Cryptofraglets Reloaded

Bioinspired Security Modeling of a RFID Protocol and Properties

Marinella Petrocchi
CNR Istitute of Informatics and Telematics
Via G. Moruzzi 1
56124 Pisa - ITALY
m.petrocchi@iit.cnr.it

Angelo Spognardi
CNR Istitute of Informatics and Telematics
Via G. Moruzzi 1
56124 Pisa - ITALY
a.spognardi@iit.cnr.it

Paolo Santi
MIT-Fraunhofer Ambient Mobility, Senseable City Lab
77 Massachusetts Avenue
Cambridge, MA 02139 USA
psanti@mit.edu

ABSTRACT
Fraglets represent an execution model for communication protocols that resembles the chemical reactions in living organisms. The strong connection between their way of transforming and reacting and formal rewriting systems makes a fraglet program amenable to automatic verification. Starting from past work where the model has been enriched and executed to specify security protocols and properties (leading to the definition of cryptofraglets), this paper updates cryptofraglets and gives examples of concrete sample analyses over a secure RFID protocol.

Keywords
Fraglets, cryptofraglets, secure RFID protocols, Maude

1. INTRODUCTION
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. They were originally introduced to automatise the protocol development process, from design, to implementation, and deployment. In the past, main fields of applications have been protocol resilience and genetic programming experiments, see, e.g. [24, 25, 26, 27].

The close connection between fraglets' basic mechanism of transforming/reacting and formal rewriting systems such as MultiSet Rewriting [2, 23] makes a fraglet program amenable to (automatic) verification. In particular, with an eye to modelling security protocols and verifying security properties, the original pool of fraglets' instructions have been incrementally extended in the past years to deal with symmetric cryptography [23], access control mechanisms [15], and dedicated primitives for trust management [16]. This led to the definition of *cryptofraglets*, i.e., fraglets enriched with capabilities of encrypting, decrypting, signing, and verifying signatures over a series of symbols. Successively, work in [17] showed a more concrete advancement towards adopting fraglets for security modelling and verification, by proposing an executable specification of cryptofraglets in Maude [4, 18], the popular engine based on rewriting logic [20].

The current paper extends the work in [17] by 1) enriching the cryptofraglets set of instructions with specific functionalities for hashing and message authentication coding: these functionalities are particularly significant, as they are used as basic building blocks in many protocols; 2) refining the fraglets threat model and the notion of the adversary knowledge (with respect to what proposed in [17]), i.e., the set of messages an adversary knows between the beginning and end of a protocol run; 3) proposing an extended executable specification of cryptofraglets in Maude. Effectiveness of the enhanced framework is demonstrated by presenting a Maude specification of a privacy preserving RFID identification protocol, RIPP-FS [5, 6], under the fraglets communication paradigm. The protocol provides a series of features, including: *secrecy* of the shared key used to authenticate the tag to the reader; *tag privacy*, intended as un-linkability of two, or more, answers coming from the same tag, against a passive adversary that eavesdrops two (or more) protocol sessions; and *forward secrecy*, i.e., impossibility for an active adversary that has captured a tag to know which previously eavesdropped answers have been produced by that tag. The specification of RIPP-FS in Maude allows for the automated analysis of such features. We show examples of such analyses, serving as a proof of concept to demonstrate feasibility of modelling and analysing security protocols specified via fraglets.

The paper is organized as follows. Section 2 recalls the fraglets model and introduces a refined version of cryptofraglets. Section 3 revises the threat model for cryptofraglets proposed in [17]. Section 4 introduces the executable specification of cryptofraglets in Maude. Section 5 shows a fraglets-based instantiation of one session of the RIPP-FS protocol, highlighting some Maude capabilities to perform automatic analyses on the protocol execution. In Section 6, we briefly revise related work in the area of fraglets and rewrite systems. Finally, Section 7 concludes the paper.

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 $send$ ($mcast$) instruction transfers a fraglet from a source store to a destination store (to a set of destination stores).

Fraglets are processed through a simple prefix programming language. *Transformation* instructions involve a single fraglet, while *reaction* instructions involve two fraglets. Table 1 shows the fraglets core instructions. The interested reader can find a comprehensive tutorial in [12].

Table 1: Fraglets core instructions

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

Two fraglets react by instruction $match$, and their tails are concatenated. With the catalytic $matchp$, the reaction rule persists.

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 $*$ ($split$).

Table 2: Fraglets communication instructions

send	$_{S_A}$[send B tail] $\rightarrow$ $_{S_B}$[tail]
mcast	$_{S_A}$[mcast (B,Slist) tail] $\rightarrow$ $_{S_A}$[mcast Slist tail], $_{S_B}$[tail]

Table 2 reports two particular transformation instructions used for enabling communication. In particular, *send* performs a communication between two fraglets stores. It transfers a fraglet from store $\mathcal{S}_A$ to store $\mathcal{S}_B$. Notation

$$_{S_A}[s1\ s2\ \ldots\ tail]$$

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 this to make the name of the store explicit.

The *mcast* instruction is newly introduced in this paper to model a multicast communication, namely a communication from a store to a group of other stores, listed in symbol *Slist*, which represents a list of stores. In case *Slist* is composed of all possible receivers, *mcast* acts a broadcast. This instruction is recursively defined, as a fraglet that transforms itself in a simpler one, while generating a new fraglet in one of the destination stores.

