Strategy for Verifying Security Protocols with Unbounded Message Size

Y. Chevalier and L. Vigneron *

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

Abstract

We present a system for automatically verifying cryptographic protocols. This system manages the knowledge of principals and checks if the protocol is runnable. In this case, it outputs a set of rewrite rules describing the protocol itself, the strategy of an intruder, and the goal to achieve. The protocol specification language permits to express commonly used descriptions of properties (authentication, short term secrecy, and so on) as well as complex data structures such as tables and hash functions. The generated rewrite rules can be used for detecting flaws with various systems: theorem proving in first-order logic, on-the-fly model-checking, or SAT-based state exploration. These three techniques are being experimented in the European Union project AVISPA.
The aim of this paper is to describe the analysis process. First, we describe the major steps of the compilation process of a protocol description by our tool Casrul. Then, we describe the behavior of the intruder defined for the analysis. Our intruder is based on a lazy strategy, and is implemented as rewrite rules for the theorem prover daTac. Another advantage of our model is that it permits to handle parallel executions of the protocol and composition of keys. And for sake of completeness, it is possible, using Casrul, to either specify an unbounded number of executions or an unbounded message size.
The combination of Casrul and daTac has permitted successful studying of various protocols, such as NSPK, EKE, RSA, Neumann-Stubblebine, Kao-Chow and Otway-Rees. We detail some of these examples in this paper. We are now studying the SET protocol and have already very encouraging results.

Keywords: Security protocols, Verification, Flaw detection, Intruder model, Automatic strategies.

*This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-39252 AVISPA project.

1 Introduction

The verification of cryptographic protocols has been intensively studied these last years. A lot of methods have been defined for analyzing particular protocols [32, 6, 14, 35, 42, 23]. Some tools, such as Casper [26], CVS [19] and CAPSL [15], 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 different verification systems.

Our work is in this last line. We have designed a protocols compiler, Casrul [25], that translates a cryptographic protocol specification into a set of rewrite rules. This translation step permits, through static analysis of the protocol, to rule out many errors.
The output of our compiler can be used to get a representation of protocols in various systems:

- As Horn clauses, it can be used either by theorem provers in first-order logic, or as a Prolog program.
- As rewrite rules, it can be used by inductive theorem provers, or as an input for a rewrite systems compiler, such as ELAN.
- As propositional formulas, the search for an attack can be seen as a planning problem, which can, after translation into SAT formulas, be solved by SAT solvers such as Chaff [34, 3].

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 state models (by not limiting the size of messages), and also to guarantee the freshness of the randomly generated information, such as nonces or keys [25]. Note that there was a first approach with narrowing by Meadows in [27], where the narrowing rules are restricted to symbolic encryption equations (see also [30]); transitions are handled by a Prolog-like backward search from a goal stating insecurity.

The main objective of this paper is, after giving a general presentation of Casrul in Section 2, to present in Section 3 an innovative model of the Intruder behavior, based on the definition of a lazy model. This lazy approach, briefly described in [8], is completely different and much more efficient than the model of the Intruder presented in [25]. It permits to handle untyped variables, and hence messages of unbounded size. In this setting, and when the number of principals is finite, the analysis terminates. This strategy has first been implemented in daTac [9], and has also been independently developed later by other authors [12, 31]. In Section 4, 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, with the timings. In Section 5, we compare the

Casrul compiler with the other compilers available. We also compare our analysis strategy with other tools.

2 Input Protocols

We present in this section the syntax used for describing security protocols, illustrated in Figure 1. This syntax has been partially detailed in [25], and is close to one of CAPSL [29] or Casper [26] 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 [40].
In the following, we present the features added in the syntax for a more powerful expressiveness. We also present some algorithms for verifying the correctness and run-ability of a 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 [25], we have proved that this compilation defines a non-ambiguous operational semantics for protocols and Intruder behavior.

The information given for describing a protocol can be decomposed into two parts: the description of the protocol itself, and the execution environment. This last part includes the legitimate participants and the abilities given to an intruder.

2.1 Protocol Information

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

2.1.1 Identifiers

This section contains the declaration of all the identifiers used in the protocol messages. This includes principals (users), keys (symmetric, public/private, table), piece of text or numbers, hash functions. Some of these 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 four kinds of supported encryption algorithms, the last two being new features, added in the last version of Casrul:

- A key K is an asymmetric key if the key K^{-1} permitting to decrypt a cipher $\{M\}_K$ encrypted by K *cannot* be easily derived from K. The keys K and K^{-1} have to be created at the same time. One is made publicly available, the *public key*, and the other shall be kept secret and is called the *private key*. In our system, public keys are not accessible to anyone by default, since the goal of a protocol may be to ensure that each participant associates a public key to the right person (*i.e.* the possessor of the private key).

[1] http://www.loria.fr/equipes/cassis/softwares/casrul/

Protocol `WLMA`;
Identifiers
A, B, S : *User*;
Na, Nb : *Number*;
Kab, Kas, Kbs : *Symmetric_key*;
Knowledge
A : B, S, Kas;
B : S, Kbs;
S : A, B, Kas, Kbs;
Messages
1. $A \rightarrow B$: A, Na
2. $B \rightarrow A$: B, Nb
3. $A \rightarrow B$: $\{A, B, Na, Nb\}Kas$
4. $B \rightarrow S$: $\{A, B, Na, Nb\}Kas$, $\{A, B, Na, Nb\}Kbs$
5. $S \rightarrow B$: $\{B, Na, Nb, Kab\}Kas$, $\{A, Na, Nb, Kab\}Kbs$
6. $B \rightarrow A$: $\{B, Na, Nb, Kab\}Kas, \{Na, Nb\}Kab$
7. $A \rightarrow B$: $\{Nb\}Kab$

Role
$A[A : b; B : I; S : se; Kas : kbs]$,
$B[B : b; S : se; Kbs : kbs]$;
Parallel
$S[A : I; B : b; S : se; Kas : kis; Kbs : kbs]$,
$S[A : b; B : I; S : se; Kas : kbs; Kbs : kis]$;
Secret kbs;
Intruder *Divert*, *Impersonate*;
Intruder_knowledge b, se, kis;
Goal B *authenticates* S *on* Kab;

Figure 1: Woo and Lam Mutual Authentication Protocol.

- A key K is a *symmetric key* if the key K^{-1} permitting to decrypt a cipher encrypted by K is K itself, or a key that can easily be derived from K.
- 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.
- A *function* f is a one-way, collision-free, hash function algorithm. Thus, for a message M and a *function* f, $f(M)$ is the hash of M calculated by the algorithm f.

2.1.2 Initial Knowledge

For defining the initial state of a protocol, we have to list the 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 (for example, Na, Nb and Kab in Figure 1). Note that this is possible to give *messages* in the initial knowledge of a principal as long as all identifiers appearing in these messages are defined.

2.1.3 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 for encoding is very classical: $\{M\}_K$ means the message M encoded by the key K. The encoding is supposed to be a public/private key encoding if K is a *public* or *private* key, or an element of a table. If K is a *symmetric* key (or a *compound message*, as used in SSL for instance), it is assumed that a symmetric encryption/decryption algorithm is used to encode the ciphers. 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 block 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 that the protocol is correct and runnable by verifying the evolution of the knowledge of each principal.

2.2 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:

- 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 with respect to the initial specification if each principal can compose the messages it is supposed to send. For some messages, principals will use parts of the received messages. Thus, 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 right key).

