Chapter 10
An Analysis of a Public Key Protocol with Membranes

Olivier Michel[1], Florent Jacquemard[2]

[1] LaMI CNRS umr 8042 – Université d'Évry
Tour Évry-2, 523 place des terrasses de l'agora, 91000 Évry, France
michel@lami.univ-evry.fr
[2] INRIA FUTURS and LSV, CNRS umr 8643 – ENS de Cachan
61 avenue du Président Wilson, 94235 Cachan Cedex, France
florent.jacquemard@lsv.ens-cachan.fr

Summary. We develop an analysis of the Needham–Schroeder public key protocol in the framework of membrane computing. This analysis is used to validate the protocol and exhibits, as expected, a well known logical attack. The novelty of our approach is to use multiset rewriting in a nest of membranes. The use of membranes enables to tight the conditions for detecting an attack. The approach has been validated by developing a full implementation for several versions of the analysis.

1 Goal and Motivations

Since the 1994 landmark demonstration by Adleman of the possibilities of DNA to solve a class of combinatorial problems, biocomputing has often be advocated to develop "chemically combinatorial problem solvers". In this chapter, we want to use an approach belonging to the membrane computing [23] area to address a well known combinatorial problem: the analysis of a cryptographic protocol.

Our starting point is the logical analysis of the Needham–Schroeder public key protocol (NSPK). The goal of the logical analysis is to find an interleaving of elementary actions (sending and answering messages) that allows an intruder to obtain confidential information. We have chosen this problem because it is simple to explain, at the same time it requires sophisticated data–structures for the exploration of its state space, it is paradigmatic of this kind of applications, and its solution is well–known – hence we can validate our result.

The approach taken in this chapter is brute force and consists in the exploration of the state space of the protocol for a systematic search of attacks.

Indeed, we are interested in the study of the states representation and generation, rather than in designing a new and smart search strategy. This approach is motivated by the opinion that the representation of data is a central problem in biocomputing.

The rest of this chapter is organized as follows. In Section 2 we give some background on the logical analysis of cryptographic protocols. Section 3 describes precisely the Needham–Schroeder public key protocol. Section 4 presents the technical meat of the chapter. We develop a version of the analysis of NSPK that improves on a similar analysis initially proposed within the ELAN rewriting framework [5], with a more accurate representation of states using nesting. In the appendix are given a short presentation of the `MGS` language, which enables a kind of membrane computing, together with the `MGS` code of the algorithms detailed in Section 4.

2 Formal Verification of Cryptographic Protocol

In this section, we give a brief introduction to the verification problem we shall consider. Cryptographic protocols define the exchange of a few messages between parties in order to distribute some secrete data like cryptographic keys or to authenticate themselves. These messages are built with cryptographic primitives, like encryption, signature or hash–functions, and therefore the security of protocols relies on the strength of the cryptographic functions in use. However, it has appeared that even though when these functions are assumed unbreakable, the security of a protocol can be compromised by an unexpected interleaving of messages between honest agents and a malicious intruder which has some limited control over the communication network (like, e.g., wire–tapping some messages or impersonating identities while sending new ones). For instance, the well known problems of the distribution of keys for symmetric cryptosystems like AES and the authenticity of public keys in PKIs are beyond the scope of the study of encryption functions.

Such *logical* attacks can be realized at almost no computational cost and hence can have disastrous consequences. Various formal methods have been proposed for the automation of the analysis of the vulnerability of cryptographic protocols to logical attacks, both for searching of flaws of this kind or for the formal proof of their absence. Several systems have been implemented in purpose for the search of flaws, e.g., [18, 17, 13]. But many general purpose languages and tools have also appeared appropriate in this setting, with the advantage of a greater expressive power, efficiency and maturity. To cite only a few examples, there are model checkers like FDR [16] or murφ [21], first order theorem provers [25, 14] and declarative languages used as model checkers [7, 5].

Our purpose in this chapter is to describe an experiment to use membranes for modeling a cryptographic protocol and finding of attacks by state exploration. The declarative style supported by the membrane computing frame-

work is strongly advocated by the intruder–centric model which is generally considered in order to apply formal methods to cryptographic protocol verification. In this model, often referred as "Dolev–Yao model" [8], the agents executing the protocol communicate asynchronously via a unique channel which has been compromised by an intruder. The intruder is able to spy and divert every message on the channel, to analyze read messages, with the restriction that he must know the appropriate encryption key in order to decipher an encrypted message. He can also build and send new messages, possibly under a fake identity. The global state of the system can hence be represented by a heterogeneous set containing the local states of each agent (with a bounded memory), the messages and sub–messages known to the intruder and the messages sent and not yet received by an agent. The actions of the agents (receiving and sending messages) as well as of the intruder can be modeled using rewriting rules on multisets. The search of an interleaving leading to an attack can be coded very simply with an appropriate pattern expression to find sequences of value or arbitrary length.

The problem of finding attacks of protocols is highly undecidable, the state space being infinite for several reasons: the unboundedness of the number of agents in presence, the ability of agents to generate fresh random data (nonces), the unlimited size of terms generated by the intruder. In order to restrict our exploration to a finite search space, while keeping our procedure reasonably complete, we shall rely on some theoretical results on protocol verification. It is shown in [24] that the problem of protocol security (non-existence of attacks) becomes decidable when the number of agents considered is bounded. Indeed, [24] shows that in this case, whenever there exists an attack, there exists an attack involving messages of a bounded size. We can use this result here to ensure the completeness of our attack search procedure, given a finite number of agents.

3 The Needham–Schroeder Public Key Protocol

The Needham–Schroeder public key protocol [22] (NSPK for short) is the favorite example for the application of formal methods to the verification of cryptographic protocols. This popularity certainly comes from one of the most famous success story in this domain, which is the discover in 1994 by G. Lowe [16] of a replay attack in this protocol 16 years after its publication. In [16], G. Lowe models the protocol in the CSP process algebra and uses the model checker FDR to explore the state space. We obtain here the same result with a model based on membranes computing, implemented in the language `MGS` .