The cryptographic version of fraglets, namely the cryptofraglets [23, 15], extend the fraglets programming language with instructions for encryption and decryption. In this paper, we introduce new instructions for hashing and message authentication coding, and their verification. Table 3 shows the set of cryptographic instructions. Note that, since fraglets processing is through matching tags, the presence of either a reserved instruction tag or of an auxiliary tag as the leftmost symbol is necessary for the computation to proceed.

The encryption instruction takes as input the [*enc newtag* k_1 *tail*] fraglet, consisting of the reserved instruction tag *enc*, an auxiliary

Table 3: Crypto-instructions for encryption, decryption, hashing, and message authentication coding

enc	[enc t k tail] $\rightarrow$ [t $tail_k$]
dec	[dec t k $tail_k$] $\rightarrow$ [t tail]
hash	[hash t tail] $\rightarrow$ [t h(tail)]
hashi	[hashi t i tail] $\rightarrow$ [hashi t i-1 h(tail)]
hmac	[hmac t k tail] $\rightarrow$ [t h(k $\|$ tail)]
hv	[hv t tail1 tail2] [t tail1] $\rightarrow$ [tail2]
hnv	[hnv t tail1 tail2] [t tail3] $\rightarrow$ [tail2]

tag *t*, the encryption key k, and a generic sequence of symbols *tail*, representing the meaningful payload to be encrypted. It returns [*t tail*], with the auxiliary tag and the cyphertext *tail*$_k$. The decryption instruction *dec* acts in the complementary way.

Instruction *hash* applies an hash function to the generic sequence of symbols *tail*. Instruction *hashi* performs hash iteration, i.e., the application of the hash function h i times on *tail*: $h^i(tail) = h(h(h(\ldots h(tail))\ldots)$. It is a transformation rule operating on fraglet [*hashi t i tail*]. Hash iteration is useful to model the Lamport's scheme, namely one-time passwords based on sequences of values iteratively obtained computing a hash function on a shared secret [19]. The Lamport's scheme is widely used for authentication purposes and to guarantee the property of "forward secrecy" (see Section 5 for an example of application). The *hashi* fraglet for hash iteration is able to transform itself and evolve to a single *hash* fraglet, eventually resulting in a fraglet with a tag and a tail sequence as input to a hash function h, when i becomes 0.

The fraglet [*hmac t k tail*] evolves to compute the hashed-MAC (Message Authentication Code), commonly realized with the combination of a shared key k and a message (the *tail* sequence). The fraglet transforms itself to a hashed fraglet, with the concatenation of the tail sequence plus the key as input to the hash function h. Operator $\|$ denotes concatenation of symbols.

It is worth noting that cryptofraglets instructions abstract from the cryptographic details concerning the operations by which they can be encrypted and decrypted. We make the so called *perfect cryptography assumption* and we consider encryption as a black box: an encrypted (sequence of) symbol(s) cannot be correctly learnt unless with the right decryption key. Similarly, we consider hash functions to be collision-resistant and non-invertible. This approach is standard in (most of) the analysis of cryptographic communication protocols, see, *e.g.*, [3, 11, 13, 14].

Finally, for modelling purposes, we define the following simple rules for hash verification, see Table 4. Basically, instruction *hv* let a computation proceed with *tail2* if two symbols sequences *tail1* in two different fraglets with matching tags are equal. In a complementary way, instruction *hnv* let a computation proceed if two symbols sequences *tail1* and *tail3* in two different fraglets with matching tags are disequal. The role of these two control instructions will be clarified in Section 5.

The set of fraglets programming instructions in Tables 1, 2, 3, 4 consists of *rewrite rules* [18, 20], 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 one. They represent

Table 4: Hash verification instructions

hv	[hv t tail1 tail2] [t tail1] $\rightarrow$ [tail2]
hnv	[hnv t tail1 tail2] [t tail3] $\rightarrow$ [tail2]

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 fraglets' set F, a new fraglets' set $\mathsf{D(F)} = \{ f \mid \mathsf{F} \rightarrow^* f\}$ is 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 encrypts a fraglet at store A, transfers the cyphertext at store B, and decrypts the cyphertext at store B.

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

$_A$[*key k*]	$_A$[*msg m*]
$_A$[*match key match msg enc t*]	$_A$[*match t send B kmsg*]

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

$_B$[*key k*]	$_B$[*match key match kmsg dec t*]

One possible execution of the program is as follows.

$_A$[*key k*] $_A$[*match key match msg enc t*]	$\rightarrow_{match}$ $_A$[*match msg enc t k*]
$_A$[*match msg enc t k*] $_A$[*msg m*]	$\rightarrow_{match}$ $_A$[*enc t k m*]
$_A$[*enc t k m*]	$\rightarrow_{enc}$ $_A$[*t* m_k]
$_A$[*match t send B kmsg*] $_A$[*t* m_k]	$\rightarrow_{match}$ $_A$[*send B kmsg* m_k]
$_A$[*send B kmsg* m_k]	$\rightarrow_{send}$ $_B$[*kmsg* m_k]
$_B$[*key k*] $_B$[*match key match kmsg dec t*]	$\rightarrow_{match}$ $_B$[*match kmsg dec t k*]
$_B$[*match kmsg dec t k*] $_B$[*kmsg* m_k]	$\rightarrow_{match}$ $_B$[*dec t k* m_k]
$_B$[*dec t k* m_k]	$\rightarrow_{dec}$ $_B$[*t m*]

Tags *key*, *msg*, and *kmsg* are auxiliary. In the above example, we assume 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 at *A* and *B*.

