Strategy for Verifying Security Protocols with Unbounded Message Size

Yannick Chevalier, Laurent Vigneron

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Strategy for Verifying Security Protocols with Unbounded Message Size

Yannick Chevalier — Laurent Vigneron

N° 4368

Janvier 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4368--FR

Strategy for Verifying Security Protocols with Unbounded Message Size

Yannick Chevalier* , Laurent Vigneron†

Thème 2 — Génie logiciel
et calcul symbolique
Projet Protheo

Rapport de recherche n° 4368 — Janvier 2002 — 18 pages

Abstract: We present a system for automatically verifying cryptographic protocols. This system, named Casrul, 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 provers in first-order logic, on-the-fly model-checking, or SAT-based state exploration. These three techniques are being experimented in combination with Casrul in the European Union project AVISS (Automated Verification of Infinite State Systems).
The aim of this paper is to describe the heart of Casrul: the model of the intruder behavior. It is based on a lazy strategy. Another advantage of our model is that it permits to handle parallel sessions and composition of keys. And for sake of completeness, we do not limit the number of sessions of the protocol to be run, nor the size of the messages sent.
We have combined Casrul with the theorem prover daTac for successfully studying 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.

Key-words: security protocols, verification, lazy strategy

* LORIA – Université Henri Poincaré
† LORIA – Université Nancy 2

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Stratégie pour la vérification de protocoles de sécurité avec messages de taille illimitée

Résumé : Nous présentons un système pour la vérification automatique de protocoles cryptographiques. Ce système, nommé Casrul, gère les connaissances des principaux et vérifie si le protocole peut être exécuté. Dans ce cas, il affiche un ensemble de règles de réécriture décrivant le protocole lui-même, la stratégie d'un intrus, et le but à atteindre. Le langage de spécification des protocoles permet d'exprimer des descriptions de propriétés usuelles (authentification, secret à court terme, etc.) ainsi que des structures de données complexes telles des tables et des fonctions de hachage. Les règles de réécriture engendrées peuvent être utilisées pour détecter des failles avec de nombreux systèmes : démonstrateurs en logique du premier ordre, model-checkers, ou explorateurs d'états basés sur SAT. Ces trois techniques sont étudiées en combinaison avec Casrul dans le projet européen AVISS (Automated Verification of Infinite State Systems).
Le but de ce document est de décrire le cœur de Casrul : le modèle du comportement de l'intrus. Il est fondé sur une stratégie paresseuse. Un autre avantage de notre modèle est qu'il permet de traiter des sessions en parallèle et des clés composées. Et pour des besoins de complétude, nous ne limitons pas le nombre de sessions du protocole exécutées, ni la taille des messages envoyés.
Nous avons combiné Casrul avec le démonstrateur daTac pour étudier, avec succès, de nombreux protocoles, tels NSPK, EKE, RSA, Neumann-Stubblebine, Kao-Chow, et Otway-Rees. Nous détaillons certains de ces exemples dans ce document. Actuellement, nous étudions le protocole SET et les premiers résultats sont très encourageants pour la suite.

Mots-clés : protocoles de sécurité, vérification, stratégie paresseuse

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 [22, 4, 7, 23, 28, 15]. Some tools, such as Casper [17], CVS [11] and CAPSL [8], 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 [16], 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 while being protocol independent.

Let us compare Casrul with the most well-known three similar tools. Casper [17] and CVS [11] 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 [24]. 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 states 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 sessions.
Casper and CVS are similar at the specification level, but CVS has been developed to support the so-called Non-Interference approach [13] 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 [20] 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 [19]. 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 ans one for the receiver. The transition rules can be executed by multiset ans 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 [9]: it generates one rule per protocol rule, as we do.
Another important difference between the CIL and the rules generated by Casrul is that we control the evolution of the knowledge of the principals. Hence, we get immediately an optimized version of the rewrite system, which minimizes the number of transitions for verification.

