Executable Specification of Cryptofraglets in Maude for Security Verification

Fabio Martinelli and Marinella Petrocchi

IIT-CNR, Pisa, Italy

Abstract. Fraglets are computation fragments flowing through a computer network. They implement a chemical reaction model where computations are carried out by having fraglets react with each other. The strong connection between their way of transforming and reacting and some formal rewriting system makes a fraglet program amenable to verification. Starting from a threat model which we intend to use for modeling secure communication protocols with fraglets, we propose an executable specification of fraglets (and fraglets-based cryptographic protocols) in the rewriting logic-based Maude interpreter.

Keywords: Fraglets, cryptofraglets, threat model, security protocols, Maude.

1 Introduction

Fraglets [22,23,24,25] represent an execution model for communication protocols that resembles the chemical reactions in living organisms. It was originally proposed for making automatic the whole process of protocol development, involving the various phases of design, implementation and deployment. Fields of applications have been protocol resilience and genetic programming experiments.

The fraglets model has been adopted in the BIONETS EU project [2] and some security and trust extensions to the original model have become necessary to make it a running framework. Thus, in the past few years, fraglets have been extended i) with instructions for symmetric cryptography [21]; ii) with access control mechanisms [14]; iii) with dedicated primitives for trust management [15]. This work has been mainly done with the intent to use fraglets for modeling security protocols and verifying security properties within BIONETS. With an eye on verification, an encoding from the programming language defined for fraglets and the MultiSet Rewriting formal rules [3] has been shown in [21], as a first brick to give a formal semantics to fraglets, the starting point for developing verification tools.

It was indeed the similarity between the fraglets programming language and the rewriting systems, combined with the need of modeling and verifying security protocols for the bio-inspired networks BIONETS, that lead us to consider the executable specification language Maude [5,16], based on rewriting logic [18]. Maude offers useful advantages for formalizing and verifying security communication protocols (see, *e.g.*, [1,6]. First, its efficient executability allows both prototyping and debugging of protocols specifications. Furthermore, since a concurrent system can have many different behaviours, exploring a single execution could not be sufficient to prove security properties of the system. Maude supports ad hoc defined strategies for exploring all the execution of a system.

The original contributions of the paper are the following. First, we define a threat model for fraglets by adding an adversary to the fraglet model of a secure communication network. Second, we present our executable specification of fraglets in Maude. Third, we show how to specify and execute a fraglets-based instantiation of the Needham-Schroeder Public Key protocol (NSPK). A security verification with respect to message secrecy is carried out by means of Maude built-in commands. The overall goal is the achievement of a general fraglets framework for the modeling and verification of security issues in communication protocols.

The paper is organized as follows. Section 2 recalls the fraglets model and introduces a refined version of cryptofraglets. Section 3 defines a threat model for cryptofraglets, by discussing the capabilities of an adversary that is going to subvert the standard operations in a secure fraglet network. We introduce the notion of the adversary's knowledge and we define the secrecy property for fraglets. Section 4 presents the executable specification of cryptofraglets in Maude, according to the defined threat model. Section 5 shows a fraglets-based instantiation of two interleaved sessions of NSPK. Finally, Section 6 gives some final remarks and discusses current limitations and future goals.

2 Fraglets

A fraglet is denoted as $[s1\ s2\ \ldots tail]$, where si $(1 \leq i \leq n)$ is a symbol and $tail$ is a (possibly empty) sequence of symbols. Nodes of a communication network may process fraglets as follows. Each node maintains a fraglet store to which incoming fraglets are added. Fraglets may be processed only within a store. The $send$ operation transfers a fraglet from a source store to a destination store.

Fraglets are processed through a simple prefix programming language. Transformation instructions involve a single fraglet, while reactions involve two fraglets. Table 1 shows the fraglets core instructions. The interested reader can find the comprehensive tutorial on [10].

Table 1. The set of fraglets core instructions

match	[match t tail1], [t tail2] $\longrightarrow$ [tail1 tail2]
matchp	[matchp t tail1], [t tail2] $\longrightarrow$ [matchp t tail1], [tail1 tail2]
send	S_A[send B tail] $\longrightarrow$ S_B[tail]
nop	[nop tail] $\longrightarrow$ [tail]
nul	[nul tail] $\longrightarrow$ []
dup	[dup t a tail] $\longrightarrow$ [t a a tail]
exch	[exch t a b tail] $\longrightarrow$ [t b a tail]
fork	[fork a b tail] $\longrightarrow$ [a tail], [b tail]
pop2	[pop2 h t a b tail] $\longrightarrow$ [h a], [t b tail]
split	[split seq1 * seq2] $\longrightarrow$ [seq1], [seq2]

Two fraglets react by instruction $match$, and their tails are concatenated. With the catalytic $matchp$, the reaction rule persists. Instruction $send$ performs a communication between fraglets stores. It transfers a fraglet from store $\mathcal{S}_A$ to store $\mathcal{S}_B$. Notation ${}_{\mathcal{S}_A}[s1\ s2\ \ldots\ tail]$ denotes that the fraglet is located at $\mathcal{S}_A$. The name of the destination store is given by the second symbol in the original fraglet $[send\ \mathcal{S}_B\ tail]$. Where not strictly necessary, we omit to make the name of the store explicit.