3. THREAT MODEL

This section introduces a new threat model for fraglets. We identify nodes $A\ B\ C, \ldots$ of a communication network as fraglets stores, *viz.* $\mathcal{S}_A, \mathcal{S}_B, \mathcal{S}_C, \ldots$. Thus, principals of a communication protocol are fraglets stores, within which fraglets (protocol code + protocol messages) are being processed. In particular, communications are via the $send$ ("one to one" communication) and $mcast$ ("one to many" communication) instructions.

We consider a protocol specification involving two, or more, honest roles. In case of two roles, we can call them *viz.* the *initiator* $\mathcal{S}_S$ and the *responder* $\mathcal{S}_R$. Moreover, when modeling and verifying security properties of communication protocols, it is quite common to include an additional intruder 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 model the intruder as an *untrusted* store $\mathcal{S}_X$, which can eavesdrop (and possibly modify) the fraglets exchanged between $\mathcal{S}_S$ and $\mathcal{S}_R$ (or, more generally, among a set of honest stores).

We do not *a priori* fix any specific behaviour for the adversary. $\mathcal{S}_X$ can process fraglets by means of all the instructions presented in Section 2. $\mathcal{S}_X$ can also honestly engage in a security protocol. To this aim, the pool of fraglets at $\mathcal{S}_X$ can contain also symmetric keys k_{SX} and k_{RX}, shared with, e.g., $\mathcal{S}_S$ and $\mathcal{S}_R$, respectively. Concerning cryptographic keys, we assume that, at deployment, each store $\mathcal{S}_I$ contains the pool of keys Λ^I needed for the store to perform encryptions and decryptions. We also assume that shared secret keys are initially contained only by the legitimate stores that share those keys.

Figure 1 shows how a multicast communication is activated among a subset of stores. Each store, within a universe of available stores, process fraglets representing both code and messages to run communication protocols sessions. Instruction $mcast$ at store $\mathcal{S}_A$ can be programmed to list the subset of stores that will actually receive messages from $\mathcal{S}_A$. In particular, solid arrows represent actual communication (i.e., $\mathcal{S}_B$ and $\mathcal{S}_X$ receive messages from $\mathcal{S}_A$), while dashed arrows represent potential communication (i.e., in principle, communication with $\mathcal{S}_A$ and $\mathcal{S}_C$ is possible, but not activated in the figure example). Note that, when specifying security protocols, the adversary store $\mathcal{S}_X$ is always included in the list of stores of every $mcast$ instruction, to model, at least, eavesdropping.

Apart from some fraglets resident at the stores (denoted, resp., as [restA], [someB], [someC], and [someX]), in the figure we highlight fraglet [mcast B;X tail], which enacts communication towards $\mathcal{S}_B$ and $\mathcal{S}_X$.

Adversary's knowledge. The adversary's knowledge [9, 22] is the set of all the messages an adversary knows from the beginning (its initial knowledge) united with the messages it can derive from the ones intercepted during a run of the protocol. In terms of fraglets, the adversary knowledge is the set of all fraglets hosted at $\mathcal{S}_X$, at a given state of the computation.

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:*

$$\Phi_{\mathcal{S}_X}^{\mathsf{F}_{\mathcal{S}_X}} = \{tail_i | f_i =_{\mathcal{S}_X} [t_i\ tail_i] \in \mathsf{D}(\mathsf{F}_{\mathcal{S}_X})\}$$

for some generic auxiliary or instruction tag $t_i, i = 0, \ldots, m$.

Security properties: Secrecy. Secrecy is one of the most common security properties. 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 (or sequence of symbols) is a secret between $\mathcal{S}_S$ and $\mathcal{S}_R$ when it is not possible for $\mathcal{S}_X$ to know that symbol (sequence).

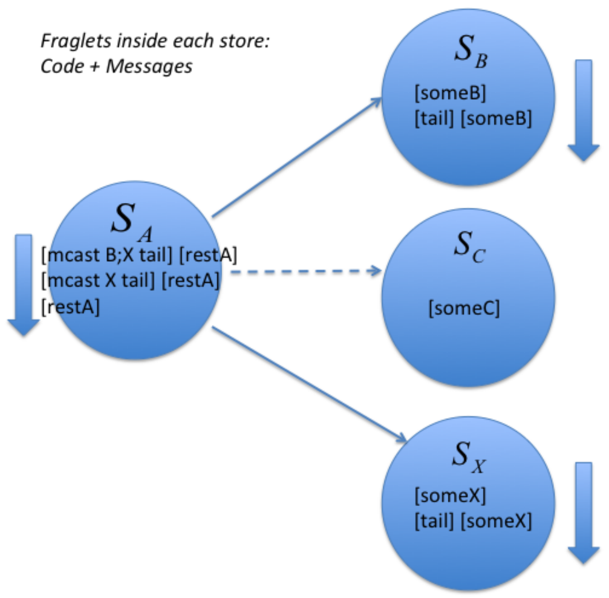


Figure 1: An example communication scenario for fraglets and fraglets stores.