3.1 Description of the Protocol

The Needham–Schroeder public key protocol involves two participants Alice (A), Bob (B) which are willing to authenticate reciprocally with three mes-

sages using public keys. The original protocol of [22] involves also a server distributing the public keys to A and B with three additional messages. We omit the server and its three messages here, assuming that A and B both initially know each other's public key, since they are not necessary in Lowe's attack. The messages are described below in the usual notation (see also Figure 1):

$$\begin{array}{ll} \texttt{REQ} & A \to B : \{A, N_a\}_{K(B)} \\ \texttt{CHAL} & B \to A : \{N_a, N_b\}_{K(A)} \\ \texttt{AUTH} & A \to B : \{N_b\}_{K(B)} \end{array}$$

In the first message (labelled `REQ`), Alice generates a random number (*nonce*) N_a, appends it to her name A (the append operator is denoted $_,_$) encrypts the results with Bob's public key $K(B)$ (public key encryption is denoted with the binary operator $\{_\}__$) and sends the result to the network. When Bob receives a message of the form of `REQ`, he deciphers it and retrieves the identity A of Alice and the nonce N_a. Then he generates a second random number N_b, appends it to N_a and sends back the result encrypted with Alice's public key $K(A)$ (message `CHAL`for a challenge). Alice, receiving message `CHAL`, can decipher it and check whether the first component corresponds to the nonce she sent in message `REQ`. Then, she resends Bob's nonce N_b encrypted with Bob's public key (message `AUTH`). Bob can check that the message `AUTH` contains the nonce N_b he has generated at second step (`CHAL`).

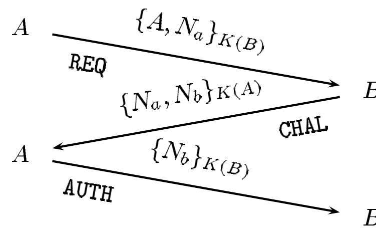

Fig. 1. Description of the NSPK protocol.

3.2 A Replay Attack

Receiving the message `AUTH` ensures Bob that Alice has really received the message `CHAL` and answered, because Alice is the only one able to decipher this message. We assume indeed that each agent, as well as the intruder (let us call him Charly, C), knows only its own private key, and that this key is necessary to decipher a message encrypted with the corresponding public key.

Similarly, when receiving the message `CHAL`, Alice is ensured that it really comes from B (and is not a fake message from Charly), as proven by the presence of N_a because the knowledge of Bob's private key is necessary for the extraction of N_a from the message `REQ`. Hence, N_a and N_b are used as *authenticators* in this protocol, and they must remain secret. However, the

attack of [22], described in Figure 2, shows that it is not the case, even with the above hypotheses concerning the private keys.

This attack involves two sessions in parallel. In the first session, Alice enters in communication with Charly (without knowing that he is an intruder). Since the message REQ is encrypted with Charly's public key $K(C)$, Charly can retrieve A, N_a and encrypts it with Bob's public key $K(B)$. He then sends this message as the first message REQ′ of a second session between A and B. In this step REQ′, Charly impersonates A, which is denoted $C(A)$. Bob answers to REQ′ and Charly diverts this message CHAL′ (it is by denoted $C(A)$). Then Charly, with two messages CHAL and AUTH of the first session uses A as an oracle in order to obtain Bob's nonce N_b.

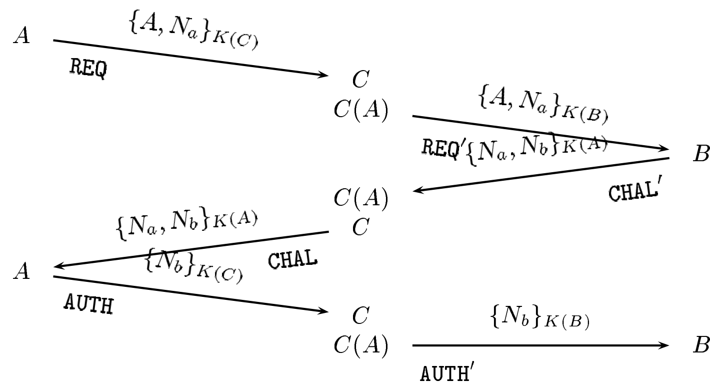

Fig. 2. A Replay attack following G. Lowe.

4 Finding an Attack on the NSPK Using Membranes

We shall describe here the specification of the Needham–Schroeder publi–key protocol and the implantation of an attack–search procedure using rules and membranes. For the implementation, we rely on rewriting modulo associativity and commutativity (AC) on terms representing nested multisets (membranes). Rewriting rules can be guarded by arbitrary conditions. This model is similar to the chemical computations presented in [1] and we use the term "chemical solution" to denote the content of a membrane. Examples of systems implementing such model of computations are Gamma, ELAN [2], MAUDE [3] or MGS [9]. The ingredients of this model of computation are rather sophisticated but a translation into the fundamental core mechanisms of P systems is possible and we give in [19] some elements that support this assertion.

We present in this chapter the principles of a simple version of the attack–search procedure. This version improves on a similar analysis initially proposed within the ELAN rewriting framework, because we use a more accurate

representation of states using nesting. The functional representation of the interleaving of actions is also a new idea. Note that another version is described and fully detailed in [20]. This last version goes further by generalizing the approach to the exploration of general state spaces and does not rely on the assumption that attacks involve messages of bounded size.