Instruction nop does nothing, except consuming the instruction tag. Instruction nul destroys a fraglet. Finally, there are a set of transformation rules that perform symbol manipulation, like duplicating a symbol (dup), swapping two tags ($exch$), copying the tail and prepending different header symbols ($fork$), popping the head element a out of a list $a\ b\ tail$ ($pop2$), and finally breaking a fraglet into two at the first occurrence of symbol $*$ ($split$).

In [21,14], we proposed the cryptofraglets, which extend the fraglets programming language with basic cryptographic instructions. In defining cryptofraglets, we abstract from the cryptographic details concerning the operations by which they can be encrypted, decrypted, hashed, *etc.*. We make the so called *perfect cryptography assumption* and we consider encryption as a black box: an encrypted symbol, or sequence of symbols, cannot be correctly learnt unless with the right decryption key. This approach is standard in (most of) the analysis of cryptographic communication protocols, see, *e.g.*, [4,9,11,13].

Here, we give a slightly modified version of the crypto-instructions, originally seen as reaction rules, and here rephrased to transformation rules, for a more convenient specification in Maude (see Section 4).

Table 2. Crypto-instructions for encryption, decryption, and hashing

enc	[enc newtag k_1 tail] $\rightarrow$ [newtag $tail_{k_1}$]
dec	[dec newtag k_2 $tail_{k_1}$] $\rightarrow$ [newtag tail]
hash	[hash newtag tail] $\rightarrow$ [newtag h(tail)]

The encryption instruction takes as input the fraglet $[enc\ newtag\ k_1\ tail]$, consisting of the reserved instruction tag enc, an auxiliary tag $newtag$, the encryption key k_1, and a generic sequence of symbols $tail$, representing the meaningful payload to be encrypted. It returns the fraglet $[newtag\ tail_{k_1}]$, with the auxiliary tag and the cyphertext $tail_{k_1}$. Decryption and hashing rules can be similarly explained. Note that, since fraglets processing is through matching tags, the presence of either a reserved instruction tag or an auxiliary tag as the leftmost symbol is necessary for the computation to go on.

We set $k_1 = k_2 = k$ when dealing with shared-key cryptography, and we consider a pair of public/private keys (pk, sk) when dealing with asymmetric cryptography, with, *e.g.*, $k_1 = sk$ and $k_2 = pk$. We employ hash functions to implement digital signatures, by applying the enc instruction with private key sk to the hash $h(tail)$. Signature verification is done by applying instruction dec with public key pk to the encrypted hash $h(tail)_{sk}$. However, in the rest of the paper, we will not consider digital signatures.

It is worth noticing that the set of fraglets programming instructions, given in Tables 1 and 2, consists of *rewrite rules* [16,18], with a simple *rewriting semantics* in which the left-hand side pattern (to the left of $\rightarrow$) is replaced by corresponding instances of the right-hand side. They represent *local transition rules* in a possibly distributed, concurrent system. Thus, we assume the presence of a *rewrite system* (defined by a single step transition operator $\rightarrow$, with $\rightarrow^*$ as its transitive and reflexive closure) operating on fraglets by means of the rewrite rules corresponding to the fraglets programming instructions. If we let f, f' range over fraglets, by applying operations from the rewrite system to a set F of fraglets, a new set $\mathsf{D}(\mathsf{F}) = \{ f \mid \mathsf{F} \rightarrow^* f\}$ of fraglets can be obtained. As an example of a simple step transition rule application, we have: $\mathsf{D}(\{[dup\ t\ a\ tail]\}) = \{ [dup\ t\ a\ tail], [t\ a\ a\ tail]\}$ since $[dup\ t\ a\ tail] \rightarrow_{dup} [t\ a\ a\ tail]\}$.

Below, we show the initial pool of fraglets, originally at stores $\mathcal{S}_A$ and $\mathcal{S}_B$, needed to execute a simple program that symmetrically encrypts a fraglet at store A, transfers the cyphertext at store B, and decrypts it at store B.

pool of fraglets originally at $\mathcal{S}_A$:

${}_A[\text{KEY K}]$	${}_A[\text{MSG M}]$
${}_A[\text{MATCH KEY MATCH MSG ENC NEWTAG}]$	${}_A[\text{MATCH NEWTAG SEND B KMSG}]$

pool of fraglets originally at $\mathcal{S}_B$:

${}_B[\text{KEY K}]$	${}_B[\text{MATCH KEY MATCH KMSG DEC NEWTAG}]$

One possible execution of the program is as follows.