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, e.g., shared secret key between $\mathcal{S}_S$ and $\mathcal{S}_R$.

DEFINITION 2. *The secrecy property $Sec(tail)_{\mathcal{S}_X}$ of a sequence of symbols tail 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* $tail \notin \Phi^{\mathsf{F}^{'}_{\mathcal{S}_X}}_{\mathcal{S}_X}$.

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 sequence.

4. EXECUTABLE FRAGLETS IN MAUDE

Maude is "a programming language that models (distributed) systems and the actions within those systems" [18]. 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 rewrite laws represents the set of (crypto)fraglets instructions given in the tables of Section 2.

The Maude modules consisting of the core cryptofraglets 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 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 keys are seen as specialized fraglets, meaning that all variables of sorts symbols, stores, and keys are fraglets too. The module also provides reserved ground terms representing the names of the instructions (match, dup, exch, . . .). Below we show an excerpt of module FRAGLETS. The complete Maude specification of fraglets and cryptofraglets, together with appropriate equations for all the declared operators, is available at `http://mib.projects.iit.cnr.it/bict14/cryptofraglets.html`.

```
fmod FRAGLETS is

  sort Symb Fraglet Instr .
  sort Key .  --- symmetric key
  sort PKey . --- asymmetric key
  sorts Store Stolist .
  sort FragletSet .
  sort FragletSet@Store .
  sort FragletStoreSet .

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

  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 .

  --- list of stores with related fraglets sets inside
  op _ ; _ : FragletStoreSet FragletStoreSet ->
                FragletStoreSet [assoc comm ] .

  --- a fraglet set from a fraglet:
  op [ _ ] : Fraglet -> FragletSet .

  ---a list of stores:
  op nstore : -> Stolist .
  op _\%_ : Stolist Stolist ->
                  Stolist [assoc comm id: nstore] .

  ---instructions keywords:
  op match : -> Instr  .
  op dup : -> Instr  .
  op exch : -> Instr  .
  op fork : -> Instr .
  op matchp : -> Instr .
  op matchs : -> Instr .
  op nop : -> Instr .
  op nul : -> Instr .
  op pop2 : -> Instr .
  op split : -> Instr .
  op send : -> Instr .
  op mcast : -> Instr .

endfm
```

Module FRAGLETS-RULES defines the rewrite rules encoding the instructions given in Tables 1 and 2. Below, we highlight excerpts of the rules for the communication instructions of Table 2.

```
mod FRAGLETS-RULES is
  protecting FRAGLETS .

  var TAIL : Fraglet .
  vars C D : Store .
  var DLIST : Stolist .
  ...

 rl [SEND] : [send D TAIL] @ C  => [TAIL] @ D .
 rl [MCAST] : [mcast (D \% DLIST) TAIL] @ C  =>
                          [mcast DLIST TAIL] @ C; [TAIL] @ D .
 rl [MCAST] : [mcast D TAIL] @ C  => [TAIL] @ D .
 ...

endm
```

Module CRYPTO-FRAGLETS-RULES defines the rewrite rules for encryption/decryption/hashing/message authentication code. Decryption and hash verification are defined as conditional rules (crl [DEC], crl [HNV]): decryption is possible only if the key used for encryption is equal to the key that one intends to use to decrypt. A hash value is verified only if it equals to some other hash that a fraglet store is able to compute. Below we show an excerpt of the module seconding cryptofraglets instructions.

```
mod CRYPTO-FRAGLETS-RULES is
  protecting FRAGLETS .
  protecting FRAGLETS-RULES .
  ...
  op crypt : Fraglet Key -> Fraglet .
  --- aux op for dec rule:
  op _ _ isKey : Key Key -> Bool .
  eq K K isKey = true .
  eq K1 K isKey = false [owise] .
  ...

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

   ---Hashing
   rl [HASH] : [hash t tail] => [t h(tail)] .
   ---Iterated (s(i) is successor of i)
   rl [HASHI] : [hashi t s(i) tail] => [hashi t i h(tail)] .
   ---Hmac
   rl [HMAC] : [hmac t tail1 tail2] => [t h(tail1 || tail2)] .
   ---Hash is not verified
   crl [HNV] : [ hnv t h(tail1) tail ], [t h(tail2) ] =>
                        [tail] if h(tail1) =/= h(tail2) .
  ...
endm
```

To actually do something with those modules, Maude uses appropriate strategies for rule application. A Maude default strategy is implemented by the *rewrite* command, that explores one possible sequence of rewrites, starting by a set of rules and an initial state [18]. For example, plugging in "rew [enc t k tail] ." into the Maude environment, we obtain as a result "[t 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.

In next sections, we show the fraglets specification of a RFID protocol guaranteeing a set of security properties. We will describe properties analysis example in Section 5, through the use of basic strategies. All the analysis examples shown in the paper illustrate how the implementation of fraglets in Maude allows us to exploit the Maude's analysis toolset. In this respect, it is worth noting that in the above analyses we have made use of only basic Maude capabilities. There are several other Maude tools whose use remains to be investigated (e.g., its SAT solver, its reachability analyser and its LTL model checker).