The function *compose* defined in Figure 2 describes the composition of a message M by a principal U at the step i of the protocol. The knowledge of U before running this function 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.

$$
\begin{array}{rcll}
compose(U,M,i) & = & t & \text{if } M \text{ is known by } U \text{ and named } t \quad (1)\\
compose(U,\langle M_1,M_2\rangle,i) & = & \langle compose(U,M_1,i), compose(U,M_2,i)\rangle & (2)\\
compose(U,(M_1)\mathsf{xor}(M_2),i) & = & (compose(U,M_1,i))\mathsf{xor}(compose(U,M_2,i)) & (3)\\
compose(U,\{M\}_K,i) & = & \{compose(U,M,i)\}_{compose(U,K,i)} & (4)\\
compose(U,T[A],i) & = & compose(U,T,i)[compose(U,A,i)] & (5)\\
compose(U,T[A]^{-1},i) & = & compose(U,T,i)[compose(U,A,i)]^{-1} & (6)\\
compose(U,f(M),i) & = & compose(U,f,i)(compose(U,M,i)) & (7)\\
compose(U,M,i) & = & fresh(M) & \text{if } M \text{ is a fresh identifier} \quad (8)\\
compose(U,M,i) & = & \mathtt{Fail} & \text{else} \quad (9)
\end{array}
$$

Figure 2: Verification that a message can be composed.

Note that when a fresh identifier (key, nonce, ...) is encountered for the first time, a unique new term is automatically generated.
As the message M has to be sent by U at step i, any problem will generate a failure in this function. This case implies that the principal does not have enough knowledge to compose the message to send, and hence the protocol is not *runnable*. The compilation process will abort and an error message indicating which piece of knowledge could not be composed is output.

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 knows, it has to check this 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 expects.
These verifications are pre-computed during the compilation process by the function *expect* defined in Figure 3. This function describes the behavior of a principal U that tries to give an as accurate as possible value to every part of a message it will receive. Each unknown cipher is replaced by a new variable $x_{U,M}$ that can be seen as a new name.
Rules 12 and 13 mean that U needs to know one of the arguments of the *Xor* operator to get the other one and study it. Rules 14, 15 and 16 describe that U has to be able to compose the key decoding the analyzed cipher, i.e. the inverse key of the one coding the message, for studying the contents of the message M.
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 or a compound term.

$$\begin{array}{rcll}
expect(U,M,i) & = & compose(U,M,i) & \text{if no } \mathtt{Fail} \ (10) \\
expect(U,\langle M_1,M_2\rangle,i) & = & \langle expect(U,M_1,i), expect(U,M_2,i)\rangle & (11) \\
expect(U,(M_1)\mathsf{xor}(M_2),i) & = & (compose(U,M_1,i))\mathsf{xor}(expect(U,M_2,i)) & \text{if no } \mathtt{Fail} \ (12) \\
expect(U,(M_1)\mathsf{xor}(M_2),i) & = & (expect(U,M_1,i))\mathsf{xor}(compose(U,M_2,i)) & \text{if no } \mathtt{Fail} \ (13) \\
expect(U,\{M\}_K,i) & = & \{expect(U,M,i)\}_{compose(U,K^{-1},i)^{-1}} & \text{if no } \mathtt{Fail} \ (14) \\
expect(U,\{M\}_{K^{-1}},i) & = & \{expect(U,M,i)\}_{compose(U,K,i)^{-1}} & \text{if no } \mathtt{Fail} \ (15) \\
expect(U,\{M\}_{SK},i) & = & \{expect(U,M,i)\}_{compose(U,SK,i)} & \text{if no } \mathtt{Fail} \ (16) \\
expect(U,M,i) & = & x_{U,M} & \text{else} \ (17)
\end{array}$$

Figure 3: Verification of a received message.

These algorithms are implemented in Casrul. This compiler can therefore generate rewrite rules modeling the behavior of principals: principals wait until a message is received, and immediately send a response. This can be summarized by the following kind of rule:

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

Considering the Woo and Lam protocol given in Figure 1, for each role, the transitions are modeled by the following rewrite rules, where x_A, x_{Na}, x_{Kbs}, ... are names of known knowledge, and x_1, x_2, ... are new names (i.e. variables) different for each role and representing previously unknown pieces of messages.

- Role A: (initial knowledge: x_A, x_B, x_S, x_{Kas})
  - Composed for x_B:
    $\Rightarrow x_A, fresh(Na)$
    A generates a new nonce and will name it x_{Na}.
  - Expected from x_B, composed for x_B:
    $x_B, x_1 \Rightarrow \{x_A, x_B, x_{Na}, x_1\}_{x_{Kas}}$
    In this case, A does not know the second information sent by B and names it x_1. But since this has to be a nonce, later it will name it x_{Nb}.
  - Expected from x_B, composed for x_B:
    $\{x_B, x_{Na}, x_{Nb}, x_2\}_{x_{Kas}}, \{x_{Na}, x_{Nb}\}_{x_2} \Rightarrow \{x_{Nb}\}_{x_2}$
    A is able to find x_2 in the first part of the received message; therefore it is able to check the composition of the second part.
- Role B: (initial knowledge: x_B, x_S, x_{Kbs})
  - Expected from x_1, composed for x_1:
    $x_1, x_2 \Rightarrow x_B, fresh(Nb)$

- Expected from x_A, composed for x_S:
  $x_3 \Rightarrow x_3, \{x_A, x_B, x_{Na}, x_{Nb}\}_{x_{Kbs}}$
  B cannot decrypt the received message ($\{A, B, Na, Nb\}_{Kas}$) and gives it the name x_3.
- Expected from x_S, composed for x_A:
  $x_4, \{x_A, x_{Na}, x_{Nb}, x_5\}_{x_{Kbs}} \Rightarrow x_4, \{x_{Na}, x_{Nb}\}_{x_5}$
- Expected from x_A:
  $\{x_{Nb}\}_{x_{Kab}} \Rightarrow$

- Role S: (initial knowledge: x_S, x_A, x_B, x_{Kas}, x_{Kbs})
  - Expected from x_B, composed for x_B:
    $\{x_A, x_B, x_1, x_2\}_{x_{Kas}}, \{x_A, x_B, x_1, x_2\}_{x_{Kbs}}$
    $\Rightarrow \{x_A, x_1, x_2, \mathit{fresh}(Kab)\}_{x_{Kas}}, \{x_B, x_1, x_2, \mathit{fresh}(Kab)\}_{x_{Kbs}}$
    S puts the same fresh key Kab in each part of the composed message.

2.3 Execution Environment

Verifying a protocol consists in trying to simulate what an intruder could do for disturbing the run of a protocol, without some participants noticing. In the previous section, we have defined how generic honest participants of the protocol behave according to the message sequence. However, trying all the possible instantiations of a protocol does not terminate since there are infinitely many.
In this section, we describe how to specify an environment of execution for the protocol. An initial state is given, representing the principals who may run the protocol, together with the roles they might assume. This environment permits to ensure termination of the verification, and also to define this initial state. It is composed using either *Roles* or *Parallel roles* declarations. We also describe how to specify the abilities and knowledge of an attacker, together with the possible goals of this attacker.