${}_A[\text{KEY K}]\ {}_A[\text{MATCH KEY MATCH MSG ENC NEWTAG K}]$	$\rightarrow_{match}$	${}_A[\text{MATCH MSG ENC NEWTAG}]$
${}_A[\text{MATCH MSG ENC NEWTAG}]\ {}_A[\text{MSG M}]$	$\rightarrow_{match}$	${}_A[\text{ENC NEWTAG M}]$
${}_A[\text{ENC NEWTAG M}]$	$\rightarrow_{enc}$	${}_A[\text{NEWTAG } m_k]$
${}_A[\text{MATCH NEWTAG SEND B KMSG}]\ {}_A[\text{NEWTAG } m_k]$	$\rightarrow_{match}$	${}_A[\text{SEND B KMSG } m_k]$
${}_A[\text{SEND B KMSG } m_k]$	$\rightarrow_{send}$	${}_B[\text{KMSG } m_k]$
${}_B[\text{KEY K}]\ {}_B[\text{MATCH KEY MATCH KMSG DEC NEWTAG}]$	$\rightarrow_{match}$	${}_B[\text{MATCH KMSG DEC NEWTAG K}]$
${}_B[\text{MATCH KMSG DEC NEWTAG K}]\ {}_B[\text{KMSG } m_k]$	$\rightarrow_{match}$	${}_B[\text{DEC NEWTAG K } m_k]$
${}_B[\text{DEC NEWTAG K } m_k]$	$\rightarrow_{dec}$	${}_B[\text{NEWTAG M}]$

Tags *key*, *msg*, and *kmsg* are auxiliary. It is understood that $\mathcal{S}_A$ and $\mathcal{S}_B$ are the only stores at stake, and that, originally, there are no other fraglets than the ones in the initial pool.

3 A Threat Model for Fraglets

In this section we present a threat model for fraglets. We identify nodes of a communication network *A B C etc.*. with their fraglets stores, *viz.* $\mathcal{S}_A$, $\mathcal{S}_B$, $\mathcal{S}_C$, *etc.*. Thus, principals of a communication protocol are fraglets stores, within which fraglets (protocol code + protocol messages) are processed. In particular, communications are via the $send$ instruction.

We consider a protocol specification involving two honest roles, *viz.* an *initiator* $\mathcal{S}_S$ and a *responder* $\mathcal{S}_R$. Rather than a direct communication between them, we assume all their communication to flow through an *untrusted* store $\mathcal{S}_X$ which can either listen to or modify (fake) the fraglets exchanged between $\mathcal{S}_S$ and $\mathcal{S}_R$. Indeed, when modeling and verifying security properties of communication protocols, it is quite common to include an additional intruder (*à la* Dolev-Yao [7]) that is supposed to be malicious and whose aim is to subvert the protocol's correct behaviour. A protocol specification is then considered secure w.r.t. a security property if it satisfies this property despite the presence of the intruder. We thus propose a framework of three types of fraglets stores:

1. $\mathcal{S}_S$ plays the role of the protocol's initiator;
2. $\mathcal{S}_R$ plays the role of the protocol's responder;
3. $\mathcal{S}_X$ plays the role of the active and malicious intruder.

We let the initiator and the responder to be forced to communicate with the untrusted store through disjoint sets of communication (*send*) actions Σ^S_{com} and Σ^R_{com}, resp., such that a direct communication between them is impossible.

It is also assumed that, at deployment, each store $\mathcal{S}_I$, with $I = \{S, R, X\}$, contains the pool of keys Λ^I needed for the store to perform encryptions and decryptions.

In Fig. 1 we have sketched the communication scenario described above and we have instantiated it with $\Sigma^S_{com} = \{s_S[send\ \mathcal{S}_X\ ktail\ tail_K],$ $s_X[send\ \mathcal{S}_S\ ktail\ tail_K]\}$ and $\Sigma^R_{com} = \{s_X[send\ \mathcal{S}_R\ ktail\ tail_K],$ $s_R[send\ \mathcal{S}_X\ ktail\ tail_K]\}$, for some generic auxiliary tag $ktail$ and some generic encrypted sequence of symbols $tail_K$. We do not explicitly specify the type of the key used for encryption (and decryption), but we let $K \in \Lambda^I$. For instance, $\Lambda^S = \{k_{SR}, k_{SX}, sk_S, pk_R, pk_X\}$, *i.e.*, the shared secret key between $\mathcal{S}_S$ and $\mathcal{S}_R$, the shared secret key between $\mathcal{S}_S$ and $\mathcal{S}_X$, the secret key of $\mathcal{S}_S$, and the public keys of $\mathcal{S}_X$ and $\mathcal{S}_R$.