5. FRAGLET SPECIFICATION OF A RFID PROTOCOL

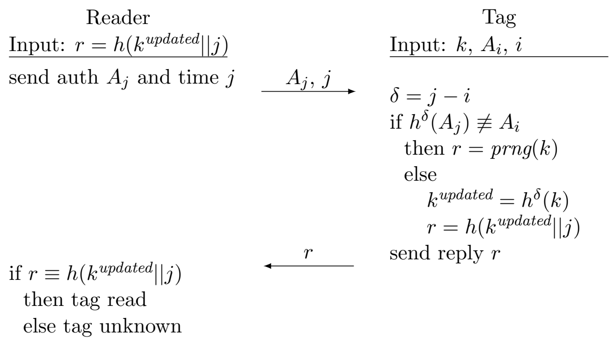
In this section, we provide a specification of a RFID protocol through cryptofraglets, together with the modelling of some of its provided security properties. We firstly introduce the RIPP-FS [5] protocol, that guarantees RFID *tag privacy*, *mutual authentication* and *forward secrecy* and, then, we provide the protocol fraglets formulation. A subsequent version of the protocol eRIPP-FS[6] was proposed to limit a timing attack to which some hardware implementation could be potentially exposed.

Before introducing the protocol, we briefly recall some security notions. With *tag privacy* we indicate the property for which a passive attacker cannot distinguish two answers of a same tag, provided that she cannot distinguish between a hmac value and a pseudo random generated number. *Mutual authentication* is the property through which two entities prove each other their own identity. *Forward secrecy* ensures that the knowledge of a piece of information does not disclose any information about the past. In the particular case of RIPP-FS, the knowledge of the key of a captured tag does not disclose any information about the previous answers of that tag.

5.1 The RIPP-FS protocol

RIPP-FS, introduced in [5, 6], is a privacy preserving identification protocol for RFID tags: it is able to perform the scanning of multiple tags with only one reading, avoiding the tracking of the tags among subsequent readings. Figure 2 concisely describes the computations and the message exchanges of the protocol.

The main building block of RIPP-FS is the use of a Lamport's scheme to provide the authentication of the reader: each tag stores a value A_i that uses to verify the authenticity of readers' query at time i, since A_i is the result of the iteration of a hash function h over a secret value A_0, namely $A_i = h^i(A_0)$, where $h^k(\cdot)$ means the function h iterated k times, $h(h(\ldots(\cdot))\ldots)$. In particular, to authenticate a reading, once receiving a new value A_j



Reader
Input: $r = h(k^{updated}||j)$
send auth A_j and time j
A_j, j
Tag
Input: k, A_i, i
$\delta = j - i$
if $h^{\delta}(A_j) \not\equiv A_i$
then $r = prng(k)$
else
$k^{updated} = h^{\delta}(k)$
$r = h(k^{updated}||j)$
send reply r
r
if $r \equiv h(k^{updated}||j)$
then tag read
else tag unknown

Figure 2: The RIPP-FS protocol

and a time j (with $j > i$), the tag must verify that $h^{\delta}(A_j) = A_i$, where $\delta = j - i$. The collision resistance and the pre-image resistance properties of hash functions guarantee that only the entity that knows A_0 could also know the value A_j, since hash functions are one-way and evaluate the pre-image is practically unfeasible.

The same properties are exploited to guarantee the forward secrecy, since the same hash function is used to modify the shared key of a tag, in order to generate a different answer for each reading. In particular, before answering to an authenticated reader, the tag iterates several times the function h over its key, in order to update the key to the current time j. This ensures that if the adversary captures the tag and extracts the key k_j, she would be unable to evaluate the previous keys k_i, since $k_j = h^{\delta}(k_i)$, for $\delta = j - i > 0$.

To obtain key secrecy, the updated key is not straightforwardly sent to the reader, but instead it is used by the tag to generate a reply that is a hmac value, namely $h(k^{\mathit{updated}}||j)$. This ensures that the shared key is never transmitted during the execution of the protocol. Eavesdropping all the communications or sending malicious messages provides no information about the shared key. Finally, if the authentication value A_j does not pass the check $h^{\delta}(A_j) \equiv A_i$, then the tag will reply with a pseudo random number of the same length of a hmac value. In this way, any reader that does not know the expected hmac $h(k^{\mathit{updated}}||j)$ is unable to determine if the reply is a pseudo random number or a legitimate reply.

5.2 RIPP-FS fraglet specification

In this section, we introduce a fraglets specification of the RIPP-FS protocol executable in Maude. The reader can be specified as follows:

```
( ['kzero-l k0tag1] ,      --- tag shared key at time 0
  ['dl delta ] ,          --- delta
  ['tl 'tnow] ,            --- actual time
  ['tnl 'tl 'tnow] ,      --- actual time to be sent
     --- this builds the new authentication value
  [hashi 'authval max-delta 'auth0] ,
     --- this broadcasts the new authentication value
  [match 'authval mcast anyone 'authl],
     --- this broadcasts the actual time
  [mcast anyone 'tl 'tnow] ,
     --- this builds new shared key
  [matchs 'dl match 'kzero-l  hashi 'expkeytag-l ] ,
     --- this builds expected hmac
  [match 'expkeytag-l  match 'tl hmac 'exphmac] ,
     --- this verifies if the answer is OK, on
     --- reception of hmac-tag
  [matchs 'hmac-tag match 'ifoktag hv 'exphmac] ,
  [matchs 'hmac-tag match 'ifkotag hnv 'exphmac] ,
  ['ifoktag 'OK] ,
  ['ifkotag 'DOH]
) @ reader ;               --- reader's store
```