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. And for concluding the comparison with the cited tools, ours has the advantage to be able to handle infinite states models, and to be closer to the original Dolev-Yao model [10].

The output of our compiler can be used to get a representation of protocols in various systems:

- As Horn clauses, it can be used 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 ELAN program.
- As propositional formulas, it 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 information, such as nonces or keys [16]. Note that there was a first approach with narrowing by Meadows in [18]. The narrowing rules were restricted to symbolic encryption equations (see also [21]). Transitions were rather 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, containing the description of new features, 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 [5], is completely different and much

more efficient than the model of the Intruder presented in [16]. 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.

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 [16], and is close to one of CAPSL [20] or Casper [17] 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 [27].
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 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 [16], 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. $P \rightarrow Q$: $\{Nq\}Kpq$
Parallel
$S[P : a; Q : I; S : se; Kps : kps; Kqs : kis]$;
Secret
kps;
Session_instances
$[P : a; Q : I; S : se; Kps : kas; Kqs : kis]$
Intruder *Divert*, *Impersonate*;
Intruder_knowledge ;
Goal *Correspondence_between P S*;

Figure 1: Woo and Lam Mutual Authentication Protocol.

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

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.

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

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), 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 four kinds of encodings, the last two being new features added in the last version of Casrul:

- 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.
- A *function* f is a one-way, injective, 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; this is the contents of this section.
An identifier (key or number) that is not in any initial knowledge will be used as a *fresh* information, created at its first use. It is possible to give *messages* in the initial knowledge of a principal as long as all identifiers appearing in the 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 is very classical for encoding: $\{M\}_K$ means the message M encode 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*, 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 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 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, with respect to the initial specification, 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 right key).

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

Figure 2: The function *compose*

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

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 done by the function *expect* defined in Figure 3. This function describes the behavior of

$$
\begin{aligned}
expect(U, M, i) &= compose(U, M, i) && \text{if no } \texttt{Fail} && (10)\\
expect(U, \langle M_1, M_2\rangle, i) &= \langle expect(U, M_1, i), expect(U, M_2, i)\rangle && && (11)\\
expect(U, (M_1)\mathsf{xor}(M_2), i) &= (compose(U, M_1, i))\mathsf{xor}(expect(U, M_2, i)) && \text{if no } \texttt{Fail} && (12)\\
expect(U, (M_1)\mathsf{xor}(M_2), i) &= (expect(U, M_1, i))\mathsf{xor}(compose(U, M_2, i)) && \text{if no } \texttt{Fail} && (13)\\
expect(U, \{M\}_K, i) &= \{expect(U, M, i)\}_{compose(U,K^{-1},i)^{-1}} && \text{if no } \texttt{Fail} && (14)\\
expect(U, \{M\}_{K^{-1}}, i) &= \{expect(U, M, i)\}_{compose(U,K,i)^{-1}} && \text{if no } \texttt{Fail} && (15)\\
expect(U, \{M\}_{SK}, i) &= \{expect(U, M, i)\}_{compose(U,SK,i)} && \text{if no } \texttt{Fail} && (16)\\
expect(U, f(M), i) &= compose(U, f(M), i) && \text{if no } \texttt{Fail} && (17)\\
expect(U, M, i) &= x_{U,M,i} && \text{else} && (18)
\end{aligned}
$$

Figure 3: The function *expect*

a principal U that tries to give an as accurate as possible value to every part of a message it will receive. All the unknown ciphers are replaced by variables.

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.

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 [10].

2.3 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. Among all the following sections, the new features proposed in the last version of Casrul are the possibility to define roles and parallel roles, and to look for new kinds of flaws: authentication and short time secrecy.

2.3.1 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 [17]) 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.

2.3.2 Roles

This field can be used together with the `Session_instances` field, or on its own. Using it, one can define principals independent one from another, instead of being bounded in one session. It thus permits greater flexibility in the definition of the protocol.

2.3.3 Parallel Roles

This field is used to specify instances of principals that can be run an unbounded number of times in parallel. 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. A secrecy goal is generated for each of those secret instances.