The description of the protocol involves two different kind of components: *entities* and *evolution rules*. The entities are records and evolution rules are given by rewrite rules. A *system state*, we shall also write *solution*, is a finite collection of entities which are of three kind: agents, messages transmitted trough the network and messages components memorized by the intruder. Several entities in a state shall react, firing an evolution rule which transforms a system state into a successor state. The model is organized into the following parts, detailed in the next sections:

- record definitions, used to describe the three kind of entities (Section 4.1);
- various predicates used to select, in the set of reacting entities, a specific entity of a given kind – an agent, a message (Section 4.1);
- rules specifying the abilities of the intruder to collect all the messages that have been exchanged between agents and extracts pertinent information (Section 4.2);
- rules specifying the abilities of the intruder to produce fake messages from the information gathered so far (Section 4.2);
- rules specifying the reception and sending of messages by agents: such rules are defined as reactions between an agent and a (received) message which fulfills some conditions (Section 4.3);
- rules implementing a state exploration procedure which halts with a predicate checking whether a bad state is reached, hence that the search of an attack was successful (Section 4.3).

4.1 Representing Agents, Messages and Intruder Knowledge

The three different kinds of entities (unstructured information) found in the system states (solutions) are represented using records (the `MGS` code for this section can be found in Appendix B.1).

Agents. We shall distinguish the *roles*, Alice and Bob in our example, which are programs, from the *agents* executing the programs, characterized by an identifier (agent's name), a role and a bounded memory. In particular, there can be several agents for one role. An agent consists in:

- an *identity* `id` (its name; several agents may have the same identity),
- two *stores* `ni` and `nr` to memorize the session–specific values of the nonces N_a and N_b,
- a *program counter* `pc`, which can take the value described below.

Every agent with either role Alice or Bob shall create a nonce and receive another one during the execution of the protocol of Section 3.1. The fields

`ni` and `nr` store these two values, for Alice, `ni` stores N_a and `nr` stores N_b, and reciprocally for Bob (`ni` stands for *nonce initial*, because we can assume that each agent initially creates the nonces before starting a session of the protocol, and `nr` stands for *nonce received*).

The program counter `pc` of an agent can take the following values (these values are arbitrary symbols and prefixed by a backquote), according to the role: `‘REQ`, `‘AUTH` and `‘FINISHED` for Alice and `‘CHAL`, `‘WAIT` and `‘FINISHED` for Bob. For Alice, `pc` = `‘REQ` means that the agent is about to send the message with the corresponding label in the protocol specified in section 3.1, and similarly for `pc` = `‘AUTH` (role Alice) and `‘CHAL` (Bob). For an agent playing the role of Bob, `pc` = `‘WAIT` means that he is waiting for the answer of Alice to his challenge `CHAL`, and `pc` = `‘FINISHED` means that the agent has completed his session of the protocol.

Messages. Three different kinds of messages are exchanged between Alice and Bob during the protocol. We define a predicate to recognize each kind of messages: `REQ`, `CHAL` and `AUTH`. Messages are also records and they are characterized by the kind of information that they hold. For instance, messages of type `REQ` contain a field `na` representing the content of the message and a field `kb` which is the public key used for encryption. For the sake of simplicity, in our program, every public key or private key is represented by the identity of the owner.

Intruder Knowledge. The knowledge of the intruder is also represented by records with fields `name`, `nonce`, `pub`, `priv`. We define several predicates (`info_name`, `info_nonce`, `info_pub` and `info_priv`) for each kind of information that the intruder will be able to reveal from the whole history of exchanged messages: name, nonce, public key and private key. These predicates are used to determine the presence of a message of a given kind with a given information in the solution.

4.2 The Intruder Transformation Rules

The network is common to all agents and the intruder, hence the latter is able to read and produce new messages. This behavior is implemented by the rules presented in the two following sections (the `MGS` code for this section can be found in Appendix B.2).

Reading and Analyzing Messages. In our approach, the existing messages are read by the intruder from the current state and they are put back unchanged. Moreover, the encrypted contents of a message are added as new known information to the state if decryption is possible. More precisely, the intruder can learn a plaintext encrypted with a public key (for instance the nonce `nb` encrypted with `kb` in message `AUTH`) only if he knows the corresponding private key.

The following three rules define the evolution of the knowledge of the intruder, according to the messages present in the network. There is exactly one

rule for each kind of message. They will actually not generate all the information that the intruder can extract from collected message. However, these transformations are sufficient to extract all the information needed to built messages with the forging rules below. For instance, if a message m present in the solution has type `REQ`, and the intruder knows the private key associated to $m.\mathtt{kb}$, then he learns the components $m.\mathtt{na}$ and $m.\mathtt{a}$ of m. Theoretically, he also learns the pair $(m.\mathtt{na}, m.\mathtt{a})$ but storing such an information is useless since we assume that the intruder is able to build pairs arbitrarily.

$$m \longrightarrow m, \{\mathtt{nonce} = m.\mathtt{na}\}, \{\mathtt{name} = m.\mathtt{a}\} \quad \mathtt{where}\ m \in \mathtt{REQ} \wedge \exists k \in \text{self s.t. } k.\mathtt{priv} = m.\mathtt{kb}$$

$$m \longrightarrow m, \{\mathtt{nonce} = m.\mathtt{na}\}, \{\mathtt{nonce} = m.\mathtt{nb}\} \quad \mathtt{where}\ m \in \mathtt{CHAL} \wedge \exists k \in \text{self s.t. } k.\mathtt{priv} = m.\mathtt{ka}$$

$$m \longrightarrow m, \{\mathtt{nonce} = m.\mathtt{nb}\} \quad \mathtt{where}\ m \in \mathtt{AUTH} \wedge \exists k \in \text{self s.t. } k.\mathtt{priv} = m.\mathtt{kb}$$

The keyword "self" used in the rules denotes the current multiset (i.e., the multiset from which m is chosen). The existential quantifier in the guard of the rules checks that some condition is satisfied by an element k in a given multiset: such kind of predicate is easily computed by set of reduction rules.