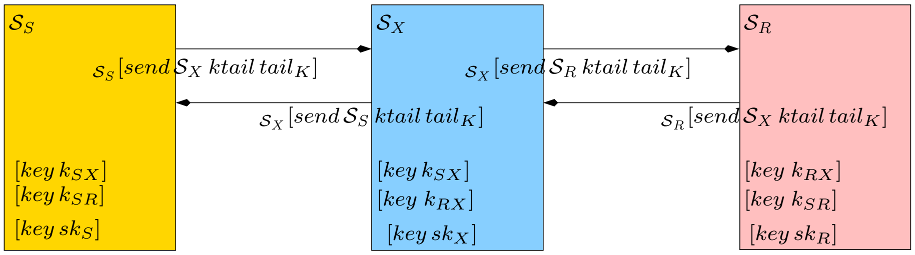


Fig. 1. The threat model for fraglets and fraglets stores

We do not fix *a priori* any specific behaviour for the intruder. $\mathcal{S}_X$ can process all the fraglets that it contains, by means of all the usual fraglets instructions. $\mathcal{S}_X$ can also honestly engage in a security protocol. Thus, it is decorated with its own pair of public/private keys (pk_X, sk_X), the symmetric key k_{SX}, shared with $\mathcal{S}_S$, and the symmetric key k_{RX}, shared with $\mathcal{S}_R$.

We also assume that i) private keys are initially contained only by the legitimate stores; ii) shared secret keys are initially contained only by the legitimate stores that share those keys. Finally, we assume that all the public keys are contained by all the stores at stake (public keys are not shown in the figure).

The intruder's knowledge. Here, we beat about the bush of the classical notion of the intruder's knowledge [8,20], *e.g.*, the set of all the messages the intruder knows from the beginning (its initial knowledge) united with the messages it can derive from the ones intercepted during a run of the protocol. Within a fraglet framework, this standard concept is slightly modified.

First, we say that a symbol is public the symbol is the second leftmost symbol of a fraglet at $\mathcal{S}_X$. Intuitively, the intruder's knowledge, at a given state of the computation, is the set of all the symbols that $\mathcal{S}_X$ knows. Let $\mathsf{F}_{\mathcal{S}_X}$ be the set of fraglets contained by $\mathcal{S}_X$.

Definition 1. *The intruder's knowledge* $\Phi_{\mathcal{S}_X}^{\mathsf{F}_{\mathcal{S}_X}}$ *is defined as:*
$$\Phi_{\mathcal{S}_X}^{\mathsf{F}_{\mathcal{S}_X}} = \{s_i | f_i =_{\mathcal{S}_X} [t_i \; s_i \; tail_i] \in \mathsf{D}(\mathsf{F}_{\mathcal{S}_X})\}$$

for some generic auxiliary or instruction tag $t_i, i = 0, \ldots, m$, and some generic sequence of symbols $tail_i, i = 0, \ldots, m$.

Security properties: Secrecy. We give here the definition of one of the most common security properties, secrecy, within a fraglets framework.

Intuitively, a message is secret when it is only known by the parties that should share that secret. Thus, in a fraglet context, a symbol is a secret between $\mathcal{S}_S$ and $\mathcal{S}_R$ when it is not possible for $\mathcal{S}_X$ to know that symbol.

We let $\mathsf{F}^0_{\mathcal{S}_S}$ and $\mathsf{F}^0_{\mathcal{S}_R}$ to be the initial, and fixed (according to the protocol in which the honest roles are engaged), set of fraglets stored at, resp., $\mathcal{S}_S$ and $\mathcal{S}_R$, at the beginning of the computation.

Analogously, $\mathsf{F}^0_{\mathcal{S}_X}$ is the set of fraglets initially contained by $\mathcal{S}_X$. *A priori*, we do not make any assumption on this set, apart from the fact that it does not contain private information of the honest roles, such as private keys of $\mathcal{S}_S$ and $\mathcal{S}_R$, and their shared secret key.

Thus, the following definition dictates when the secrecy property is preserved.

Definition 2. *The secrecy property* $Sec(s)_{\mathcal{S}_X}$ *of a symbol* s *is preserved if* $\forall \mathsf{F}^0_{\mathcal{S}_X}$ *and* $\forall (\mathsf{F}'_{\mathcal{S}_S} \cup \mathsf{F}'_{\mathcal{S}_X} \cup \mathsf{F}'_{\mathcal{S}_R}) \in \mathsf{D}(\mathsf{F}^0_{\mathcal{S}_S} \cup \mathsf{F}^0_{\mathcal{S}_X} \cup \mathsf{F}^0_{\mathcal{S}_R})$ *then* $s \notin \Phi_{\mathcal{S}_X}^{\mathsf{F}'_{\mathcal{S}_X}}$.

This means that, for every possible set of fraglets initially contained by the adversary's store, and for every possible union of fraglets' sets contained at $\mathcal{S}_S$, $\mathcal{S}_X$, and $\mathcal{S}_R$ that are derivable from the initial sets by applying every possible rule of the rewrite system, $\mathcal{S}_X$ will never know the secret symbol. This notion of secrecy is violated in the fraglets-based instantiation of the flawed NSPK, as shown in Section 5.

4 Specification and Execution of Fraglets in Maude

Maude is “a programming language that models (distributed) systems and the actions within those systems” [17]. The system is specified by defining algebraic data types axiomatizing system's states, and rewrite rules axiomatizing system's local transitions.