2.3.4 Intruder

The `Intruder` field describes which strategies the Intruder can use, among three ones: passive `eaves_dropping`, `divert` and `impersonate`. If nothing is specified, this means that we want a simulation of the protocol without Intruder.
If an Intruder is present, he can record all the messages spread over the net. In addition, if `divert` is selected, he can remove messages; if `eaves_dropping` is selected, he cannot.
The Intruder is then able to reconstruct terms as it wishes, using all the information it got. He can send arbitrary messages in his own name.
If moreover `impersonate` is selected, then he can fake others identity in sent messages.

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

2.3.5 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 `Session_instances`, `Role` or `Parallel` role, as an effective knowledge (and not a formal one used for describing the messages).

2.3.6 Goal

This field gives the kind of flaw we want to detect. There are several possibilities, but the two main ones are `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 has been 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 [16].

In addition, and in order to cope with the `Role` field introduction, we also use the `authenticate` goal. The goal:

$$B \texttt{ authenticates } A \texttt{ on } Na, Nb$$

means that if a principal b playing the role B ends its part of the protocol and believes it has interacted with a principal a playing the role of A, and if it believes it has received Na and Nb sent by a, then, at some point, the principal a must have sent Na and Nb to b. This goal is more precise than the `Correspondence_between` goal, since there is no more a privileged link between the principals of one session, and it is not symmetric anymore.

A last goal has been introduced in order to automatically handle the case of some compromised secret: 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 Dolev-Yao's model of an Intruder [10] 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 of the number of sessions. Thus, those bounds are not relevant in our approach, and this has led us to bring a new model of the Intruder.

A proposed approach to deal with this infinite-space problem is to use a lazy model while testing the protocol by model-checking [2]. Though in a different approach, 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, thus defining our model as a lazy one.
This strategy may look similar to the one described in [25] (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, thus preparing those messages in advance, statically. One advantage of our method is that we can find some type flaws (in the Otway-Rees protocol, for instance) that cannot be found at the compilation time.
In the previous version of Casrul, 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.

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 `Session_instances`, `Parallel` and `Role` 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 coupling;
- 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;

- $\text{FUNC}(t_1, t_2)$ is valid if t_1 is a function symbol and t_2 is a message. In this case, $\text{FUNC}(t_1, t_2)$ is the hash of t_2 computed with algorithm t_1.

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 associative and commutative (AC) operator.

3.1 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 4 tries to reduce the expression $Comp(t)$ *from* C ; Id (where Id stands for the identity substitution), building a substitution τ.

$$Comp(t).T \text{ from } t.C \ ; \ \tau \quad \rightarrow \quad T \text{ from } t.C \ ; \ \tau \tag{19}$$

$$Comp(r).T \text{ from } s.C \ ; \ \tau \quad \overset{r\sigma = s\sigma}{\rightarrow} \quad T\sigma \text{ from } s\sigma.C\sigma \ ; \ \tau\sigma \tag{20}$$

$$Comp(\text{C}(t_1, t_2)).T \text{ from } C \ ; \ \tau \quad \rightarrow \quad Comp(t_1).Comp(t_2).T \text{ from } C \ ; \ \tau \tag{21}$$

$$Comp(\text{CRYPT}(t_1, t_2)).T \text{ from } C \ ; \ \tau \quad \rightarrow \quad Comp(t_1).Comp(t_2).T \text{ from } C \ ; \ \tau \tag{22}$$

$$Comp(\text{SCRYPT}(t_1, t_2)).T \text{ from } C \ ; \ \tau \quad \rightarrow \quad Comp(t_1).Comp(t_2).T \text{ from } C \ ; \ \tau \tag{23}$$

$$Comp(\text{XCRYPT}(t_1, t_2)).T \text{ from } C \ ; \ \tau \quad \rightarrow \quad Comp(t_1).Comp(t_2).T \text{ from } C \ ; \ \tau \tag{24}$$