Forging Some New Messages. In the previous section, we have describe the `intruder` rules set which only reveals information according to already known messages an keys. The following rule *produces* a new fake `REQ` message from know information in the solution:

$$k, n, m \longrightarrow \{\mathtt{na} = m.\mathtt{nonce}, \mathtt{a} = n.\mathtt{name}, \mathtt{kb} = k.\mathtt{pub}\} \quad \mathtt{where\ info_pub}(k) \wedge \mathtt{info_name}(n) \wedge \mathtt{info_nonce}(m)$$

There is one such rule for the two other kinds of messages `CHAL` and `AUTH`. These rules are used to produce by saturation (fixed point computation) all possible fake message that can be forged from the known facts in a multiset.

An attack consists in revealing all possible information using the above rules of the intruder after having forged all possible fake messages. Actually, we'll see in the following that a *real* attack always consists in the application of the attack rules of Section 4.3 until a fixed point is reached.

4.3 *Nested* Multiset Rewriting to Explore the State Space

The first idea to implement the logical analysis of NSPK is to aggregate all the entities involved into the protocol in a single multiset acting as a chemical solution containing the agents, the messages and the revealed information. The agents and the intruder will react with messages to augment the solution with new information. All information are in the solution at the same level. An attack on the NSPK protocol consists here in finding an interleaving of

the agents actions described below such that Bob's nonce is revealed (the `MGS` code corresponding to this section can be found in Appendix B.3).

This approach suffers from the following problem: let S be a solution and a be an agent in a state where he might reply to two different messages m_1 and m_2. The two following scenarios could happen:

1. The agent replies to both messages: to m_1 to give m'_1 and to m_2 to give m'_2. Here, after the agent action, S becomes $S \cup m'_1 \cup m'_2$. In the future evolution of the protocol, another agent may react to *both* m'_1 *and* m'_2 leading to an incorrect situation, even where the intruder may break the protocol and reveal the nonce.
2. The agent replies to only one of the two messages: to m_i to produce m'_i. In that case, an attack might not be found because the case where the reply should have concerned the other message has not been considered. The protocol analysis is therefore too weak.

The consequence is that we have to take into account the different evolutions of the protocol that might happen when an agent receives more than one message. To model such a situation, we make use of several multisets (membranes) to localize the computation and to avoid the (possible) interferences. The initial state consists in a multiset of multisets. Each element in the top multiset (the *skin* in the language of P systems) is a possible state in the protocol and represent some possible evolution, as depicted in Figure 3.

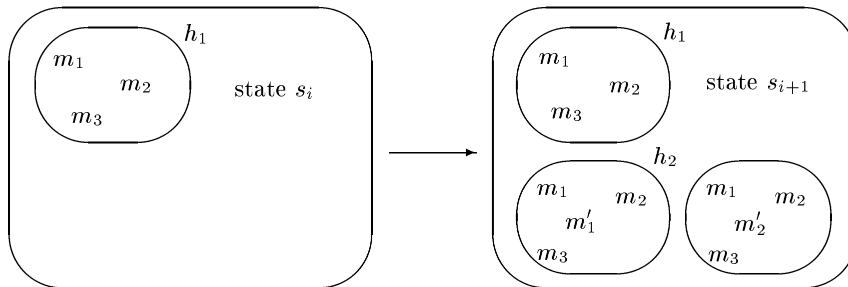

Fig. 3. Creations of membranes.

The Agents. The behavior of each agent, at each possible `pc`, is described by a set of rules. For example, the behavior of `Alice` with `pc` = `‘AUTH` is to switch to the state `‘AUTH` and to produce a new message:

$$x, t \longrightarrow (x + \{\texttt{pc} = \texttt{`AUTH}\}), \{\texttt{kb} = x.\texttt{dest}, \texttt{na} = x.\texttt{ni}, \texttt{a} = x.\texttt{id}\}$$
$$\texttt{where}\ \ x \in \texttt{REQ} \wedge x.\texttt{id} = \texttt{alice} \wedge t = \texttt{`OK}$$

the operator + is the asymmetric merge of records and the results of $x + \{\texttt{pc} = \texttt{`AUTH}\}$ is a record equal to $x$ except for the field `pc` that takes the value `‘AUTH`.

The variable t matches a symbol used to inhibit or activate the rule: if the symbol is present (e.g., $t =$ `'OK`) then the rule can be triggered. If the symbol `'OK` is not present in the chemical solution, the rule is inhibited.

There are three additional similar rules to describe the evolution of Alice waiting for the authentication, Bob waiting for a challenge and Bob in the finishing state.

Note that the messages addressed to an agent must not be removed from the solution and are available for other rule applications.

The Initial State. The initial state for the attack search consists in a multiset (of multisets) with only one element:

- the two agents, Alice and Bob, initialized with their respective identity, the destination of the message for Alice, initial nonces to arbitrary integer values, program counter,
- intruder knowledge (public keys for all participants and its own private key).

Looking for an Attack. In our definition of the initial state, the number of agents is fixed and remains such. Therefore, the number of execution steps is bounded accordingly. The problem consists in finding the correct interleaving of Alice and Bob actions leading to a successful attack.

The basic idea is to generate all strings of bounded length made of four symbols representing an evolution of one of the agent (see the rule set of an agent described above). The combinatorial generation of such string is easy and can be done randomly. Then a rule is used to trigger the "application" one of such string to an agent to make this agent evolve:

$$m, \text{`}\mathtt{alice_req::}s \longrightarrow m, \text{`}\mathtt{OK}, s$$

The expression `'alice_req::`s denotes a string beginning with the symbol `'Alice_req`. Note that the tail s of the string is released in the solution. The production of the triggering symbol `'OK` activates the evolution rule on m. By adjoining a trigger to this rule, which is released by the agent evolution rule and consumed by this rule application, we can interleave correctly the evolution of an agent until the exhaustion of the string s.

We still look for an interleaving leading to revealing the nonce. A successful attack is to find in the chemical solution the nonce of Bob revealed. This is done by adding a specific rule, e.g., a rule leading to a dissolution of all enclosing membranes.

Validation in the `MGS` Programming Language. To validate our propositions, we have completely implemented and validated several versions of the logical analysis using the `MGS` programming language. A presentation of `MGS` and the commented code can be found in the Appendix.