In this section we present our Maude executable specification for (crypto)fraglets. In particular, we define an algebra for them, *i.e.*, the *sorts* (types for values), and the *equationally specifiable operators* acting on those sorts (and constants). Also, we define the *rewrite laws* for describing the transitions that occur within and between the set of operators. Actually, the set of the rewrite laws represent the set of (crypto)fraglets instructions given in Tables 1, 2.

The Maude modules consisting of the core fraglets specification are basically three: FRAGLETS, FRAGLETS-RULES, and CRYPTO-FRAGLETS-RULES.

The functional module FRAGLETS provides declarations of sorts, *e.g.*, fraglets, symbols, stores, and public and private keys, and operators on those sorts, *e.g.*, concatenation of fraglets, and concatenations of fraglet stores. It also defines subsort relationships. For instance, symbols, stores, and public and private keys are understood as specialized fraglets. The module also provides reserved ground terms representing the names of the instructions (match, dup, exch, ...), and operators to encrypt fraglets, by means of either symmetric (crypt) or asymmetric (asymcrypt) encryption.[1]

```
fmod FRAGLETS is

  sort Fraglet .
  sort Key PKey . ---PKey is the sort for asymmetric cryptography
  sort Symb Store .
  sort FragletSet FragletSet@Store FragletStoreSet .

  subsort Symb Store Key PKey < Fraglet < FragletSet .
  subsort FragletSet@Store < FragletStoreSet .

  op nil : -> Fraglet .
  op _ _ : Fraglet Fraglet -> Fraglet [ctor assoc id: nil] .

  op empty : -> FragletSet .
  op _ , _ : FragletSet FragletSet -> FragletSet [assoc comm id: empty] .

  op _@_ : FragletSet Store -> FragletSet@Store .

  --- store concatenation
  op _ ; _ : FragletStoreSet FragletStoreSet -> FragletStoreSet [assoc comm ] .
  op [ _ ] : Fraglet -> FragletSet .

  op match : -> Symb .
  op dup : -> Symb .
  op exch : -> Symb  .
  ...
  ...
  op split : -> Symb .
  op send : -> Symb .

  op enc : -> Symb .
  op dec : -> Symb  .

  op fst : -> Symb .
  op snd : -> Symb .

  op crypt : Fraglet Key -> Fraglet .      --- symmetric encryption
  op asymcrypt : Fraglet PKey -> Fraglet . --- asymmetric encryption
  ...

endfm
```

[1] Appropriate equations for all the operators are defined in the complete specification.

The system module FRAGLETS-RULES defines the rewrite rules encoding the instructions given in Table 1. Below, we highlight (part of) the rules for dup and for a modified send instruction following the threat model defined in Section 3.

```
mod FRAGLETS-RULES is
  protecting FRAGLETS .

  vars T S : Symb .
  var TAIL : Fraglet .
  vars A B X : Store .
  vars  FS1 FS2 : FragletSet .

  rl [DUP] : [dup T S TAIL] => [T S S TAIL] .
  ...
  ...
  rl [SEND] : (([send X TAIL], FS1) @ A ) => (FS1 @ A) ; [TAIL] @ X .
  rl [SEND] : (([send X TAIL], FS1) @ A ) ;  (FS2 @ X ) => (FS1 @ A) ;
              (([TAIL], FS2) @ X) .
  ...
endm
```

CRYPTO-FRAGLETS-RULES defines the rewrite rules for cryptography and declares the pairs of public/private keys used for protocol specifications. Decryption instructions are defined as conditional rules (crl [DEC]): with symmetric keys, decryption is possible only if the key used for encryption is equal to the key that one intends to use for decryption; analogously, with asymmetric keys, decryption is possible only if the key used for encryption and the key intended to use for decryption form a pair. To this aim, auxiliary operators isKey and keypair have been equationally defined.

```
mod CRYPTO-FRAGLETS-RULES is
  protecting FRAGLETS .
  protecting FRAGLETS-RULES .

  op _ _ isKey : Key Key -> Bool .            --- aux op for dec rule
  op _ _ keypair : PKey PKey -> Bool [comm] . --- aux op for asym dec rule

  --- public and private keys declarations:
  ops pka pkb pkx : -> PKey .  --- public keys
  ops ska skb skx : -> PKey .  --- private keys

  vars   --- some variables declarations

  eq K K isKey = true .
  eq K1 K isKey = false [owise] .

  ---  which keys form a pair:
  eq pka ska keypair = true .
  eq pkb skb keypair = true .
  eq pkx skx keypair = true .
  eq K K1 keypair = false [owise] .

  --- SYMMETRIC
  rl [ENC] : [enc NewTag K TAIL] => [ NewTag crypt(TAIL,K) ] .
  rl [ENC] : [enc] => empty .
  crl [DEC] : [dec NewTag K crypt(TAIL,K1)] => [NewTag TAIL]
              if (K K1 isKey == true) .

  --- ASYMMETRIC
  rl [ENC] : [enc NewTag PK TAIL] => [ NewTag asymcrypt(TAIL,PK) ] .
  rl [ENC] : [enc NewTag SK TAIL] => [ NewTag asymcrypt(TAIL,SK) ] .
  crl [DEC] : [dec NewTag PK asymcrypt(TAIL,SK)] => [NewTag TAIL]
```

```
            if (PK SK keypair == true) .
  crl [DEC] : [dec NewTag SK asymcrypt(TAIL,PK)] => [NewTag TAIL]
            if (PK SK keypair == true) .
  ...
endm
```