$$Comp(\text{TABLE}(t_1, t_2)).T \text{ from } C \ ; \ \tau \quad \rightarrow \quad Comp(t_1).Comp(t_2).T \text{ from } C \ ; \ \tau \tag{25}$$

$$Comp(\text{TABLE}(t_1, t_2)^{-1}).T \text{ from } C \ ; \ \tau \quad \rightarrow \quad Comp(t_1).Comp(t_2).T \text{ from } C \ ; \ \tau \tag{26}$$

$$Comp(\text{FUNC}(t_1, t_2)).T \text{ from } C \ ; \ \tau \quad \rightarrow \quad Comp(t_1).Comp(t_2).T \text{ from } C \ ; \ \tau \tag{27}$$

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

This system, being complete[2] 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 (20) 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. This restriction is mandatory in our system, since the Intruder would otherwise test terms of unbounded depth.

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:

$$\begin{aligned}
& Comp(\text{CRYPT}(\text{C}(\text{MR}(a), x_1), \text{SCRYPT}(sk(k_a), x_2))) \\
& \qquad \text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(22)} \quad & Comp(\text{C}(\text{MR}(a), x_1)).Comp(\text{SCRYPT}(sk(k_a), x_2)) \\
& \qquad \text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(21)} \quad & Comp(\text{MR}(a)).Comp(x_1).Comp(\text{SCRYPT}(sk(k_a), x_2)) \\
& \qquad \text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(19)} \quad & Comp(x_1).Comp(\text{SCRYPT}(sk(k_a), x_2)) \\
& \qquad \text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(20)} \quad & Comp(x_1) \\
& \qquad \text{from } \text{MR}(a).\text{SCRYPT}(sk(k_a), nonce(Na)) \ ; \ \sigma
\end{aligned}$$

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.

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

[2] Except in the case of rule (24), which is an approximate rule for handling *Xor*-encryption.

$$C.UFO(\text{F}(t).C') \rightarrow C.\text{F}(t).UFO(C') \quad (28)$$

$$C.UFO(\text{C}(t_1,t_2).C') \rightarrow C.UFO(t_1.t_2.C') \quad (29)$$

$$C.UFO(\text{CRYPT}(t_1,t_2).C') \rightarrow C.\text{CRYPT}(t_1,t_2).UFO(\text{TEST}(\text{CRYPT}(t_1,t_2)).C') \quad (30)$$

$$\textit{if } A({t_1}^{-1},C,\sigma): \quad C.UFO(\text{TEST}(\text{CRYPT}(t_1,t_2)).C') \rightarrow C.UFO(t_2.C')\sigma \quad (31)$$

$$C.UFO(\text{SCRYPT}(t_1,t_2).C') \rightarrow C.\text{SCRYPT}(t_1,t_2).UFO(\text{TEST}(\text{SCRYPT}(t_1,t_2)).C') \quad (32)$$

$$\textit{if } A(t_1,C,\sigma): \quad C.UFO(\text{TEST}(\text{SCRYPT}(t_1,t_2)).C') \rightarrow C.UFO(t_2.C')\sigma \quad (33)$$

$$C.UFO(\text{XCRYPT}(t_1,t_2).C') \rightarrow C.\text{XCRYPT}(t_1,t_2).UFO(\text{TEST}(\text{XCRYPT}(t_1,t_2)).C') \quad (34)$$

$$\textit{if } A(t_1,C,\sigma): \quad C.UFO(\text{TEST}(\text{XCRYPT}(t_1,t_2)).C') \rightarrow C.UFO(t_2.C')\sigma \quad (35)$$

$$\textit{if } A(t_2,C,\sigma): \quad C.UFO(\text{TEST}(\text{XCRYPT}(t_1,t_2)).C') \rightarrow C.UFO(t_1.C')\sigma \quad (36)$$

$$C.UFO(\text{TABLE}(t_1,t_2).C') \rightarrow C.\text{TABLE}(t_1,t_2).UFO(C')\sigma \quad (37)$$

$$C.UFO(\text{TABLE}(t_1,t_2)^{-1}.C') \rightarrow C.\text{TABLE}(t_1,t_2)^{-1}.UFO(C')\sigma \quad (38)$$

$$C.UFO(\text{FUNC}(t_1,t_2).C') \rightarrow C.\text{FUNC}(t_1,t_2).UFO(C')\sigma \quad (39)$$

Figure 5: System Simplifying the Intruder's Knowledge.

a new one, faked so as it looks like it has been sent by another principal (chosen by the Intruder). We define 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 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 ${t^{-1}}^{-1} = t$.