2.3.1 Roles

This field describes the possible instances of roles taken by principals in the protocol. Formally, roles such as A, B,... are instantiated by principals. One role may be instantiated zero, one or more times. This is possible to define independently the participants, thus permitting a large flexibility in the definition of the initial state of the protocol.
For example, in the protocol of Figure 1, two roles are defined: b can play the role A with the Intruder I; b can also play the role B.

2.3.2 Parallel Roles

This field is used to specify instances of roles that can be run an unbounded number of times in parallel. As is the case for Roles declaration, it is only possible to

specify a finite number of *different* instances. It corresponds to a weakening of a role definition, because we do not allow those instances to create different nonces. It is used in conjunction with a field `Secret`, that defines instances the Intruder should never be able to know. These instances permit to reduce the number of rules generated by Casrul. A secrecy goal is generated for each of those secret instances. The parallel roles will be played by honest users, but who will be manipulated by the Intruder.
For example, in the protocol of Figure 1, in parallel to the official roles, the Intruder can use a server with whom he can either pretend to be b and play role B, or play role B with its real identity.

2.3.3 Intruder

The `Intruder` field describes which strategies the Intruder can use, among three possibilities: passive `eaves_dropping`, `divert` and `impersonate`. These properties depend on the assumptions made on the execution environment of the protocol. If nothing is specified, this means that we want a simulation of the protocol in a safe network.
When communicating through an unsafe media, one should assume an Intruder is present. Depending on the network, he can have the ability `divert`, `eaves_dropping`, `impersonate` or any combination of these. If `divert` is selected, he can remove messages from the network; if `eaves_dropping` is selected, he can just record the contents of messages exchanged.
The Intruder is then able to reconstruct terms as he wishes, using all the information he got. He can send arbitrary messages in his own name. If moreover `impersonate` is selected, he can also send messages in the name of another principal.

In the description of the Intruder model (Section 3), we will focus on the case where he may divert messages and impersonate principals.

2.3.4 Intruder Knowledge

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

2.3.5 Goal

This field gives the kind of flaw we want to detect. There are several possibilities, but the two main ones are `authenticates` and `Secrecy_of`.
– *Secrecy* means that some secret information (e.g. a key or a number) exchanged during the protocol is kept secret.

– An authentication goal is defined for instance by

$$B \texttt{ authenticates } S \texttt{ on } Kab$$

which means that if a principal b playing the role B ends its part of the protocol and believes it has interacted with a principal se playing the role S, and if it believes it has received Kab sent by se, then, at some point, the principal se must have sent Kab to b.
Note that the authentication goal relies on the freshness of information in the system of rewrite rules. For example, in the system devised by Blanchet [5], nonces are not guaranteed to be different from one session of the protocol to another, and in this abstraction, fake authentication attacks can be found on all protocols.
– A last goal has been introduced in order to automatically handle the case of some compromised secrets: the `Short_term_secrecy` goal. In this case, one can define identifiers that should remain secret in a part of the protocol (usually during the session instance where they have been created). Then they are released, i.e. they are added to the knowledge of the Intruder.

3 Intruder's Model

One of the biggest problem in the area of cryptographic protocols verification is the definition of the Intruder. The most referred model is the one defined by *Dolev and Yao* in [17]. It says that the Intruder controls entirely the communication network. This means that he can intercept, record, modify, compose, send, crypt and decrypt (if he knows the appropriate key) each message. He can also send messages in the name of another principal.
However, the Dolev-Yao's model of an Intruder 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, and so on.

In some approaches people try to bound the size of the messages, but these bounds are valid only 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 w.r.t. the initial state. Thus, those bounds are not relevant in our approach, and this has led us to bring a new model of the Intruder.

A common approach to deal with infinite-space problem is to use a lazy exploration of the state space while analyzing the protocol by model-checking. This approach was proposed for example in [4]. In a totally different way, our work can be connected to this since we have developed a lazy version of Dolev-Yao's Intruder: we replace the terms building step of the Intruder by a step in which, at the same time, the Intruder analyzes his knowledge and tests if he can build a term matching the message awaited by a principal; the pattern of the awaited message is given by the principal, instead of being blindly composed by the Intruder. This defines our

model as a lazy one, where a symbolic analysis is performed on *classes* of possible executions, those classes being lazily constructed. More information on this is given in [10].
This strategy may look similar to the one described in [37] (Chapter 15), but our lazy model is applied dynamically during the execution of the protocol, while Roscoe's model consists in looking for the messages that can be composed by the Intruder before the execution of the protocol. The messages are prepared statically in advance, and some type information permits to bound the total size of the system. This is a strong restriction to the Dolev-Yao model. One advantage of our method is that we can find some type flaws (in the Otway-Rees protocol, for instance) that cannot be found when the size of messages is bounded by typing.
In the first version of Casrul [25], we used to have a static method similar to Roscoe's, where we were generating many rules in which the Intruder was impersonating the principals. This method was found unsuitable for complex protocols, where a given rule can be applied in many different, yet often equivalent, ways.

In the following, we first 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 4.2) 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 declared in the `Role` and `Parallel` fields;
- 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 building a couple (i.e. a concatenation) of messages;
- CRYPT, SCRYPT and XCRYPT operators stand respectively for public or private key encryption, symmetric key encryption and *Xor*-encryption;
- TABLE(t_1, t_2) is valid if t_1 is the name of a table, and t_2 the name of a principal. In this case, TABLE(t_1, t_2) stands for the public key of t_2 registered in table t_1;
- FUNC(t_1, t_2) is valid if t_1 is a function symbol and t_2 is a message. In this case, FUNC(t_1, t_2) is the hash of t_2 computed with the algorithm t_1.

To describe our lazy version of Dolev and Yao's intruder using a set of rewrite rules, we also use other operators whose meaning should be clear from the name. For instance, *Comp* is used for the composition of a message; note that the "`.`" operator is not a list constructor, but an associative and commutative (AC) operator. These rules originate from the implementation of the lazy intruder in daTac.

3.1 Test of 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 4 tries to reduce the expression $Comp(t)$ *from* C ; Id , building a substitution τ. In this expression, Id stands for the identity substitution, and $Comp(t)$ *from* C is a constraint for the Intruder meaning that the term t has to be composable from the knowledge list C.
This test is a constraint solving algorithm; it cannot be compared with functions *compose* and *expect* described in Section 2.2, which are defined for normal principals: *compose* is used for checking that the protocol is runnable; *expect* is used for verifying the contents of received messages.

$$
\begin{array}{rcll}
Comp(t).T \textit{ from } t.C \;;\; \tau & \rightarrow & T \textit{ from } t.C \;;\; \tau & (18) \\
Comp(r).T \textit{ from } s.C \;;\; \tau & \rightarrow & T\sigma \textit{ from } s\sigma.C\sigma \;;\; \tau\sigma \quad \text{if } r\sigma = s\sigma & (19) \\
Comp(\textsc{c}(t_1,t_2)).T \textit{ from } C \;;\; \tau & \rightarrow & Comp(t_1).Comp(t_2).T \textit{ from } C \;;\; \tau & (20) \\
Comp(\textsc{crypt}(t_1,t_2)).T \textit{ from } C \;;\; \tau & \rightarrow & Comp(t_1).Comp(t_2).T \textit{ from } C \;;\; \tau & (21) \\
Comp(\textsc{scrypt}(t_1,t_2)).T \textit{ from } C \;;\; \tau & \rightarrow & Comp(t_1).Comp(t_2).T \textit{ from } C \;;\; \tau & (22) \\
Comp(\textsc{xcrypt}(t_1,t_2)).T \textit{ from } C \;;\; \tau & \rightarrow & Comp(t_1).Comp(t_2).T \textit{ from } C \;;\; \tau & (23) \\
Comp(\textsc{table}(t_1,t_2)).T \textit{ from } C \;;\; \tau & \rightarrow & Comp(t_1).Comp(t_2).T \textit{ from } C \;;\; \tau & (24) \\
Comp(\textsc{table}(t_1,t_2)^{-1}).T \textit{ from } C \;;\; \tau & \rightarrow & Comp(t_1).Comp(t_2).T \textit{ from } C \;;\; \tau & (25) \\
Comp(\textsc{func}(t_1,t_2)).T \textit{ from } C \;;\; \tau & \rightarrow & Comp(t_1).Comp(t_2).T \textit{ from } C \;;\; \tau & (26)
\end{array}
$$

Figure 4: System testing if a term may be composed from some given knowledge.