To actually do something with those modules, one should use some strategies for applying the rules. A default strategy provided by Maude is implemented by the *rewrite* command, that explores one possible sequence of rewrites, starting by the set of rules and an initial state [16]. For example, plugging in "rew [enc newtag k tail] ." into the Maude environment, we obtain as a result "[newtag crypt(tail, k)]". The *search* command is also very convenient. *A priori*, it gives all the possible sequence of rewrites between an initial and a final state supplied by the user. Practically, since for certain systems the search could not terminate, the command is decorated with an optional bound on the number of desired solutions and on the maximum depth of the search.

5 A Case Study: Fraglets-Based NSPK

In this section, we first recall the Needham-Schroeder Public Key protocol (NSPK) [19], a paradigmatic security protocol, widely examined by protocol researchers. Considering two interleaved runs of the protocol, [12] found an attack leading to both an authentication and a secrecy failure, and supplied an amended version of the protocol.

Then, we show the Maude specification of the fraglets-based NSPK. Executing the specification, we find the secrecy attack. This starting example illustrates the usefulness of executing fraglets specifications in the Maude engine for validation purposes.

The NSPK protocol. NSPK tries to establish an authenticated communication between a pair of agents, *A* and *B*. The protocol is based on public-key cryptography: each agent possesses a pair of private/public keys. While the public key can be accessed by all agents of the distributed system, the private key should remain a secret of its owner. Also, the protocol makes use of the nonces *nA*, and *nB*. Nonces are freshly generated, random numbers, generally exploited in cryptographic protocols to assure freshness of messages. Actually, one of the intents of NSPK is also to exchange secret values *nA*, and *nB* to be used for subsequent encrypted communication between *A* and *B*.

The original version of (part of) the protocol is hereafter presented. We denote the transmission of message *msg* from sender *S* to receiver *R* as $S \rightarrow R \ : \ msg$. Also, encryption of message *msg* with public key *pk* is denoted as $\{msg\}_{pk}$.

$$\begin{array}{ll} 1\ A \rightarrow B: & \{A, nA\}_{pkB} \\ 2\ B \rightarrow A: & \{nA, nB\}_{pkA} \\ 3\ A \rightarrow B: & \{nB\}_{pkB} \end{array}$$

In message 1, *A* sends his identity and his newly generated nonce *nA* to *B*, encrypted with the public key of *B*, pkB. *B* decrypts message 1 with the correspondent private key skB. Then, *B* creates her nonce *nB* and sends back to *A* the two nonces, encrypted with the public key of *A*, pkA (message 2). Finally, *A* decrypts message 2, retrieves the nonce *nB*, and sends it back, encrypted, to *B*.

Once terminated a run of this protocol, it would seem reasonable that:

1. *A* and *B* know with whom they have been interacting; indeed, *A* can be assured that message 2 came from *B*, because *B* is the only agent who can decrypt message 1, *i.e.*, the message sent by *A* and containing nA. Analogously, *B* can be assured of being talked to *A*, because *A* is the only agent who can decrypt message 2.
2. *A* and *B* agree on the values of nA and nB.
3. No one else knows the values of nA and nB (secrecy).

Maude specification and execution of the fraglets-based NSPK. For many years the NSPK protocol has been believed to satisfy those properties. In [12], Gavin Lowe discovered an attack, in which an adversary *X* acts as a honest principal with *A* in a first run of the protocol, while she masquerades as *A* for *B* in a second run of the protocol:

$$\begin{array}{lrl} a.1 & A \rightarrow X: & \{A, nA\}_{pkX} \\ b.1 & X(A) \rightarrow B: & \{A, nA\}_{pkB} \\ b.2 & B \rightarrow X(A): & \{nA, nB\}_{pkA} \\ a.2 & X \rightarrow A: & \{nA, nB\}_{pkA} \\ a.3 & A \rightarrow X: & \{nB\}_{pkX} \\ b.3 & X \rightarrow B: & \{nB\}_{pkB} \end{array}$$

where *X(A)* represents *X* generating (resp. receiving) the message, making it appear as generated (resp. received) by *A*. What happens is that *A* starts Session *a* with *X*, that, in its turn, starts Session *b* with *B*, pretending to be *A*. At the end of the two sessions, *B* thinks that i) she has been communicating with *A*, while this is not the case, and ii) she and *A* share exclusively *nA* and *nB*, while this is not the case.

Module NSPK-flawed-intruder declares the names of the stores, the nonces and the auxiliary tags necessary to make fraglets opportunely react with each other according to NSPK.

