A Tool for Lazy Verification of Security Protocols [*]

Y. Chevalier 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 the lazy strategy implemented in a compiler of cryptographic protocols, Casrul. The purpose of this compiler is to verify protocols and to translate them into rewrite rules that can be used by several kinds of automatic or semi-automatic tools for finding flaws, or proving properties. It is entirely automatic, and the efficiency of the generated rules is guaranteed because of the use of a lazy model of an Intruder behavior. This efficiency is illustrated on several examples.

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 [13, 3, 6, 15]. Some tools (Casper [11], CVS [9], CAPSL [7]) have also been developed for automating one of the most sensitive step: the translation of a protocol specification into a low-level language that can be handled by automated verification systems.

Our work is in this last line. We have designed a protocols compiler, Casrul [10], that translates a cryptographic protocol specification 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. A comparison of Casrul with systems such as CAPSL and Casper can be found in [10]. But for short, let us say that our tool handles infinite states models, and is closer to the original Dolev-Yao model.

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 nonces or keys [10]. Note that there was a first approach with narrowing by Meadows in [12].

The main objective of this paper is to present an innovative model of the Intruder behavior, based on the definition of a lazy model. This lazy approach is completely different and much more efficient than the model of the Intruder presented in [10]. In Section 3, we show that our method can be successfully applied to many different kinds of protocols.
A more complete description of Casrul and its lazy strategy, illustrated on several examples, can be found in [5].

2 Intruder's model

One of the biggest problem in the area of cryptographic protocols verification is the definition of the Intruder. The Dolev-Yao's model of an Intruder [8] is not scalable, since there are rules for composing messages and these rules, such as building a couple from two terms, do not terminate: given a term, it is possible to build a couple with two copies of this term, and to do it again with that couple.

In some approaches people try to bound the size of the messages, but these bounds are valid 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 led us to bring a new model of the Intruder.

A proposed approach to deal with this infinite-space

[*]This work was supported by the Future Emerging Technologies, Information Society Technologies (IST) Programme, FET Open Assessment Project: IST-2000-26410.

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 its knowledge and tests if it 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 [16] (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 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 (in the Otway-Rees protocol for instance) 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 constants declared in the protocol;
- Some unary operators are used to type those constants, such as $\textsc{mr}$ to describe a principal; we also use $\textsc{f}$ for representing any of those operators;
- The $\textsc{c}$ operator stands for coupling;
- $\textsc{crypt}$, $\textsc{scrypt}$ and $\textsc{xcrypt}$ operators stand respectively for public or private key encryption, symmetric key encryption and xor-encryption.

We also use other operators whose meaning should be clear from the name, e.g. the $Comp$ operator. The "**.**" operator is an AC operator.

2.1 Test of the Composition of a Term

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

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

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

where σ is the most general unifier of r and s

$$Comp(\textsc{c}(t_1,t_2)).T \textit{ from } C \ ; \ \tau \ \rightarrow \ Comp(t_1).\,Comp(t_2).T \textit{ from } C \ ; \ \tau \qquad (3)$$

$$Comp(\textsc{crypt}(t_1,t_2)).T \textit{ from } C \ ; \ \tau \ \rightarrow \ Comp(t_1).\,Comp(t_2).T \textit{ from } C \ ; \ \tau \qquad (4)$$

$$Comp(\textsc{scrypt}(t_1,t_2)).T \textit{ from } C \ ; \ \tau \ \rightarrow \ Comp(t_1).\,Comp(t_2).T \textit{ from } C \ ; \ \tau \qquad (5)$$

$$Comp(\textsc{xcrypt}(t_1,t_2)).T \textit{ from } C \ ; \ \tau \ \rightarrow \ Comp(t_1).\,Comp(t_2).T \textit{ from } C \ ; \ \tau \qquad (6)$$

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

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

$$\begin{aligned}
& Comp(\textsc{crypt}(\textsc{c}(\textsc{mr}(a), x_1), \textsc{scrypt}(sk(k_a), x_2))) \\
& \quad \textit{from } \textsc{mr}(a).\textsc{scrypt}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(4)} \quad & Comp(\textsc{c}(\textsc{mr}(a), x_1)).\,Comp(\textsc{scrypt}(sk(k_a), x_2)) \\
& \quad \textit{from } \textsc{mr}(a).\textsc{scrypt}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(3)} \quad & Comp(\textsc{mr}(a)).\,Comp(x_1).\,Comp(\textsc{scrypt}(sk(k_a), x_2)) \\
& \quad \textit{from } \textsc{mr}(a).\textsc{scrypt}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(1)} \quad & Comp(x_1).\,Comp(\textsc{scrypt}(sk(k_a), x_2)) \\
& \quad \textit{from } \textsc{mr}(a).\textsc{scrypt}(sk(k_a), nonce(Na)) \ ; \ Id \\
\rightarrow^{(2)} \quad & Comp(x_1) \\
& \quad \textit{from } \textsc{mr}(a).\textsc{scrypt}(sk(k_a), nonce(Na)) \ ; \ \sigma
\end{aligned}$$