The system of Figure 4, 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. Further investigation is needed to handle the case of the Xor-encryption. The rule (23) cannot handle, for example, that:

$$(x \oplus y) \oplus (y \oplus z) = x \oplus z$$

The effectiveness of this system heavily relies on the fact that we *do not* use the rule (19) when the term r is a variable, thereby reducing the test of composability of a term to the test of composability of some of its variables, which can then be instantiated later. Moreover, termination is ensured by restricting the applications of rules (20-26) to the cases where they apply on terms which are not variables. This last restriction is mandatory to ensure termination, since the Intruder would otherwise test terms of unbounded depth.

Theorem of Completeness. For any set of constraints, if there exists a substitution that satisfies all its constraints, our strategy permits to transform this set into either an empty set, or a set containing only simple constraints, i.e. of the form $Comp(x_1).\ldots.Comp(x_n)$ *from* C where the x_i are variables (such a constraints are solved by replacing x_i by anything).

This completeness result can be proved in two steps. First, one shows that forbidding unification between a variable and another term is complete when one considers satisfiability of the system. The main point of the proof here is that if a variable appears in the knowledge set I of the Intruder, it also appears in a constraint for a knowledge set I', with $I' \subset I$. This constraint can be satisfied by I', hence it is possible to replace a unification between a term t in a constraint and a variable x in the knowledge set by another derivation leading to the satisfaction of the constraint.
The second step is to show that the restriction on the application of rules (20-26) is complete, which can be easily done once one can ensure there is no unification between a variable and another term. Further details are given in [10]. Note also that the proof of completeness and correctness of a system derived from the one presented here was given in [38] as a NP-completeness result for finding attacks in the case of a finite number of principals.

For example, from the Intruder's knowledge $\textsc{mr}(b).\textsc{scrypt}(sk(kbs), \textsc{nonce}(Nb))$, we may test if a term matching $\textsc{crypt}(\textsc{c}(\textsc{mr}(b), x_1), \textsc{scrypt}(sk(kbs), x_2))$ can be built:

$$
\begin{array}{ll}
\multicolumn{2}{l}{Comp(\textsc{crypt}(\textsc{c}(\textsc{mr}(b), x_1), \textsc{scrypt}(sk(kbs), x_2)))} \\
\multicolumn{2}{l}{\qquad \textit{from } \textsc{mr}(b).\textsc{scrypt}(sk(kbs), \textsc{nonce}(Nb)) \ ; \ Id} \\
\rightarrow^{(21)} & Comp(\textsc{c}(\textsc{mr}(b), x_1)).Comp(\textsc{scrypt}(sk(kbs), x_2)) \\
 & \qquad \textit{from } \textsc{mr}(b).\textsc{scrypt}(sk(kbs), \textsc{nonce}(Nb)) \ ; \ Id \\
\rightarrow^{(20)} & Comp(\textsc{mr}(b)).Comp(x_1).Comp(\textsc{scrypt}(sk(kbs), x_2)) \\
 & \qquad \textit{from } \textsc{mr}(b).\textsc{scrypt}(sk(kbs), \textsc{nonce}(Nb)) \ ; \ Id \\
\rightarrow^{(18)} & Comp(x_1).Comp(\textsc{scrypt}(sk(kbs), x_2)) \\
 & \qquad \textit{from } \textsc{mr}(b).\textsc{scrypt}(sk(kbs), \textsc{nonce}(Nb)) \ ; \ Id \\
\rightarrow^{(19)} & Comp(x_1) \\
 & \qquad \textit{from } \textsc{mr}(b).\textsc{scrypt}(sk(kbs), \textsc{nonce}(Nb)) \ ; \ \sigma
\end{array}
$$

The test is successful, generating the substitution $\sigma : x_2 \leftarrow \textsc{nonce}(Nb)$ 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.

3.2 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 looks like it has been sent by another principal (chosen by the Intruder). We define

$$C.UFO(\textsc{f}(t).C') \rightarrow C.\textsc{f}(t).UFO(C') \quad (27)$$
$$C.UFO(\textsc{c}(t_1,t_2).C') \rightarrow C.UFO(t_1.t_2.C') \quad (28)$$
$$C.UFO(\textsc{crypt}(t_1,t_2).C') \rightarrow C.\textsc{crypt}(t_1,t_2).UFO(\textsc{test}(\textsc{crypt}(t_1,t_2)).C') \quad (29)$$
$$C.UFO(\textsc{test}(\textsc{crypt}(t_1,t_2)).C') \rightarrow C.UFO(t_2.C')\sigma \quad \textit{if } A({t_1}^{\text{-}1},C,\sigma) \quad (30)$$
$$C.UFO(\textsc{scrypt}(t_1,t_2).C') \rightarrow C.\textsc{scrypt}(t_1,t_2).UFO(\textsc{test}(\textsc{scrypt}(t_1,t_2)).C') \quad (31)$$
$$C.UFO(\textsc{test}(\textsc{scrypt}(t_1,t_2)).C') \rightarrow C.UFO(t_2.C')\sigma \quad \textit{if } A(t_1,C,\sigma) \quad (32)$$
$$C.UFO(\textsc{xcrypt}(t_1,t_2).C') \rightarrow C.\textsc{xcrypt}(t_1,t_2).UFO(\textsc{test}(\textsc{xcrypt}(t_1,t_2)).C') \quad (33)$$
$$C.UFO(\textsc{test}(\textsc{xcrypt}(t_1,t_2)).C') \rightarrow C.UFO(t_2.C')\sigma \quad \textit{if } A(t_1,C,\sigma) \quad (34)$$
$$C.UFO(\textsc{test}(\textsc{xcrypt}(t_1,t_2)).C') \rightarrow C.UFO(t_1.C')\sigma \quad \textit{if} A(t_2,C,\sigma) \quad (35)$$
$$C.UFO(\textsc{table}(t_1,t_2).C') \rightarrow C.\textsc{table}(t_1,t_2).UFO(C')\sigma \quad (36)$$
$$C.UFO(\textsc{table}(t_1,t_2)^{-1}.C') \rightarrow C.\textsc{table}(t_1,t_2)^{-1}.UFO(C')\sigma \quad (37)$$
$$C.UFO(\textsc{func}(t_1,t_2).C') \rightarrow C.\textsc{func}(t_1,t_2).UFO(C')\sigma \quad (38)$$

Figure 5: System Simplifying the Intruder's Knowledge.

a system that keeps in a predicate, *UFO*, the data that are not already treated by the Intruder, and moves the non-decomposable knowledge out of *UFO*. For the decryption of a cipher (but this should also apply to hash functions), we use a predicate (TEST) and a conditional rewrite rule. The resulting system described in Figure 5 only deals with *decomposing* the knowledge of the Intruder, where we are always using, together with the fourth rule, the equality $t^{\text{-}1\text{-}1} = t$.