`MGS` is a research project devoted to the design and the development of a programming language dedicated to the simulation of biological pro-

cesses [9, 11]. Based on topological notions, `MGS` supports the notion of transformation: a localized computation specified by rules. One can for example defines multiset rewriting rules [1] that act on a nest of multisets (i.e., membranes). These rules can be used to move values from a multiset to another one, as well as to dissolve, divide or create new multisets. So, `MGS` can potentially be used to process membranes. However, we outline that the `MGS` project focuses on the design of a programming language rather than the development of a well founded computational model.

5 Summary

In this chapter, we have used the membrane computing approach to describe and analyze the NSPK protocol. This application of membrane computing is new to the best of our knowledge. It has been shown that using our approach, the well–know security hole of [16] is easily (in less than one second) discovered by our state exploration procedure.

In the proposed version, we are searching for the correct interleaving of the agents actions leading to a possible attack. Using membranes permits us to handle correctly the fact that an agent may have to react to more than one message leading to more than one evolution of the state.

Nevertheless, this method is tailored for the search of an interleaving of agents actions leading to the revelation of the nonce. This is possible because we actually know that such an interleaving *will* lead to a successful attack. We have proposed in [19] a more general approach where a full state space search is done. The complete running code of the two versions have been implemented in `MGS` and is detailed in [20]. The complete code is particularly simple and readable. Moreover, it is also easy to evolve the initial analysis to more sophisticated ones.

The approach presented here has been developed for this special protocol and heavily relies on the nesting of membranes to localize the computation and to avoid evolution interference leading to more approximate analysis. We believe that the principles of our modeling are general enough to envision a systematic way to derive a program for searching attacks from an abstract description of the messages of a protocol given with the notations of Section 3.1, following [14].

Acknowledgments. The authors are grateful to Jean–Louis Giavitto, Julien Cohen and Antoine Spicher at LaMI for stimulating discussions and thoughtful remarks. This research is supported in part by the CNRS, the GDR ALP, the University of Évry, Genopole©, INRIA and ENS Cachan, the RNTL project PROUVÉ and the ACI-SI Rossignol.

References

1. J.–P. Banâtre, P. Fradet, D. Le Métayer: Gamma and the Chemical Reaction Model: Fifteen Years After. *Lecture Notes in Computer Science* 2235, Springer, Berlin, 2001, 17–44.
2. P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, M. Vittek: ELAN – A Logical Framework Based on Computational Systems. *Electronic Notes in Theoretical Computer Science*, 4 (1996).
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí–Oliet, J. Meseguer, J.F. Quesada: The Maude System. *Lecture Notes in Computer Science* 1631, Springer, Berlin, 1999, 240–243
4. I. Cervesato, N. Durgin, P.D. Lincoln, J.C. Mitchell, A. Scedrov: A Meta–Notation for Protocol Analysis. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW1999)*, Mordano, Italy, 55–69.
5. H. Cirstea: Specifying Authentication Protocols Using ELAN. In *Workshop on Modeling and Verification*, 1999.
6. D.L. Dill, A.J. Drexler, A.J. Hu, C.H. Yang: Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design, VLSI in Computers and Processors (ICCD1992)*, 522–525, Los Alamitos, Ca., USA, 522–525.
7. G. Denker, J. Meseguer, C. Talcott: Protocol Specification and Analysis in Maude. In *Workshop on Formal Methods and Security Protocols*, 1998.
8. D. Dolev, A. Yao: On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, IT-29, 2 (1983), 198–208.
9. J.–L. Giavitto: Topological Collections, Transformations and Their Application to the Modeling and the Simulation of Dynamical Systems. In *Rewriting Technics and Applications (RTA'03)*, *Lecture Notes in Computer Science* 2706, Springer, Berlin, 2003, 208–233.
10. J.–L. Giavitto, O. Michel: The Topological Structures of Membrane Computing. *Fundamenta Informaticae*, 49 (2002), 107–129.
11. J.–L. Giavitto, G. Malcolm, O. Michel: Rewriting Systems and the Modeling of Biological Systems. *Comparative and Functional Genomics*, 5 (2004), 95–99.
12. S. Peyton Jones, C. Hall, K. Hammond, W. Partain, P. Wadler: The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology Technical Conference*, 1993.
13. A. Huima: Efficient Infinite–State Analysis of Security Protocols. In *Proceedings of FLOC'99 Workshop on Formal Methods and Security Protocols*, 1999.
14. F. Jacquemard, M. Rusinowitch, L. Vigneron: Compiling and Verifying Security Protocols. In *Logic for Programming and Automated Reasoning (LPAR'00)*, *Lecture Notes in Computer Science* 1955, Springer, Berlin, 2000.
15. X. Leroy: The Objective CAML System, Release 3.07. Documentation and User's Manual. Technical report, INRIA, 2004.
16. G. Lowe: An Attack on the Needham–Schroeder Public Key Authentication Protocol. *Information Processing Letters*, 56, 3 (1995).
17. C.A. Meadows: The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26, 2 (1995), 113–131.
18. J.K. Millen, S.C. Clark, S.B. Freedman: The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, SE-13, 2 (1987).
19. O. Michel, F. Jacquemard: An Analysis of the Needham–Schroeder Public Key Protocol with MGS. In *Fifth Workshop on Membrane Computing (WMC5)*, Milano, 2004, 295–315.

20. O. Michel, F. Jacquemard, J.–L. Giavitto: Three Variations on the Analysis of the Needham–Schroeder Public Key Protocol with MGS. Technical Report LaMI-98-2004, Univ. d'Évry - CNRS, 2004 – 25 pages.
21. J. Mitchell, M. Mitchell, U. Stern: Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1997, 141–151.
22. R.M. Needham, M.D. Schroeder: Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21, 12 (1978), 993–999.
23. Gh. Păun: *Membrane Computing. An Introduction.* Springer, Berlin, 2002.
24. M. Rusinowitch, M. Turuani: Protocol Insecurity with Finite Number of Sessions is NP-Complete. In *Proceedings of the 14th Computer Security Foundations Workshop (CSFW2001)*, 174–190.
25. C. Weidenbach: Towards an Automatic Analysis of Security Protocols in First–Order Logic. *Lecture Notes in Computer Science* 1632, Springer, Berlin, 1999, 378–382.