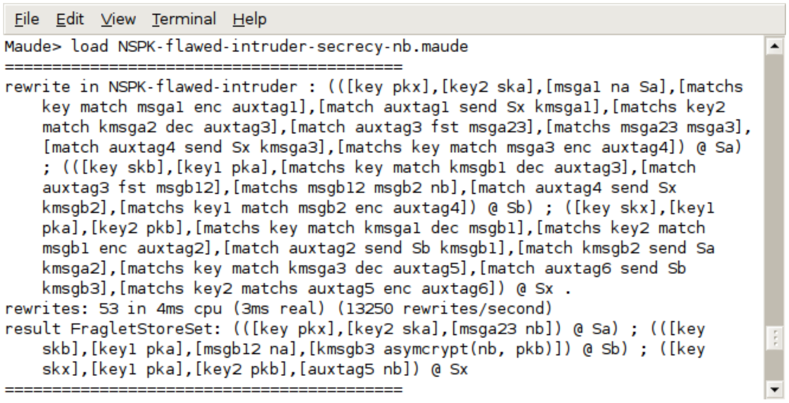
```
mod NSPK-flawed-intruder is
    protecting FRAGLETS .
    protecting FRAGLETS-RULES .
    protecting CRYPTO-FRAGLETS-RULES .

   --- two interleaved runs
   --- we consider three participants, A, B, and X.
   --- secrecy attack on Nb: at the end of the runs, B is convinced that
       Nb is a secret known only by B and A, while X knows Nb.

   ops Sa Sb Sx : -> Store .
   ops na nb : -> Key .

   --- all the auxiliary tags for fraglets transformations and reactions
   ops key key1 key2 key3  : -> Symb .
   ops auxtag auxtag1 auxtag2 auxtag3 auxtag4 auxtag5 auxtag6 : -> Symb .
   ops msga1 msga2 msga3 msga12 msga23 msgb1 msgb2 msgb3 msgb12 msgb23
       kmsga1 kmsga2 kmsga3  kmsgb1 kmsgb2 kmsgb3 secretb : -> Symb .
endm
```

We can explore one possible sequence of rewrites by running the *rewrite* command. The argument of *rewrite* is the initial configuration of the three stores $\mathcal{S}_A$, $\mathcal{S}_B$, and $\mathcal{S}_X$. Actually, this configuration represents the fraglets program to execute two interleaved sessions of NSPK. Figure 2 shows the screenshot of executing the fraglets-based NSPK.



```
Maude> load NSPK-flawed-intruder-secrecy-nb.maude
==========================================
rewrite in NSPK-flawed-intruder : (([key pkx],[key2 ska],[msga1 na Sa],[matchs
    key match msga1 enc auxtag1],[match auxtag1 send Sx kmsga1],[matchs key2
    match kmsga2 dec auxtag3],[match auxtag3 fst msga23],[matchs msga23 msga3],
    [match auxtag4 send Sx kmsga3],[matchs key match msga3 enc auxtag4]) @ Sa)
    ; (([key skb],[key1 pka],[matchs key match kmsgb1 dec auxtag3],[match
    auxtag3 fst msgb12],[matchs msgb12 msgb2 nb],[match auxtag4 send Sx
    kmsgb2],[matchs key1 match msgb2 enc auxtag4]) @ Sb) ; ([key skx],[key1
    pka],[key2 pkb],[matchs key match kmsga1 dec msgb1],[matchs key2 match
    msgb1 enc auxtag2],[match auxtag2 send Sb kmsgb1],[match kmsgb2 send Sa
    kmsga2],[matchs key match kmsga3 dec auxtag5],[match auxtag6 send Sb
    kmsgb3],[matchs key2 matchs auxtag5 enc auxtag6]) @ Sx .
rewrites: 53 in 4ms cpu (3ms real) (13250 rewrites/second)
result FragletStoreSet: (([key pkx],[key2 ska],[msga23 nb]) @ Sa) ; (([key
    skb],[key1 pka],[msgb12 na],[kmsgb3 asymcrypt(nb, pkb)]) @ Sb) ; ([key
    skx],[key1 pka],[key2 pkb],[auxtag5 nb]) @ Sx
==========================================
```

Fig. 2. Fraglets-based NSPK: execution in Maude

At the end of the computation, the result is of sort FragletStoreSet (*i.e.*, a set of fraglet stores). In particular, $\mathcal{S}_A$ contains the newly received nonce nb. $\mathcal{S}_B$ contains nonce na, received in message b1, and the last encrypted message received by $\mathcal{S}_A$ (with her nonce nb). Finally, the adversary's store contains fraglet $[auxtag5\ nb]$, leading to a secrecy attack with respect to nb.

We can also use *search*, looking for more than one possible sequence of rewrites from the initial configuration in the screenshot to a state where $[auxtag5\ nb]$ belongs to $\mathcal{S}_X$. The result gives one of such sequences obtained after 13422 rewrites in 880 milliseconds (cpu).