We note this system with $A(t,C,\sigma)$, 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.

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:

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

This is used, for instance in the example of Section 3.1, to prove it is sound to substitute $\text{NONCE}(Na)$ 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 the Intruder has to verify he 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.

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. The, with the EKE protocol, we show how we deal with parallel sessions. We also list the results obtained for other protocols that can be found in [6].
But first, let us give a short presentation of the prover used.

4.1 The Prover daTac

For studying the protocols, we have used the theorem prover daTac[3], 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. 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.

The deduction techniques used by daTac are Resolution and Paramodulation [26]. They are combined with efficient simplification techniques for eliminating redundant information. Another important property is that this theorem prover is refutationnaly 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 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
% rdatac -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:

[3] `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. $A \rightarrow B$: $M, A, B, \{Na, M, A, B\}Kas$
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. $A \rightarrow B$: $\{X\}Kab$

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

Figure 6: Otway-Rees Protocol.

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

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:

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

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:

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

The Intruder has proved that he can send a term matching the pattern of 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 correspondence-between-principals attack. The Casrul specification of this protocol is given in Figure 7.

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)			

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

Figure 7: Encrypted Key Exchange Protocol.

We can study in deeper details this trace in order to find the scenario of the attack. Since the principal a appears in two sessions, 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:

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

Now, 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 [5], where we used two sessions running concurrently. The time needed to reach this flaw was around 2 minutes, and the number of states explored before reaching the flaw was around 200 (here, only 7!).

4.4 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 old version of Casrul, and the new one that permits to use parallel sessions and roles.

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 no attack was reported in the literature. All those results have been obtained with a PC under Linux. 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 rewriting rules produced by Casrul, can be turned into a time and space efficient procedure.

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, showing that there is no flaw in the Card-holder Registration part as already done by Paulson et al. in [3]. 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) [14].

5 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, 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. For instance, in the European Union project AVISS (Automated Verification of Infinite State Systems)[5], it is also used with an on-the-fly model-checker and with a propositional formulae satisfiability checker.
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. 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.

[4] In this case, both errors are found. We give the time for finding the first flaw (Intruder impersonates the client).
[5] `http://www.informatik.uni-freiburg.de/~softech/research/projects/aviss/`

Protocol	Old Timings	New Timings	Kind of Flaw
Andrew Secure RPC	7s	7s	Authenticate Flaw
Encrypted Key Exchange	268s	3s	Authenticate Flaw
Kehne-Schoene-Langendorf	69s	20s	Correspondence 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 [4]	53s		Correspondence Flaw
SPLICE (Intruder impersonates client)		6s	Authenticate Flaw
SPLICE (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

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

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 sessions, 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 [12], 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. Technical report, INRIA Research Report 3915, Marseille (France), 2000.

[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] G. Bella, F. Massacci, L. C. Paulson, and P. Tramontano. Formal Verification of Cardholder Registration in SET. In *Proceedings of the Sixth European Symposium on Research in Computer Security (ESORICS)*, volume 1895 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2000.

[4] D. Bolignano. Towards the formal verification of electronic commerce protocols. In *10th IEEE Computer Security Foundations Workshop*, pages 133–146. IEEE Computer Society, 1997.

[5] 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), November 2001. IEEE CS Press.

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

[22] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, pages 141–154. IEEE Computer Society, 1997.

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

[24] 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.

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

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

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

[28] 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.

Contents

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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