We note this system $A(t, C, \sigma)$, a predicate that is true whenever the term t can be built from the knowledge C using a substitution σ. The system of Figure 4 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.

3.3 Use of this Model for Flaws Detection

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

1. For each active state s of the protocol, and for each principal p, the Intruder creates a new active state s' where he tries to send a message to the principal p: the principal p brings a pattern m that the Intruder's message should match. At the same time, p gives the pattern of the message t that the Intruder will receive if he succeeds in sending a message;

2. Second, the Intruder analyzes his knowledge and tests if he can compose a

message matching this pattern m;

3. If he can compose a message matching the pattern m, he goes back to step 1 with s' as active state. If not, this state s' fails and is removed from the active states.

3.4 Properties of our Model

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:

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

This is used, for instance in the example of Section 3.1, to prove it is sound to substitute $\text{NONCE}(Nb)$ for x_2.
We also maintain a set of knowledge C representing the Intruder's knowledge evolution whenever he succeeds in sending an appropriate message. We model a protocol step with the rule:

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

Comparing this model to an execution model where an Oracle tells a message (ground term) that is accepted by the principal and where the Intruder has to verify he can send this message, this exhaustive exploration system turns out to be both *sound* (see the semantics given in [25]) and *complete* as long as we consider only a bounded number of principals [10, 38]. The variables here are untyped, thus allowing *messages of unbounded size* and the discovery of *type flaws*.

The *complexity* of our strategy is exponential in the number of roles. For parallel roles, the initial computation is exponential, but once done, the execution depends only on the number of constraints generated.

3.5 Example: Woo-Lam Protocol

The execution of the specification given in Figure 1 gives the following result:

$I \rightarrow b$	$: I, x_3$
$I \rightarrow b$	$: x_5$
$I(se) \rightarrow b$	$: x_4, \{I, x_3, Nb, x_2\}kbs$
$I \rightarrow b$	$: \{Nb\}x_2$

This corresponds to the messages sent by the Intruder, in the run of the protocol where I plays role A and b plays role B, for getting an authentication flaw between b and se. They are messages 1, 3, 5 and 7 of the protocol. Nb is the nonce generated by b.
Variables x_2, x_3, x_4 and x_5 represent parts of messages that are independent of this protocol run. b cannot check their value.

The most tricky message is the third. How can I compose $\{I, x_3, Nb, x_2\}kbs$ without knowing the key kbs? This is a constraint that the Intruder has to solve. Our model has concluded that this constraint is solvable and I can send this message. This conclusion is the result of the study of the Intruder's knowledge, of the protocol messages and of the roles and parallel roles. I cannot compose this part of the message, but he can get one that matches with it:

$A(b) \rightarrow I$	:	b, Na
$I \rightarrow A(b)$	:	I, Nb
$A(b) \rightarrow I$	:	$\{b, I, Na, Nb\}kbs$

$I \rightarrow se$	:	$\{b, I, Na, Nb\}kbs, \{b, I, Na, Nb\}kis$
$se \rightarrow I$	:	$\{I, Na, Nb, Kai\}kbs, \{b, Na, Nb, Kai\}kis$

In the first part, he uses b as player of role A and Nb as its own nonce; in the second part, he uses the server se as a parallel role. The consequence is that the first part of the last message ($\{I, Na, Nb, Kai\}kbs$) matches with the message to build in the main run, $\{I, x_3, Nb, x_2\}kbs$. The result is a flaw because b thinks that the server has generated Kai for the protocol run where it has played role B; and the server se thinks it has generated Kai for b playing role A.

Thanks to our model of the Intruder, the role b as A and the parallel role se as S are not run. This is an important improvement for the efficiency of flaw detection (see Table 1), that is possible because the Intruder knows the nonces Na and Nb.

4 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. Then, with the EKE protocol, we show how we deal with concurrent runs of the protocol. We also list the results obtained for other protocols that can be found in [11].
But first, let us give a short presentation of the prover used for looking for flaws from the rewrite rules generated by Casrul.

4.1 The Prover daTac

For studying the protocols with the rules generated by Casrul, we have used the theorem prover daTac[2], specialized for automated deduction in first-order logic with equality and associative-commutative (AC) 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. A more pertinent use is the possibility we have to express commutative properties of constructors. This enables us, for example, to express the commutativity of encryption in the RSA protocol.

[2] http://www.loria.fr/equipes/cassis/softwares/daTac/

The deduction techniques used by daTac are Resolution and Paramodulation [39]. They are combined with efficient simplification techniques for eliminating redundant information. Another important property is that this theorem prover is refutationally complete. Our model being complete with respect to the Dolev-Yao's model, we are certain to find all expressible flaws.

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

4.2 The Otway-Rees Protocol

The Casrul specification of this well-known protocol is given in Figure 6. To study

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. $A \rightarrow B$: $M, A, B, \{Na, M, A, B\}Kas$
2. $B \rightarrow S$: $M, A, B, \{Na, M, A, B\}Kas,\ \{Nb, M, A, B\}Kbs$
3. $S \rightarrow B$: $M, \{Na, Kab\}Kas, \{Nb, Kab\}Kbs$
4. $B \rightarrow A$: $M, \{Na, Kab\}Kas$
5. $A \rightarrow B$: $\{X\}Kab$

Role
$A[A : a; B : b; S : se; Kas : kas]$,
$B[B : b; S : se; Kbs : kbs]$,
$S[A : a; B : b; S : se; Kas : kas; Kbs : kbs]$;
Intruder *Divert*, *Impersonate*;
Intruder_knowledge a;
Goal *Secrecy_Of* X;

Figure 6: Otway-Rees Protocol.

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:

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

The trace of an execution is quite hard to analyze if one is not familiar with the techniques implemented in daTac, but hopefully, the result is the sequence of derivations leading to the discovery of the flaw *(in 6s)*:

> 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. Only the clauses generated by a resolution step and fully simplified matter.

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:

$a \ \rightarrow \ \_ \quad : M, a, b, \{Na, M, a, b\}kas$

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:

$a \ \rightarrow \ \_ \quad : M, a, b, \{Na, M, a, b\}kas$
$\_ \ \rightarrow \ a \quad : M, \{Na, x5\}kas$
$a \ \rightarrow \ \_ \quad : \{X\}x5$

At this point, we can only say that the Intruder has tried to send to the principal a a message *matching* $M, \{Na, x5\}kas$. He 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:

$a \ \rightarrow \ \_ \quad : M, a, b, \{Na, M, a, b\}kas$
$\_ \ \rightarrow \ a \quad : M, \{Na, M, a, b\}kas$
$a \ \rightarrow \ \_ \quad : \{X\}(M, a, b)$

The Intruder has proved that he can send a term matching the pattern of the awaited message, so we can go on to the next step, the decomposition of the second message sent by a ($66 = Simpl(52, 66)$). But after that, decomposing what he 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.