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

2.2 Decomposition of the Intruder's Knowledge

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

$$C.\mathit{UFO}(\textsc{f}(t).C') \;\rightarrow\; C.\textsc{f}(t).\mathit{UFO}(C') \quad (7)$$

$$C.\mathit{UFO}(\textsc{c}(t_1, t_2).C') \;\rightarrow\; C.\mathit{UFO}(t_1.t_2.C') \quad (8)$$

$$C.\mathit{UFO}(\textsc{crypt}(t_1, t_2).C') \;\rightarrow\; C.\textsc{crypt}(t_1, t_2).\mathit{UFO}(\textsc{test}(\textsc{crypt}(t_1, t_2)).C') \quad (9)$$

$$\mathit{if}\, A({t_1}^{-1}, C, \sigma) : C.\mathit{UFO}(\textsc{test}(\textsc{crypt}(t_1, t_2)).C') \;\rightarrow\; C.\mathit{UFO}(t_2.C')\sigma \quad (10)$$

$$C.\mathit{UFO}(\textsc{scrypt}(t_1, t_2).C') \;\rightarrow\; C.\textsc{scrypt}(t_1, t_2).\mathit{UFO}(\textsc{test}(\textsc{scrypt}(t_1, t_2)).C') \quad (11)$$

$$\mathit{if}\, A(t_1, C, \sigma) : C.\mathit{UFO}(\textsc{test}(\textsc{scrypt}(t_1, t_2)).C') \;\rightarrow\; C.\mathit{UFO}(t_2.C')\sigma \quad (12)$$

$$C.\mathit{UFO}(\textsc{xcrypt}(t_1, t_2).C') \;\rightarrow\; C.\textsc{xcrypt}(t_1, t_2).\mathit{UFO}(\textsc{test}(\textsc{xcrypt}(t_1, t_2)).C') \quad (13)$$

$$\mathit{if}\, A(t_1, C, \sigma) : C.\mathit{UFO}(\textsc{test}(\textsc{xcrypt}(t_1, t_2)).C') \;\rightarrow\; C.\mathit{UFO}(t_2.C')\sigma \quad (14)$$

$$\mathit{if}\, A(t_2, C, \sigma) : C.\mathit{UFO}(\textsc{test}(\textsc{xcrypt}(t_1, t_2)).C') \;\rightarrow\; C.\mathit{UFO}(t_1.C')\sigma \quad (15)$$

In this system, $A(t, C, \sigma)$ denotes a predicate that is true whenever the term t can be build from the knowledge C using a substitution σ. The system of Section 2.1 shows that this predicate can be implemented with rewrite rules similar to those that are used to test if a principal can compose a message that matches the pattern of an awaited message.

2.3 Use of this Model for Flaws Detection

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

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

$$l \overset{\mathrm{def}}{=} (T_1\ \mathit{from}\ C_1) : \ldots : (T_n\ \mathit{from}\ C_n)$$

This is used, for instance in the example of Section 2.1, to prove it is sound to substitute $\textsc{nonce}(Na)$ for x_2.
We also maintain a set of knowledge C representing the Intruder's knowledge evolution whenever it succeeds in sending an appropriate message. We model a protocol step with the rule:

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

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

3 Experimentations

For studying the protocols, we have used the theorem prover daTac, 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 technique used by daTac is narrowing, based on Resolution and Paramodulation [17]. 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.

The study of the protocols given in the next table is straightforward and done in an automatic way. These protocols can be found in [4].

We point out that, in all protocols but one studied up to now, we have always detected 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 explorated clauses, using a breadth-first search strategy, is always smaller than a few hundreds. This demonstrates that our lazy model represented by the rewriting rules produced by Casrul can be turned into a time efficient procedure.

4 Conclusion

We have designed and implemented in Casrul a compiler of cryptographic protocols, transforming a general specification into a set of rewrite rules. The transformation to rewrite rules is fully automatic and high level enough to permit further extensions or case specific extensions. For

Protocol	User Time	Kind of Flaw
Secure RPC	41s	Compromised Key
Encrypted Key Exchange	110s	Correspondence Between Principals
Encrypted Key Exchange[1]	8s	Correspondence Between Principals
NSPK Exchange	23s	Correspondence Between Principals
TMN	18s	Correspondence Between Principals
RSA	0.5s	Secrecy Flaw
Woo-Lam (π)	22s	Correspondence Between Principals
Woo-Lam (3)	4s	Correspondence Between Principals
Woo-Lam Mutual Authentification	340s	Correspondence Between Principals
SPLICE/AS	59s	Correspondence Between Principals
Neumann-Stubblebine (Part 1)	2s	Correspondence Between Principals
Kao Chow	24s	Compromised Key
Otway-Rees	1.5s	Secrecy

example, one can model specific key properties such as key commutativity in the RSA protocol.

The protocol model generated is general enough to be used for various verification methods: model-checking, proof by induction, narrowing, . . . In addition, its lazy strategy makes it efficient for all these kinds of methods.
In our case, we have used narrowing with the theorem prover daTac, demonstrating that many different examples can be efficiently compiled with Casrul.

We are now moving on to the study of more complex protocols, we have already added hash functions, modeled as free constructors. Another direction is to express parallel sessions with simplified versions of the principals, and demand-driven rules. It allows the use of an unbounded number of such sessions, thus permitting us a more refined study of protocols in which our tool failed to find a flaw. 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 [14], we would have to restrict our model in order to keep implementability.

References

[1] R. Amadio and D Lugiez. On the Reachability Problem in Cryptographic Protocols. In *RR-INRIA — 3915*. Marseille (France).

[2] D. Basin. Lazy Infinite-State Analysis of Security Protocols. In R. Baumgart, editor, *Secure Networking — CQRE'99*, volume 1740 of *Lecture Notes in Computer Science*, pages 30-42. Springer-Verlag, Düsseldorf (Germany), 1999.

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

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

[5] Y. Chevalier and L. Vigneron. A Tool for Lazy Verification of Security Protocols (full version). *Research Report A01-RR-140*, LORIA, Nancy (France), 2001.

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

[7] G. Denker and J. Millen. CAPSL Intermediate Language. In *Formal Methods and Security Protocols*, 1999. FLOC'99 Workshop.

[8] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29:198-208, 1983. Also STAN-CS-81-854, 1981, Stanford U.

[9] 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.

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

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

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

[13] J. 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.

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

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

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

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

[1] With roles in parallel simplification.