow \ \_ & : \{\{R\}Ka\}P \\
\_ \ \rightarrow \ a(1) & : \{\{R\}Ka\}P \\
a(1) \ \rightarrow \ \_ & : \{Na\}R
\end{array}
$$

Now, the intruder can go on like this until it arrives at this point:

$$
\begin{array}{lll}
a(1) \ \rightarrow \ \_ & : \{Ka\}P \\
\_ \ \rightarrow \ a(2) & : \{Ka\}P \\
a(2) \ \rightarrow \ \_ & : \{\{R\}Ka\}P \\
\_ \ \rightarrow \ a(1) & : \{\{R\}Ka\}P \\
a(1) \ \rightarrow \ \_ & : \{Na\}R \\
\_ \ \rightarrow \ a(2) & : \{Na\}R \\
a(2) \ \rightarrow \ \_ & : \{Na, Nb\}R \\
\_ \ \rightarrow \ a(1) & : \{Na, Nb\}R \\
a(1) \ \rightarrow \ \_ & : \{Nb\}R
\end{array}
$$

Now, the first principal of the first session has finished his part of the protocol, but the second one hasn't yet started. This is a correspondence flaw between principals, indicated by the last clausal simplification ($274 = Clausal\ Simpl(59, 274)$). The total time of execution is less than 2 minutes.

We are now moving forward to replace the definition of severals sessions by the definition of only one session, in which we test whether there is a flaw or not, and adding only principals who always use the same nonces to help the intruder.

### 3.4   Other Protocols Already Studied

The study of the protocols given in Figure 6 is straightforward, and is done in an automatic way similar to the one used for the Otway-Rees protocol.

We point out that, in all protocols but one studied up to now, we have, every time, obtained an attack when there is one, and we have not found any attack when no attack was reported in the literature. The only exception is the Yahalom's protocol, for which an error is reported in [4], but the error report seems to have an error itself.
All those results have been obtained with a PC under Linux, with a processor at 800MHz.

## Conclusion

We have designed and implemented in Casrul a compiler of cryptographic protocols, transforming a general specification into a set of rewrite rules. The user can specify some strategies for the verification of the protocol, such as the number of parallel sessions, the initial knowledge and the general behavior of the intruder,

13

| Protocol | User Time | Kind of Flaw |
|---|---|---|
| Secure RPC | 41s | Compromised Key |
| Encrypted Key Exchange | 110s | Correspondence Between Principals |
| NSPK Exchange | 6s | Correspondence Between Principals |
| TMN | 18s | Correspondence Between Principals |
| RSA | 0.5s | Secrecy Flaw |
| Woo-Lam ($\pi$) | 22s | Correspondence Between Principals |
| Woo-Lam (3) | 4s | Correspondence Between Principals |
| Woo-Lam Mutual Authentification | 340s | Correspondence Between Principals |
| SPLICE/AS | 59s | Correspondence Between Principals |
| Neumann-Stubblebine (Part 1) | 2s | Correspondence Between Principals |
| Kao Chow | 24s | Compromised Key |
| Otway-Rees | 1.5s | Secrecy |

**Fig. 6.** Results obtained with daTac for several cryptographic protocols.

and the kind of attack to look for.

The transformation to rewrite rules is fully automatic and high level enough to permit further extensions or case specific extensions. For example, one can model specific key properties such as key commutativity in the RSA protocol.

The protocol model generated is general enough to be used for various verification methods: model checking, proof by induction, narrowing, ... In our case, we have used narrowing with the theorem prover daTac. The AC properties proposed by this system permit us to handle general rewrite rules, simplifying the translation from the Casrul output to the daTac input by Casdat.

We're now moving on to add expressiveness to the CAS syntax, as we just did in the case of $xor-$encryption. Another direction is to express parallel sessions (with same nonces) so that it allows the use of an unbounded number of such sessions. We also plan to work on the study of an unbounded number of sequential sessions, which should be useful in the study of One Time Password protocols. In this case, each session would have its own nonces. But, because of undecidability results [15], we would have to restrain our model in order to keep implementability.

# References

1. R. Amadio and D Lugiez. On the Reachability Problem in Cryptographic Protocols. In *RR-INRIA — 3915*. Marseille (France).
2. D. Basin. Lazy Infinite-State Analysis of Security Protocols. In R. Baumgart, editor, *Secure Networking — CQRE'99*, volume 1740 of *Lecture Notes in Computer Science*, pages 30–42. Springer-Verlag, Düsseldorf (Germany), 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. 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.

8. A. Durante, R. Focardi, and R. Gorrieri. A Compiler for Analysing Cryptographic Protocols Using Non-Interference. *ACM Transactions on Software Engineering and Methodology*, 9(4):489–530, 2000.

9. T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In D. A. McAllester, editor, *17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290, Pittsburgh (PA, USA), 2000. Springer-Verlag.

10. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Computer Science*, pages 131–160, St Gilles (Réunion, France), November 2000. Springer-Verlag.

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

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

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

14. J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Mur$\phi$. In *IEEE Symposium on Security and Privacy*, pages 141–154. IEEE Computer Society, 1997.

15. J. Mitchell A. Scedrov N. Durgin, P. Lincoln. Undecidability of Bounded Security Protocols. In *Workshop on Formalk Methods and Security Protocols*. Trento, Italy (part of FLOC'99).

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

17. M. Rusinowitch and L. Vigneron. Automated Deduction with Associative-Commutative Operators. *Applicable Algebra in Engineering, Communication and Computation*, 6(1):23–56, January 1995.

18. B. Schneier. *Applied Cryptography*. John Wiley, 1996.

19. C. Weidenbach. Towards an Automatic Analysis of Security Protocols. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 378–382, Trento (Italy), 1999. Springer-Verlag.