4.3 The Encrypted Key Exchange *(EKE)* Protocol

We shall now study the EKE protocol, known to have a parallel authentication attack. The Casrul specification of this protocol is given in Figure 7.

Protocol EKE;
Identifiers
A, B : *User*;
Na, Nb : *Number*;
Ka : *Public_key*;
P, R : *Symmetric_key*;
Knowledge
A : B, P;
B : P;
Messages
1. $A \rightarrow B$: $\{Ka\}P$
2. $B \rightarrow A$: $\{\{R\}Ka\}P$
3. $A \rightarrow B$: $\{Na\}R$
4. $B \rightarrow A$: $\{Na, Nb\}R$
5. $A \rightarrow B$: $\{Nb\}R$

Role
$A[A : a; B : b; P : p]$;
Parallel
$B[B : a; P : p; R : re]$;
Secret p, re;
Intruder *Divert*, *Impersonate*;
Intruder_knowledge ;
Goal A *authenticates* B *on* Nb;

Figure 7: Encrypted Key Exchange Protocol.

The trace of the execution does also, in this case, lead to a flaw. This time, this is an authentication flaw:

> Inference steps to generate the empty clause:

87 = Resol(1,85)	87 = Simpl(9,87)	...	87 = Simpl(76,87)
88 = Resol(2,87)	88 = Simpl(9,88)	...	88 = Simpl(31,88)
89 = Resol(81,88)	89 = Simpl(49,89)	...	89 = Simpl(4,89)
90 = Resol(4,89)	90 = Simpl(9,90)	...	90 = Simpl(31,90)
92 = Resol(78,90)	92 = Simpl(27,92)	...	92 = Simpl(70,92)
92 = Clausal Simpl({33},92)			

We can study in deeper details this trace in order to find the scenario of the attack. Since the principal a appears in two instances, we will give, right after its name, a string (*seq* or //) that identifies the principal either as the one defined in the `Role` or in the `Parallel` field. First of all, the principal of the `Role` field starts with sending its first message:

$a(seq) \rightarrow \_$: $\{Ka\}p$

Here, $a(seq)$ has to generate a fresh key Ka. Then, the Intruder tries to send a message to the principal a defined in the `Role` field:

$a(seq) \rightarrow \_$	$: \{Ka\}p$
$\_ \rightarrow a(seq)$	$: \{\{x_1\}Ka\}p$
$a(seq) \rightarrow \_$	$: \{Na\}x_1$

The Intruder now has to prove he could send the message $\{\{x_1\}Ka\}p$. He cannot compose this message using his current knowledge, but he can have a term unifying with this by interacting with $a(//)$. This is what is done in the next resolution:

$a(seq) \rightarrow \_$	$: \{Ka\}p$
$\_ \rightarrow a(//)$	$: \{Ka\}p$
$a(//) \rightarrow \_$	$: \{\{re\}Ka\}p$
$\_ \rightarrow a(seq)$	$: \{\{re\}Ka\}p$
$a(seq) \rightarrow \_$	$: \{Na\}re$

Now, the Intruder can go on like this until he arrives at this point:

$a(seq) \rightarrow \_$	$: \{Ka\}p$
$\_ \rightarrow a(//)$	$: \{Ka\}p$
$a(//) \rightarrow \_$	$: \{\{re\}Ka\}p$
$\_ \rightarrow a(seq)$	$: \{\{re\}Ka\}p$
$a(seq) \rightarrow \_$	$: \{Na\}re$
$\_ \rightarrow a(//)$	$: \{Na\}re$
$a(//) \rightarrow \_$	$: \{Na, Nb\}re$
$\_ \rightarrow a(seq)$	$: \{Na, Nb\}re$
$a(seq) \rightarrow \_$	$: \{Nb\}re$

The principal $a(seq)$ has finished its part of the protocol, and it is possible to see if Nb was a nonce created by a principal b communicating with a. This is not the case here, so we reach an authentication flaw, as indicated by the last clausal simplification ($92 = Clausal\ Simpl(33, 92)$). The total time of execution is a few seconds.

One can note that, in this case, we perform much better than in [8], where we used two instantiations of both roles running concurrently. The time needed to reach this flaw was around 4 minutes, and the number of states explored before reaching the flaw was around 200 (here, only 7!).

4.4 Some of the Other Protocols Already Studied

The study of the protocols given in Table 1 is straightforward, and is done in an automatic way similar to the one used for the two previously detailed examples.
The table shows the difference of timings between the original version of Casrul,

Protocol	Lazy	Lazy //	Kind of Flaw
Andrew Secure RPC	7s	7s	Authenticate Flaw
Encrypted Key Exchange	268s	3s	Authenticate Flaw
Kehne-Schoene-Langendorf	69s	20s	Authenticate Flaw
Kao Chow (unamended)	24s	2s	Authenticate Flaw
Needham Schroeder Conventional Key Protocol	11s	11s	Compromised Key/Authenticate Flaw
Needham Schroeder Public Key Protocol	18s	3s	Authenticate Flaw
Neumann-Stubblebine (initial part)	8s	2s	Authenticate Flaw
Neumann-Stubblebine (repeated part)	34s	4s	Authenticate/Secrecy Flaw
Otway Rees	6s	6s	Authenticate Flaw
RSA protocol	2s	2s	Secrecy Flaw
SPLICE	53s [3]		Authenticate Flaw
Intruder impersonates client		6s	Authenticate Flaw
Intruder impersonates server		26s	Authenticate Flaw
Davis Swick Authentication Protocol	181s	18s	Authenticate Flaw
TMN (compromised key flaw)	67s	67s	Short Term Secrecy Flaw
TMN (authentication flaw)	41s	41s	Authenticate Flaw
Woo-Lam $\pi - (3)$ Protocol	10s	10s	Authenticate Flaw
Woo-Lam π Protocol	59s	1s	Authenticate Flaw
Woo-Lam Mutual Authentication Protocol	512s	30s	Authenticate Flaw

Table 1: Results obtained with Casrul+daTac for several cryptographic protocols.

with the lazy model of the Intruder, and the new one that permits in addition to use parallel roles. All those results have been obtained with a PC under Linux (Pentium 3, 800 MHz, 128 Mb RAM).

All the flaws have been found *automatically* by trying several roles and/or parallel roles; the knowledge of existing flaws has sometimes guided us for limiting those roles, for avoiding useless computations.

We point out that, in all the protocols studied up to now, we have, every time, obtained an attack when there is one, and we have not found any attack when

[3] In this case, both errors are found. We give the time for finding the first flaw (Intruder impersonates the client).

no attack was reported in the literature. One shall also note that the number of explored clauses, using a breadth-first search strategy, is always smaller than a few hundreds. This demonstrates that our lazy strategy for the Intruder, represented by the rewrite rules produced by Casrul, can be turned into a time and space efficient procedure, independently of the tool used afterwards for finding flaws. This result is obtained *without limiting the number of possible messages*, contrarily to many other tools. This is due to the fact that we do not type informations, i.e. we use a pure symbolic calculus.

Our objective is to continue to improve our verification technique, and to study more and more protocols. But we are also working on the positive verification of protocols. For instance, we have already studied some parts of the SET protocol of VISA and Mastercard, and we are currently studying other parts of this protocol. Since the new version of Casrul permits it, we are also studying protocols that use composed keys, such as SSL (Secure Sockets Layer) [22].