The reader's store contains an initial set of fraglets (protocol messages + protocol code) to perform its steps of the protocol: it can broadcast the authentication value and the time, it can evaluate the expected answer of the tag and verify the actual answer of the tag. It is worth noting the use of the fraglet instruction *hashi* that 1) produces the *authval* iterating *max-delta* times the hash function over the *auth0* value and 2) builds the new shared key iterating h over the initial *k0tag1* key. In particular, this evaluation originates from the combination with the *delta* fraglet message that corresponds to the reading number. Finally, we use the two hash verification instructions *hv* and *hnv*: only one of them will react with the *expmac* tag, that is *hv* if the check is passed, *hnv* otherwise.

The tag can be specified as follows:

```
( ['kzero-l k0tag1] ,    --- tag shared key at time 0
  ['dl delta ] ,               --- (t.old - t.now)
  ['authlast h(h(h(h('auth0)] , --- last auth. value
     --- this hashes delta times the received authval.
  [matchs 'dl matchs 'authl  hashi 'authnew ],
     --- this checks the authnew against authlast
  [matchs 'authlast match 'ifauthl hv 'authnew],
  [matchs 'authlast match 'ifnotauthl hnv 'authnew],
     --- if authenticated, builds new key know...
  ['ifauthl matchs 'dl match 'klast-l hashi 'know-l],
     --- ... and the hmac ...
  [matchs 'know-l  matchs 'tl hmac 'tagauth-l] ,
     --- ... and broadcasts the hmac
  [match 'tagauth-l mcast anyone 'hmac-tag] ,
     --- if not authenticated, broadcast garbage
  ['ifnotauthl mcast anyone h('PRNG)] )
) @ tag1 ;                --- RFID tag1's store
```

The initial fraglets in the tag's store start reacting with the reception of a fraglet with the *authl* tag. This ignites the reaction of the authentication value check and, then, the broadcast of the answer. It is worth noting that, if the hash verification does not succeed, then the pseudo-random value is sent, since the fraglet with *authlast* will react with the *hnv* fraglet. Otherwise, the fraglet reaction will produce the update of the key with the *hashi* cryptofraglet, the evaluation of the legitimate answer with the *hmac* cryptofraglet and the broadcast of the hmac value, with the *mcast anyone* fraglet.

5.3 Modelling security properties with fraglets

In this section, we show some security analysis over the fraglets specification of RIPP-FS. We highlight that the examples we depict in the following do not guarantee the fulfilment of the properties under all the possible configurations of the fraglets at stake. For instance, an exhaustive analysis of the protocol would necessitate to explore all the possible initial configurations of the fraglets stores representing the tag, the reader, and the adversary, as well as interactions among the potential universe of other fraglets stores. However, this kind of analysis is out of scope in this paper, whose main purpose is to show how the bio-inspired fraglet paradigm can model protocols as well as security properties.

5.3.1 Key secrecy against passive eavesdropping

To model the key secrecy against a passive eavesdropper, we introduce a malicious reader that eavesdrops on all the communications between a genuine reader and a legitimate tag. It silently exploits the inherently insecure wireless channel to collect the messages exchanged by the honest parties. Her aim is to collect the secret shared key of the tag or any other useful piece of information that would enable its disclosure.

To verify that the key is never disclosed, we leverage the Maude *search* command that explores all the possible derivatives of a given initial configuration. In particular, to model our adversary, we ask for any final state where the adversary knows the key, as follows:

```
select IS-KEY-SECRET .
search(
 ( empty @ malreader) ;    --- the store of mal. reader
 ( ...   @ reader) ;       --- the store of gen. reader
 ( ...   @ tag1)           --- the store of tag1
) =>! ([tl h(h(k0tag1)) t2], more @ malreader) ; rest .
```

With the above Maude excerpt we are looking for any possible evolution of the model in which the malreader's store contains a fraglet with the key of the tag: *t1* and *t2* can be any fraglet (even `nil` or a tag), while *more* denotes any other possible tag within the store; *rest* denotes the remaining fraglets of the model that correspond to the stores of *tag1* and the genuine *reader*. The omitted parts denoted with `...` are the protocol specification as in section 5.2. The outcome of the above specification is the following:

```
No solution.
states: 108855  rewrites: 713026 in 11828ms cpu ...
```

showing that all the possible branches of the model never reached a state in which the key secrecy was violated by the malicious reader.

We remark that the excerpt only describes one possible system configuration: other settings can be explored considering different evolutions of the tag key to be disclosed (for example *k0tag1*, *h(k0tag1)* and so on) or different sets of initial fraglets in the malicious reader's store (for example to make the eavesdropped data reacting with other fraglets).