6 Conclusions

In this paper, we defined a threat model for a refined version of cryptofraglets. On that model, it is possible to define security properties for fraglets, starting from the notion of the adversary's knowledge. As an example, we dealt with the secrecy property. Then, we proposed an executable specification of cryptofraglets in Maude, together with an executable specification of a fraglets-based version of the well known security protocol NSPK. A security verification was achieved with respect to secrecy.

On the one hand, this illustrates the usefulness of executing fraglets specifications in the Maude engine for validation purposes. However, this represents only a partial verification on a reduced scenario. Indeed, we focused on a fixed initial set of fraglets in the adversary store. Furthermore, the sets of fraglets derivable from the initial sets by applying every possible rule of the rewrite system is fixed.

One of the main goals for the future is to extend this work to more comprehensive scenarios, in order to verify the secrecy property as defined in section 3, as well as other kinds of security properties, in a more exhaustive way. Indeed, it is possible to express strategies for executing Maude specifications in Maude itself (see, *e.g.*, [6] for strategies examples). As future work, we intend to use some ad-hoc defined strategies for extending our verification framework.

Acknowledgments. This work is partly supported by the FP6-027748 EU project BIONETS (*BIOlogically-inspired autonomic NETworks and Services*).

The authors would like to thank Roberto Bruni and Alberto Lluch Lafuente for their kind support on Maude matters, Grit Denker for providing pointers to useful Maude specifications, and Daniel Schreckling for valued discussions on the cryptofraglets
syntax.

References

1. Van Baalen, J., Böhne, T.: Automated protocol analysis in Maude. In: Hinchey, M.G., Rash, J.L., Truszkowski, W.F., Rouff, C.A., Gordon-Spears, D.F. (eds.) FAABS 2002. LNCS (LNAI), vol. 2699, pp. 68–78. Springer, Heidelberg (2003)
2. The BIONETS website, `http://www.bionets.eu/`
3. Cervesato, I., Durgin, N., Lincoln, P.D., Mitchell, J.C., Scedrov, A.: A meta-notation for protocol analysis. In: Proc. CSFW-12, pp. 55–69. IEEE, Los Alamitos (1999)
4. Clarke, E., Jha, S., Marrero, W.: Verifying security protocols with Brutus. ACM Transactions on Software Engineering and Methodology 9(4), 443–487 (2000)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Denker, G., Meseguer, J., Talcott, C.: Protocol specification and analysis in Maude. In: Formal Methods and Security Protocols (1998)
7. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Trans. Inf. Theory 29(2), 198–208 (1983)
8. Egidi, L., Petrocchi, M.: Modelling a secure agent with team automata. In: Proc VODCA 2004. ENTCS, pp. 119–134. Elsevier, Amsterdam (2005)
9. Focardi, R., Martinelli, F.: A uniform approach for the definition of security properties. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 794–813. Springer, Heidelberg (1999)
10. FRAGLETS website (last access: 18/06/2009), `http://www.fraglets.net`
11. Lenzini, G., Gnesi, S., Latella, D.: Spider: a Security Model Checker. In: Proc. FAST 2003, pp. 163–180 (2003); Also, Technical Report ITT-CNR-10, Informal proceedings (2003)
12. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using fdr. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
13. Lynch, N.: I/O automaton models and proofs for shared-key communication systems. In: Proc. CSFW 1999, pp. 14–31. IEEE, Los Alamitos (1999)
14. Martinelli, F., Petrocchi, M.: Access control mechanisms for fraglets. In: BIONETICS, ICST (2007)
15. Martinelli, F., Petrocchi, M.: Signed and weighted trust credentials for fraglets. In: BIONETICS, ICST (2008)
16. Maude System website (last access: 18/06/2009), `http://maude.cs.uiuc.edu/`
17. McCombs, T.: Maude 2.0 Primer (2003), `http://maude.cs.uiuc.edu/download/`
18. Meseguer, J.: Research directions in rewriting logic. In: Computational Logic. LNCS, vol. 165, Springer, Heidelberg (1997)
19. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. Communications of the ACM (21), 393–399 (1978)

20. Petrocchi, M.: Formal techniques for modeling and verifying secure procedures. PhD thesis, University of Pisa (May 2005)
21. Petrocchi, M.: Crypto-fraglets. In: BIONETICS. IEEE, Los Alamitos (2006)
22. Tschudin, C.: Fraglets - a metabolistic execution model for communication protocols. In: Proc. AINS 2003 (2003)
23. Tschudin, C., Yamamoto, L.: A metabolic approach to protocol resilience. In: Smirnov, M. (ed.) WAC 2004. LNCS, vol. 3457, pp. 191–206. Springer, Heidelberg (2005)
24. Yamamoto, L., Tschudin, C.: Experiments on the automatic evolution of protocols using genetic programming. In: Stavrakakis, I., Smirnov, M. (eds.) WAC 2005. LNCS, vol. 3854, pp. 13–28. Springer, Heidelberg (2006)
25. Yamamoto, L., Tschudin, C.: Genetic evolution of protocol implementations and configurations. In: Proc. SelfMan 2005 (2005)