5 Discussion and Related Works

In this section, we summarize the results presented in this paper, compare Casrul with other protocol compilers, and compare our technique with other protocol analysis methods.

5.1 Summary of our Results

We believe that a strong advantage of our method (shared with Casper and CAPSL) 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. Protocols specification are usually given together with the knowledge of participants. Thus, the only part of the verification process where a user of Casrul has to be imaginative is the description of the environment of execution. The Intruder can always be given the strongest abilities `divert` and `impersonate`. And it is now possible, after a work by M. Bouallagui and J. Himanshu [7], to only specify the number of participants instead of their instances. This permits to incrementally search for flaws in the protocol by increasing the number of participants, thereby reducing user's intervention to the important part of the protocol, that is the specification of role instances.

Another important advantage of our method is the ability to handle infinite state models, and to be closer to the original Dolev-Yao model [17] because of our dynamic lazy intruder model.

A limitation of our method is that two fresh information are always different. This means that we do not consider the case where two nonces, for example, are equal by accident. Another limitation is that we cannot consider Diffie-Hellman protocols because we cannot deal with the exponential yet. Otherwise, any protocol that can be expressed in our specification can be studied by our method.

In the following, we summarize the results presented in this paper, depending on the limitation or not of the number of sessions.

5.1.1 Unbounded Number of Sessions

For studying the case of an unbounded number of runs of a protocol, we bound both the size of messages and the number of different nonces in order to ensure termination. The result of the analysis (that uses parallel roles) is expressed as rewrite rules on the knowledge of the Intruder. Thus, no flaw is searched on the principals which can run an unbounded number of concurrent executions. This enables us to study authentication flaws when an unbounded number of concurrent executions are involved in a terminating system. This method is very efficient and complete: if a flaw exists, it will find it; if it terminates without finding any flaw, the protocol is correct. The main drawback is that it is not sound: the abstraction on nonces implies that the attacks found may be fake.

5.1.2 Bounded Number of Sessions

When considering a bounded number of runs of a protocol, we do not need to bound the size of messages. In that case, our method terminates, is sound and is complete for finding flaws.

5.2 Other Protocols Compilers

Let us compare Casrul with the most well-known three similar tools. Casper [26] and CVS [19] are compilers from protocols descriptions to process algebra (CSP for Casper, and SPA for CVS). Both have been applied to a large number of protocols. The Casper approach is oriented towards finite-state verification by model-checking with FDR [36]. The syntax we use for Casrul is similar to the Casper syntax for protocols description. However, our verification techniques, based on theorem proving methods, relax many of the strong assumptions for bounding the information (to get a finite state model) in model checking. For instance, our technique based on narrowing ensures directly that all randomly generated nonces are pairwise different. This guarantees the freshness of information over different executions.
Casper and CVS are similar at the specification level, but CVS has been developed to support the so-called Non-Interference approach [21] to protocol analysis. This approach requires the tedious extra-work of analyzing interference traces in order to check whether they are real flaws. Our verification technique is also based on analyzing traces, but it captures automatically the traces corresponding to attacks.

CAPSL [29] is a specification language for authentication protocols in the flavor of Casper's input. There exists a compiler from CAPSL to an intermediate formalism, CIL, which may be converted to an input for automated verification tools such as Maude, PVS, NRL [28]. A CIL basically contains a set of rules expressing the state transitions of a protocol. Initially, every protocol rule from a standard notation gives rise to two transition rules, one for the sender and one for the receiver. The

transition rules can be executed by multiset and standard pattern matching. 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 the initial CIL which has two rules. More recently, an optimized version of CIL has been defined [16]: it generates one rule per protocol rule, as we do.
Another difference between Casrul and CAPSL is that it is not possible to define an initial state in CAPSL. Thus, back-ends of the CAPSL parser need to define this information separately. Note this is not a limitation for Casrul users, since if initial states currently have to be defined, the receive/send rules of the protocol are independent w.r.t. this initial state. Thus, it can be dropped by tools having no use for it.
Another important difference between the CIL and the rules generated by Casrul is that the evolution of the knowledge of the principals is automatically handled by our system. Hence, we get immediately an optimized version of the rewrite system, which minimizes the number of transitions for verification.
CAPSL takes the advantage on Casrul when it comes to expressiveness. It is already possible to specify choice points and to have a modular specification of protocols whereas this is not implemented yet in Casrul. The extension MuCAPSL of CAPSL also permits to define group protocols.

5.3 Other Protocol Analysis Methods

A lot of other methods have been implemented to verify or find attacks on cryptographic protocols. In this discussion, we focus on automated methods, which can be divided into three main categories.

First, there are methods where an attack is a search starting from an initial state of the protocol. Generally speaking, search for attacks in this setting is most of the time both fast and sound. The drawback of these methods being that no conclusion can be drawn from the absence of attack when starting from an initial state. One can gather in a second category model-checkers aiming at proving correctness of a protocol without starting from an initial state. Undecidability of search for attacks in an unbounded number of executions, where either message size or number of different nonces is not bounded, makes these methods either not sound or not terminating, or both. A third category of methods relies on *abstraction*, and often also on the use of tree automata, to model the behavior of the Intruder and of the principals. These methods permit to prove the correctness of a protocol, but the abstraction done can also lead to the discovery of false flaws.

5.3.1 Model-checking with an Initial State

Lowe uses, together with Casper, the FDR model-checker. The aim of the analysis, in this case, is to prove that the protocol is a refinement of a security property. If a counter-example (*i.e.* an attack) is found, it is given as the result of the computation.

Time comparisons with Casper are difficult, since it is proprietary system. However, one can note that its use by Lowe has permitted the discovery of numerous flaws in the protocols given in the Clark and Jacob library [11, 18]. The main drawback of this tool is that possible type flaws have to be specified by the user.

The Murφ [32] tool integrates a compiler for a high-level protocol specification language relying on guarded commands with a model-checker using aggressive state space reduction through the use of symmetries. While this reduction permits a fast analysis of the protocols, it has been reported that some attacks found using this tool are fake, because they relied on the confusion of nonces created by two different principals. One shall note that the use of symbolic reduction achieves the same state space reduction as the use of symmetries, but in our case the classes of executions are sound and no false attack is found.

A third tool, among those starting from an initial state, is the one developed by Lugiez, Amadio and Vanackère [1]. The lazy intruder strategy is a generalization of their symbolic reduction that permits to handle type flaws and composed keys. It is, to our knowledge, one of the first methods using constraints to analyze cryptographic protocols.

5.3.2 Model-checking without an Initial State

The method proposed by Debbabi et al. [13] is a tool using symbolic constraints but in a somewhat different fashion. The capacities of the Intruder and of the principals are abstracted as inference rules. The inference system is non-terminating, but an algorithm is given permitting to transform this inference system into a terminating one. When this algorithm terminates, it is possible to conclude whether the protocol is flawed. One shall note that in this system, no bound is assumed neither on the message size nor on the number of different nonces. The drawback of this method, however, is that the algorithm used permits only to find special kinds of attacks, *i.e.* those respecting an order on the application of inference rules. Moreover, the method relies on some properties of messages of the protocol to be used, and it is not clear whether it can be extended to generic protocols definitions.