5.3.2 Tag privacy against passive eavesdropping

Similarly to the key secrecy, we check tag privacy with the Maude *search* command. In particular, we model the prior knowledge of the malicious reader including in its store some *hmac*s collected during previous exchanges between the genuine reader and two different RFID tags (*tag1* and *tag2*). Moreover, we provide the malreader with a secret key (*k*), possibly extracted from a third tag. Finally, the malicious reader's store has a set of fraglets that can react with any eavesdropped *hmac*. We ask Maude to find any evolution of the system in which the cryptofraglet that successfully verifies the *hmac* reacts within the store of the malicious reader. Thus, we model the tag privacy as follows:

```
select IS-TAG-PRIVATE .
search(
 ( ['authl h('auth0) ],    --- the previous auth. value
   ['hmac-tag h(h(k1)||'told)], --- previous tag1 hmac
   ['hmac-tag h(h(k2)||'told)], --- previous tag2 hmac
   ['kzero-l k],            --- a key k =/= k1 and k2
   ...                      --- some other fraglets
   [matchs 'hmac-tag match 'ifoktag hv 'exphmac] ,
   [matchs 'hmac-tag match 'ifkotag hnv 'exphmac] ,
   ['ifoktag 'OK] ,
   ['ifkotag 'DOH]  @ malreader) ; --- malreader store
   ( ...   @ reader) ;      --- the store of gen. reader
   ( ...   @ tag1)          --- the store of tag1
) =>! (['OK], more @ malreader) ; rest .
```

With the *search* command specified as above, Maude will explore all possible evolutions where there the malreader store includes a fraglet with tag *'OK*, meaning that the *hv* tag reacted with the *exphmac* tag. The outcome is:

```
No solution.
states: 128790  rewrites: 951104 in 15508ms cpu ...
```

meaning that the malicious reader is unable to relate the eavesdropped *'hmac-tag* with any of the possible tags.

Note that other possible configurations can consider more fraglets in the malicious reader's store, in order to model a different prior knowledge or more hacking strategies, for example one that tries to disclose the information within the *hmac* fraglet and to relate them with any of the possible tags.

5.3.3 Forward secrecy against a tag capture

In order to test the forward secrecy property, we simply model the store of the malicious reader. We initialise a configuration where the attacker has eavesdropped some previous successfully acknowledged protocol readings of some tags (tag1 and tag2) and, some time later — eventually after some other readings, she violates the two tags and extracts all the cryptographic material inside them. We ask Maude if the adversary is able to relate any of the collected messages with any of the tags she compromised.

```
select FORWARD-SECRECY .
search(
 ( ['authl h(h(h(h(h(h('auth0))))))) ],--- auth. value
   ['hmac-tag h(h(k1)||'told)], --- previous hmac of tag1
   ['hmac-tag h(h(k2)||'told)],--- previous hmac of tag2
   ['kzero-l h(h(k1))],         --- key compromised tag1
   ['kzero-l h(h(k2))],         --- key compromised tag2
   ['dl 1 ] ,                   --- delta to be checked
   ['dl 2 ] ,                   --- delta to be checked
   ['dl 3 ] ,                   --- delta to be checked
   ['tl 'told] ,                --- old time
    --- try to build a new shared key
   [match 'dl match 'kzero-l  hashi 'expkeytag-l ] ,
    --- try to build a matching hmac
   [matchs 'expkeytag-l  match 'tl hmac 'exphmac] ,
    --- check if the hmac matches with the previous hmacs
   [matchs 'hmac-tag match 'ifoktag hv 'exphmac] ,
   [matchs 'hmac-tag match 'ifkotag hnv 'exphmac] ,
   ['ifoktag 'OK] ,
   ['ifkotag 'DOH]  @ malreader) --- malreader's store
 ) =>! (['OK] , more @ malreader) .
```

The fraglets in the store of the malicious reader can combine in many ways, realizing a kind of brute force attack against the collected hmac. This is the outcome of the Maude execution:

```
No solution.
states: 1180  rewrites: 3859 in 56ms cpu ...
```

Again, there was no evolution of the system in which the fraglets in the store were able to produce a *'OK* fraglet. This outcome is because of the one-way property of the hash function, since we are assuming that there exists no practical mechanism to have a pre-image of a hashed value.

We remind the reader that the complete Maude specification of cryptofraglets, as well as the Maude files of the example analyses shown in the paper, are available at
`mib.projects.iit.cnr.it/bict14/cryptofraglets.html`

6. RELATED WORK

The BIONETS EU project [1], *BIOlogically inspired NETwork and Services*, seeked inspiration from biological systems to provide a fully integrated network and service environment ,able to scale to large amounts of highly heterogeneous devices, and that is able to adapt and evolve in an autonomic way. The fraglets model has been extensively adopted in BIONETS and some security and trust extensions to the original model have become necessary to make it a running framework. That was the main reason why we started reasoning on cryptofraglets, first from a theoretical point of view, then realising a prototypal implementation of cryptofraglets in Maude to run some example analyses.