Appendix A. A Brief Introduction to the `MGS` Language

We briefly present in this section the `MGS` language. We do not detail all the features of the language but we rather focus on the notions required to understand the next section.

A.1 `MGS` as a Functional Language

`MGS` embeds a complete, impure, dynamically typed, strict, functional language. We only describe here the major differences between the constructions available in `MGS` with respect to functional languages like `OCAML` [15] or `HASKELL` [12].

Values. Atomic values (like integers, floats, booleans, strings,...) with their usual functions, are available. Constants are denoted with a backquote: `‘REQ` (they are reminiscent of `LISP` symbols). The only operations allowed on a constant is to store it or to compare it for equality with another value.

Records (cartesian products with labels) are defined using braces: `{x=0, y=1}` creates a pair with label `x` and `y` (`MGS` record are similar to `Pascal`'s record or `C`'s struct). The fields are accessible using the dot notation: `let v = {x=0, y=1} in v.x` has value `0`. Since records are used in `MGS` to define a particular state of an entity, `MGS` allows the definition of predicates based upon the fields found in a record. The keyword `record` is used to define such predicates:

```
record agent = \{id, ni, nr, pc\}
```

defines the predicate `agent` that holds only if applied on a record value that has at least all the fields `id, ni, nr` and `pc`. Record `alice` defined as `record alice = {dest} + agent` extends predicate `agent` with the additionally required field `dest`. So far, the record predicates only required to have the fields to hold. The predicate `req` defined as `record req = {pc = ‘REQ}` holds only if its argument has a field `pc` with a value equal to the constant `‘REQ`.

Imperative Variables and Sequencing. Variables in a functional languages are not true variables: they refer to values and cannot be updated. `MGS` has a notion of *imperative* variable (also called *mutables*) that can be updated. The `:=` operator allows to define such variables. For example `imp := 0` defines `imp` with value `0` that can be later updated with the same construction.

The semi column operator `;` is used to express the sequencing of expressions: the value of `f();g()` is the value returned by `g()` but `f()` has been computed before.

Functions. Since `MGS` is a functional language, it has functions as first–class values. Functions are defined either using the construction `fun` like in `fun max(x, y) = if (x > y) then x else y fi` or using the classical lambda notation as in `\x.\y.if (x > y) then x else y fi`

Computations by fixpoints are heavily used in applications like simulations or state space explorations. `MGS` provides an operator to compute iterations and fixpoints of functions. Let `f` be a function, then `f[iter = n](x)` computes `f^n(x)` and `f[*](x)` denotes the fixpoint of `f` starting from `x`.

Functions together with mutables and iterations allows to define functions that pass informations between calls. For example, function `f` defined as `fun f[acc=0](x)=(acc := acc+1; x+acc)` allows to define an accumulator `acc` which stores a value that is incremented between each call. The value of `f['iter = 10, acc = 0](1)` is 56.

A.2 Topological Collections and their Transformations

The distinctive features of the `MGS` language is its handling of entities structured by *abstract topologies* using *transformations* [10]. A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation inducing a notion of *sub–collection*. A sub–collection B of a collection A is a subset of connected elements of A and inheriting its organization from A.

Collection Types. Many different predefined and user–defined collection types are available in `MGS`. We won't describe them here since sets, multisets and sequences are the only collection type used in this chapter.

For any collection type `T`, the corresponding empty collection is written `():T`. The name of a collection type is also a predicate used to test if a value is of this type: `T`(v) holds only if v is of type `T`. Each collection type can be subtyped. The type declaration `collection U = T` introduces a new collection type `U` which is a subtype of `T`. The new type `U` shares the same topology as `T`. However, a value of type `U` can be distinguished from a value of type `T` using the `U` predicate (i.e., the subtyping relation implies that `U`$(u) \Rightarrow$ `T`(u), for any value u, but not the reverse). Elements in a collection can be of any type, including collections.

Operations on Collections. The join of two collections C_1 and C_2 (written by a comma: C_1`,`C_2) is the main operation on collections. The comma

operator is overloaded in `MGS` and can be used to build any collection (the type of the arguments disambiguates the collection built). So, the expression `1, 1+1, 2+1, ():set` builds the set with the three elements $1, 2$ and 3, while the expression `1, 1+1, 2+1, ():bag` makes a multiset with the same three elements.

Transformations. The *global transformation* of a topological collection C consists in the *parallel application* of a set of *local transformations.* A local transformation is specified by a rewriting rule r that specifies the replacement of a sub–collection by another one. The application of a rewriting rule $\beta \Rightarrow f(\beta, ...)$ to a collection A:

1. selects a sub–collection B of A whose elements match the *pattern* β,
2. computes a new collection C as a function f of B and its neighbors,
3. and specifies the insertion of C in place of B into A.

One should pay attention to the fact that, due to the parallel application strategy of rules, *all distinct instances B_i of the sub–collections matched by the β pattern are "simultaneously replaced" by the $f(B_i)$.* This is very different from the evaluation strategies followed by classical rewriting tools like MAUDE [3], ELAN [2], Murφ [6], MSR [4], etc.

The `MGS` experimental programming language implements the idea of transformations of topological collection into the framework of a simple dynamically typed functional language. Collections are just new kind of values and transformations are functions acting on collections and defined by a specific syntax using rules. Transformations (like functions) are first–class values and can be passed as arguments or returned as the result of an application.