A very similar method is used by Athena [41]. This tool recursively constructs the set of possible executions of the protocol without bound on the number of the executions. Typing permits to bound the size of messages, but the number of different nonces is unbounded, thus implying an infinite state space. The major improvement of this method is that halting conditions can be specified by the user, hence permitting more flexibility in the analysis. However, it seems that the performance of this tool relies on the strategy of the exploration of the state space.

A last tool using *backward search* is described by Blanchet in [5]. The main part of the analysis algorithm builds a finite number of rules describing the possible actions of an intruder between two messages. If this construction terminates, it is possible to analyze an unbounded number of simultaneous runs of the protocol. However, in this case, the number of different nonces is finite. While such construction is very useful for the analysis of secrecy properties of a protocol, it is unsuited for the analysis of authentication, since messages from one execution of any protocol can be replayed later, the nonces being identical. Moreover, the abstraction on nonces leads to an unsound system: some fake secrecy flaws can be found.

5.3.3 Abstraction and Tree Automata

Another possibility for automatic analysis of cryptographic protocols is to use tree automata in conjunction with an abstraction on the protocol. Along this line, one can note the work by Klay and Genet [23] or by Monniaux [33], where the set of messages that the Intruder can compose from a finite set of knowledge is over-approximated by a regular tree language. Such systems can find attacks or verify systems where only a finite number of executions of the protocol is considered. Their drawback is that even in this case, they are not sound: the over-approximation of the knowledge of the Intruder may lead to false attacks. These approaches were extended using an intricate setting in [24] to the case of an unbounded number of parallel executions of the protocol. Logical properties are used to characterize the knowledge of the Intruder, and the states of the automata are logical formulas. One important point is that principals running the protocol in parallel are also used as accomplices of the Intruder. However, further investigation is needed to compare our system with Goubault-Larrecq's.

6 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 simultaneous executions, the initial knowledge, the general behavior of the Intruder, 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, as exemplified in the European Union project AVISPA[4] [2].
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. The timings obtained for verifying protocols could be much better, but using a general theorem prover such as daTac shows how efficient are the rules generated by Casrul. This is confirmed by the large number of protocols that have been verified entirely automatically, the most well-known being listed in Table 1. Recently, we have even found a new attack on the Denning-Sacco symmetric key protocol (see [10]).

[4] http://www.avispa-project.org/

We are currently using all the expressiveness of Casrul for studying large protocols, such as SET and SSL, and the first results are very positive. We also plan to work on the study of an unbounded number of sequential executions, which should be useful in the study of One Time Password protocols, for example. In this case, each session would have its own nonces. But, because of undecidability results [20], we would have to restrain our model in order to keep implementability.

Acknowledgments: We would like to thank the referees for their numerous interesting comments that helped us to improve this article.

References

[1] R. Amadio and D. Lugiez. On the Reachability Problem in Cryptographic Protocols. Technical report, INRIA Research Report 3915, Marseille (France), 2000.

[2] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Vigano, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In *14th International Conference on Computer Aided Verification, CAV'2002*, LNCS 2404, pages 349–353, Copenhaguen (Denmark), 2002. Springer.

[3] A. Armando and L. Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In *22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, Houston, Texas, 2002.

[4] D. Basin. Lazy Infinite-State Analysis of Security Protocols. In *Secure Networking — CQRE'99*, LNCS 1740, pages 30–42, Düsseldorf (Germany), 1999. Springer.

[5] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop*, 2001.

[6] D. Bolignano. Towards the Formal Verification of Electronic Commerce Protocols. In *10th IEEE Computer Security Foundations Workshop*, pages 133–146. IEEE Computer Society, 1997.

[7] M. Bouallagui and J. Himanshu. Automatic Generation of Session Instances. Technical report, LORIA, Nancy (France), 2003.

[8] Y. Chevalier and L. Vigneron. A Tool for Lazy Verification of Security Protocols (short paper). In *Proceedings of ASE-2001: The 16th IEEE Conference on Automated Software Engineering*, San Diego (CA), 2001. IEEE CS Press.

[9] Y. Chevalier and L. Vigneron. Towards Efficient Automated Verification of Security Protocols. In *Proceedings of the Verification Workshop (VERIFY'01) (in connection with IJCAR'01), Università degli studi di Siena, TR DII 08/01*, pages 19–33, 2001.

[10] Y. Chevalier and L. Vigneron. Automated Unbounded Verification of Security Protocols. In *14th International Conference on Computer Aided Verification, CAV'2002*, LNCS 2404, pages 324–337, Copenhaguen (Denmark), 2002. Springer.

[11] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature. `http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps`, 1997.

[12] R. Corin and S. Etalle. An Improved Constraint-Based System for the Verification of Security Protocols. In M. V. Hermenegildo and G. Puebla, editors, *9th Int. Static Analysis Symposium (SAS)*, LNCS 2477, pages 326–341, Madrid (Spain), 2002. Springer.

[13] M. Debbabi, M. Mejri, M. Tawbi, and I. Yahmadi. Formal Automatic Verification of Authentication Cryptographic Protocols. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, 1997.

[14] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In *LICS' Workshop on Formal Methods and Security Protocols*, 1998.

[15] G. Denker and J. Millen. CAPSL Intermediate Language. In *FLOC's Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[16] G. Denker, J. Millen, J. Kuester-Filipe, and A. Grau. Optimizing Protocol Rewrite Rules of CIL Specifications. In *13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2000.

[17] 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, 1981, Stanford U.

[18] B. Donovan, P. Norris, and Lowem G. Analyzing a Library of Security Protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[19] 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.

[20] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of Bounded Security Protocols. In *FLOC's Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[21] R. Focardi, A. Ghelli, and R. Gorrieri. Using Non Interference for the Analysis of Security Protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[22] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol — Version 3.0, 1996. `draft-freier-ssl-version3-02.txt`.

[23] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *17th International Conference on Automated Deduction*, LNCS 1831, pages 271–290, Pittsburgh (PA, USA), 2000. Springer.

[24] J Goubault-Larrecq. A Method for Automatic Cryptographic Protocol Verification. In *Proc. 15 IPDPS 2000 Workshops, Cancun, Mexico*, LNCS 1800, pages 977–984. Springer, 2000.

[25] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In *Logic for Programming and Automated Reasoning*, LNCS 1955, pages 131–160, St Gilles (Réunion, France), 2000. Springer.

[26] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[27] C. Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. *Journal of Computer Security*, 1(1):5–36, 1992.

[28] C. Meadows. The NRL Protocol Analyzer: an Overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

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

[30] J. Millen and H.-P. Ko. Narrowing Terminates for Encryption. In *PCSFW: Proceedings of the 9th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1996.

[31] J. Millen and V. Shmatikov. Constraint Solving for Bounded-process Cryptographic Protocol Analysis. In *8th ACM Conference on Computer and Communication Security*, pages 166–175. ACM SIGSAC, 2001.

[32] 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.

[33] D. Monniaux. Abstracting Cryptographic Protocols with Tree Automata. In *Sixth International Static Analysis Symposium (SAS'99)*, LNCS 1694. Springer, 1999.

[34] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

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

[36] 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.

[37] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[38] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*. IEEE, 2001.

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

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

[41] D. Song, S. Berezin, and A. Perrig. Athena, a Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9((1,2)):47–74, 2001.

[42] C. Weidenbach. Towards an Automatic Analysis of Security Protocols. In *16th International Conference on Automated Deduction*, LNCS 1632, pages 378–382, Trento (Italy), 1999. Springer.