Sub–collection Patterns. A transformation is defined by a set of rules (listed between braces). A pattern β that appears in the left hand side of a rule is an expression used to select a sub–collection to be replaced. Several operators are available; we will review here only few of them:

- **Literal**: a literal value matches an element with the same value. For example, `123` matches an element with the integer value 123.
- **Variable**: a pattern variable a matches exactly one element. The variable a can then occur elsewhere in the rest of the rule and denotes the value of the matched element. The identifier of a pattern variable can be used only once in a pattern. To match an element without giving it a name, an underscore _ can be used.
- **Alias**: the pattern p `as` X associates the variable X to the value matched by the pattern p. X is a regular variable than can be used as previously described.
- **Neighbor**: the pattern b `,` p matches a sub–collection composed of an element matched by b neighbor of a sub–collection matched by p.
- **Guard**: p/exp matches a sub–collection matched by p such that the predicate exp hold. For instance, x `,` y `/` y `>` x matches two neighbor elements such that the second one is greater than the first one.

- **Repetition**: p* matches a sub–collection made of a (possibly empty) repetition of sub–collections matched by p. If p is a pattern variable, then its value refers to the sequence of matched elements and not to one of the individual values. For example, `3+` matches a non-empty sub–collection made only of `3`'s.

`MGS` and Membrane Computing. The `MGS` language enables a kind of membrane computing. It embeds the rewriting of multisets (or sets) in the following way: in a multiset, an element is susceptible to interact with any other element, so the abstract topology of a multiset is the topology of a complete connected graph: the neighbors of an element are all the other elements in the multiset. Then, a pattern β can select an arbitrary sub–multiset and a multiset rewriting rule is simply a local transformation in this topology.

A.3 Example: Computing all the n-tuple in a Set

Let S be a set of values. To compute all the n-tuples one can use the transformation:

```
trans n_tuple[acc, n] = {
  (_* as c) / size(c) == n / (acc := c::acc; false) => !!(0);
  _                                                 => return(acc)
}
```

In transformation `n_tuple`, parameters `acc` and `n` are mutables whose definition are local to the transformation. They are set at the first call of the transformation. Applied to a collection C, pattern of the first rule `(_* as X) / size(X) == n)` matches a sub–collection c of C of size n such that all elements of c are neighbors (with respect to the topology induced by C). Once c is found, predicate `(acc := c::acc; false)` is calculated: collection c is added to the accumulator (`::` is the concatenation of a value to a collection) and the value `false` is returned. Since the predicate does not hold, the right hand side of the rule is not evaluated (the expression `!!(0)` aborts the program) and the rule is tried against another instance, storing each time the solution of the matching into the accumulator. Once all the possibilities have been tried and failed, the second rule is tried. That rule succeeds in matching anything and returns the value of the accumulator. Transformation `n_tuple[acc=set:(), n=2]((3,4,5,6,set:()));;` computes all the pairs

```
((3, 4):'seq, (3, 5):'seq, (3, 6):'seq, (4, 3):'seq, (4, 5):'seq,
(4, 6):'seq, (5, 3):'seq, (5, 4):'seq, (5, 6):'seq, (6, 3):'seq,
(6, 4):'seq, (6, 5):'seq):'set
```

where `(3, 4):'seq` is a pair holding the two integers value.

Appendix B. `MGS` Code for the Description of the Attack

We give in the following sections the `MGS` code that implements the search for an attack that is described in Section 4.

B.1 Representing Agents, Messages and Intruder Knowledge

The code presented in this section implements the data structures defined in Section 4.1.

Agents. One set of records is used to define *agents*, defined as:

```
record agent    = { id, ni, nr, pc};;
record alice    = { dest } + agent;;
record bob      = agent;;
```

Some records for the the various possible agent `pc` are defined as follows:

```
record req      = { pc = 'REQ };;
record chal     = { pc = 'CHAL };;
record auth     = { pc = 'AUTH };;
record wait     = { pc = 'WAIT };;
record finished = { pc = 'FINISHED };;
```

Messages. A predicate is defined for each kind of message:

```
record messageReq  = { na, a, kb  };;
record messageChal = { na, nb, ka };;
record messageAuth = { nb, kb     };;
```

Intruder Knowledge. Finally, we define a predicate for each kind of information that the intruder will be able to reveal from the whole history of exchanged messages:

```
record info_name  = { name  };;
record info_nonce = { nonce };;
record info_pub   = { pub   };;
record info_priv  = { priv  };;
```

Predicates are defined for each kind of message to determine the presence of a message of a given kind in the solution:

```
fun messageReqCond(a, m)  = messageReq(m) & (m.kb == a.id);;
fun messageChalCond(a, m) = messageChal(m) & (m.ka == a.id)
                            & (m.na == a.ni);;
fun messageAuthCond(a, m) = messageAuth(m) & (m.kb == a.id)
                            & (m.nb == a.ni);;
fun PmessageReq(b, all)   = exists(messageReqCond(b), all);;
fun PmessageChal(a, all)  = exists(messageChalCond(a), all);;
fun PmessageAuth(a, all)  = exists(messageChalCond(a), all);;
```

B.2 The Intruder Transformation Rules

The intruder's behaviour described in Section 4.2 is defined here in terms of `MGS` transformations.

Reading and Analyzing Messages. The following transformation rules define the evolution of the knowledge of the intruder, according to the messages present in the network :

```
trans intruder = {
  m / messageReq(m)  & exists((\k.(info_priv(k)
                     & (k.priv == m.kb))), neighbors(m))
  => m, {nonce = m.na}, {name = m.a};
  m / messageChal(m) & exists((\k.(info_priv(k)
                     & (k.priv == m.ka))), neighbors(m))
  => m, {nonce = m.na}, {nonce = m.nb};
  m / messageAuth(m) & exists((\k.(info_priv(k)
                     & (k.priv == m.kb))), neighbors(m))
  => m, {nonce = m.nb}
};;
```

The function neighbors used in the transformation is a special form that returns all the neighbors of the element denoted by a pattern variable.

Forging Some New Messages. In the previous section, we have described the intruder transformation which only reveals information according to already known messages an keys. The following transformation *produces* new fake messages from know informations in the solution. There is one transformation for each kind of message:

```
trans forge_req[acc = set:()] =
{
    ((k:info_pub), (n:info_name), (m:info_nonce)) as X
    / acc := {na = m.nonce, a = n.name, kb = k.pub},acc; false
    => !!(0);
    _  => return(acc)
};;
trans forge_chal[acc = set:()] =
{
    ((k:info_pub), (n:info_nonce), (m:info_nonce)) as X
    / acc := {na=m.nonce, nb=n.nonce, ka=k.pub},
             {nb=m.nonce, na=n.nonce, ka=k.pub},acc; false
    => !!(0);
    _ => return(acc)
};;
trans forge_auth[acc = set:()] =
{
    ((k:info_pub), (m:info_nonce)) as X
    / acc := {nb=m.nonce, kb=k.pub}, acc; false
    => !!(0);
    _ => return(acc)
};;
fun forge(s)  =
    s, forge_req[acc=set:()](s), forge_chal[acc=set:()](s),
    forge_auth[acc=set:()](s);;
fun attack(s) = intruder(forge(s));;
```

Consider the first transformation: one should remark that, since the record made of info_pub, info_name and info_nonce might not be unique, we have

to use the same kind of procedure described in Section A.3 to produce *all* matching triple. This way, we produce all possible fake messages knowing public keys, names of agents involved in the sessions and revealed nonces.

Function `forge`, applied to the solution `s` adds to the original solution the result of the application of the three `forge` transformations.

An attack, described by the `attack` consists in the revealing of all possibles informations by the `intruder` after having forged all possible fake messages.

B.3 Nested Multiset Rewriting A new collection type is defined: `membrane` which derives from the collection type `seq` (`membrane` is then just a sequence with a different name). The empty collection of that kind is `():membrane`.

```
collection membrane = seq;;
```

The Agents. The transformations describing the behavior of each agent are described below:

```
trans alice_req = {
   x / (req(x) & alice(x))  => (x + {pc = ‘AUTH}),
                               {kb = x.dest, na = x.ni, a = x.id}
};;
trans bob_chal = {
   y / bob(y) & chal(y) & PmessageReq(y, neighbors(y))
   => let all_messages = filter(messageReqCond(y), neighbors(y))
      in return(map((\m.((y + {pc = ‘WAIT, nr = m.na}),
                         {ka = m.a, na = m.na, nb = y.ni},
                         setify(neighbors(y)))), all_messages))
};;
trans alice_auth = {
 x / auth(x) & alice(x) & PmessageChal(x, neighbors(x))
 => let all_messages = filter(messageChalCond(x), neighbors(x))
    in return(map((\m.((x + {pc = ‘FINISHED}),
                          {kb = x.dest, nb = m.nb},
                          setify(neighbors(x)))), all_messages))
};;
trans bob_finish = {
 y / bob(y) & wait(y) & PmessageAuth(y, neighbors(y))
 => let all_messages = filter(messageAuthCond(y), neighbors(y))
    in return(map((\m.((y + {pc = ‘FINISHED}),
                  setify(neighbors(y)))),
                  all_messages))
};;
```

Notice that the messages addressed to Alice are not removed from the solution. Since they do not appear in the pattern part of the rule, they are not matched and therefore not "consumed" from the solution.

Care has been taken in the previous transformations to generate the correct membrane structure (this is the `setify(neighbors(y))` argument in the `map` of the r.h.s. of each transformation; `setify` computes the set of elements of

its collection argument and the function `neighbors` returns all the neighbors of the element denoted by a pattern variable).

Revealing a Successful Attack. A successful attack is to find in the chemical solution the nonce of Bob revealed. Since we have a membrane of sets, revealing a successful attack consists in looking in each set if the nonce is revealed:

```
fun isbroken(x) = member({nonce = 1}, x);;
fun broken(x)   = exists(isbroken, x);;
```

The Initial State. The initial state is a membrane of sets with only one set:

```
initial := ({id = "alice", ni = 0, nr, pc = ‘REQ, dest = "charly",},
           {id = "bob",   ni = 1, nr, pc = ‘CHAL},
           {priv = "charly"}, {pub = "charly"}, {pub = "alice"},
           {pub = "bob"}, set:()
          ):: membrane:();;
```

Remark that the `nr` field is not set in the definitions: in this case, it is defined with an undefined value (and will later be set to a relevant value once a message is received).

Looking for an Attack. As stated in Section 4.3, the problem consists in finding the correct interleaving of Alice and Bob actions leading to a successful attack. Transformation `breaks` succeeds if such an interleaving exists. It is applied on `functions` which is the set of the transformations describing the agents behavior. The `MGS` pattern expression `(_*) as F` will match all possible permutations of the elements of `functions`. For the sake of explanation, let `F` be the sequence $[f_1, ..., f_n]$ of *one possible permutation*. The guard checks whether `broken` holds for an attack on the state $\texttt{attack}^* \circ f_1 \circ ... \circ \texttt{attack}^* \circ f_n(\texttt{initial})$.

As for the search of an attack, we still look for an interleaving leading to revealing the nonce. We now have to `map` and `flatten` the attack that follows an action of one of the agents:

```
fun fmap(f, e) = flatten(map(f, e));;
trans break = {
   (_*) as F / broken(fold((\fn.\s.(fmap(attack[*], fmap(fn,s)))),
                           initial, F))
     => return(true)
};;
functions := alice_req, alice_auth, bob_chal, bob_finish, set:();;
successful := break(functions);;
```

The search for an attack succeeds in less than a second on a `AMD-1.4Ghz Linux Debian/Woody` computer, and reveals that the correct interleaving of functions is, as expected, `bob_finish`∘`alice_auth`∘`bob_chal`∘`alice_req`. The implemented code and the `MGS` interpreter are available from URL `mgs.lami.univ-evry.fr` .