In the literature, there exist remarkable examples of the use of rewriting systems for modelling security protocols and analysing their properties, including, *e.g.*, [21, 8, 10]. Work in [21] shows how the Dolev-Yao model [7] of security protocol analysis may be-formalized using a notation based on multi-set rewriting with existential quantification and exemplifies the formalisation of subtle security properties. Under the same context, in [8] the authors analyze the complexity of the secrecy problem under various restrictions, showing that, even with a restricted size of messages, the secrecy problem is undecidable for the case of an unrestricted number of protocol roles and an unbounded number of freshly generated messages (so called nonces). The open complexity problem is indeed the main issue that one needs to explore for better defining limits and advantages in adopting cryptofraglets for security analysis. Indeed, as pointed out in the above sections, the example analyses that we have shown consider a limited number of actors and no generation of fresh messages. Finally, for page limits, we refer the interested reader to the tutorial in [10], describing the Maude-NRL Protocol Analyzer, a Maude-based tool for the analysis of cryptographic protocols. The tutorial also points to related work in the area of security protocols models and analysis, with an eye to rewriting systems.

7. CONCLUSIONS

In this paper, we renewed the cryptofraglets definition, enriching it with instructions for multicasting, hashing, and message authentication coding. Based on this new communication model, we also present a refined threat model where fraglets stores can engage in communication protocols together with either no intruder, or a passive intruder, or an active one. The executable specification of cryptofraglets in Maude has been refreshed too and modelling of properties as *forward secrecy* and *tag privacy* (intended as unlinkability of two, or more, answers coming from the same tag) has been presented as a proof of concept that paves the way for further investigation. As an example, we aim at extending the current work to deal with more general scenarios, where, e.g., the initial set of fraglets in the adversary store is not a priori fixed and considering more actors in a protocol run. Regarding the verification formal tool, we have exploited only a minimal set of the Maude capabilities, while in the future we intend to move to more sophisticated analysis tools among the Maude toolkit.

Acknowledgments

Work partly supported by the Registro.it project *My Information Bubble* MIB; the regional project *MyChoice*, funded by Tuscany Region; the MIUR funded PRIN project *Security Horizons*.

8. REFERENCES

[1] BIONETS website. http://www.bionets.eu/.

[2] I. Cervesato, N. Durgin, P. D. L. J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. CSFW-12*, pages 55–69. IEEE, 1999.

[3] E. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9(4):443–487, 2000.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[5] M. Conti, R. Di Pietro, L. V. Mancini, and A. Spognardi. RIPP-FS: an RFID Identification, Privacy Preserving protocol with Forward Secrecy. In *Pervasive Computing and Communications Workshops, 2007. Fifth Annual IEEE International Conference on*, pages 229–234. IEEE, 2007.

[6] M. Conti, R. Di Pietro, L. V. Mancini, and A. Spognardi. eRIPP-FS: Enforcing privacy and security in RFID. *Security and Communication Networks*, 3(1):58–70, 2010.

[7] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–208, 1983.

[8] N. A. Durgin, P. Lincoln, and J. C. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[9] L. Egidi and M. Petrocchi. Modelling a secure agent with team automata. In *Proc VODCA'04*, pages 119–134. Elsevier, 2005. ENTCS.

[10] S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer Berlin Heidelberg, 2009.

[11] R. Focardi and F. Martinelli. A uniform approach for the definition of security properties. In *Proc. FM'99*, volume 1708 of *LNCS*, pages 794–813. Springer, 1999.

[12] FRAGLETS website. http://www.fraglets.net.

[13] G. Lenzini, S. Gnesi, and D. Latella. Spider: a Security Model Checker. In *Proc. FAST'03*, pages 163–180, 2003. Informal proceedings.

[14] N. Lynch. I/O automaton models and proofs for shared-key communication systems. In *Proc. CSFW'99*, pages 14–31. IEEE, 1999.

[15] F. Martinelli and M. Petrocchi. Access control mechanisms for fraglets. In *BIONETICS*. ICST, 2007.

[16] F. Martinelli and M. Petrocchi. Signed and weighted trust credentials for fraglets. In *BIONETICS*. ICST, 2008.

[17] F. Martinelli and M. Petrocchi. Executable Specification of Cryptofraglets in Maude for Security Verification. In *BIONETICS*, pages 11–23, 2009.

[18] Maude website. http://maude.cs.uiuc.edu/.

[19] A. J. Menezes, S. A. Vanstone, and P. C. V. Orschot. *Handbook of Applied Cryptography 5th ed.* CRC Press, Inc., 2001.

[20] J. Meseguer. Research directions in rewriting logic. In *Computational Logic*, volume 165 of *LNCS*. Springer-Verlag, 1997.

[21] J. C. Mitchell. Multiset rewriting and security protocol analysis. In S. Tison, editor, *Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 19–22. Springer Berlin Heidelberg, 2002.

[22] M. Petrocchi. *Formal techniques for modeling and verifying secure procedures*. PhD thesis, University of Pisa, May 2005.

[23] M. Petrocchi. Crypto-fraglets. In *BIONETICS*. IEEE, 2006.

[24] C. Tschudin. Fraglets - a metabolistic execution model for communication protocols. In *Proc. AINS'03*, 2003.

[25] C. Tschudin and L. Yamamoto. A metabolic approach to protocol resilience. In *Proc. WAC'04, LNCS 3457*, pages 191–206. Springer, 2004.

[26] L. Yamamoto and C. Tschudin. Experiments on the automatic evolution of protocols using genetic programming. In *Proc. WAC'05, LNCS 3854*, pages 13–28. Springer, 2005.

[27] L. Yamamoto and C. Tschudin. Genetic evolution of protocol implementations and configurations. In *Proc. SelfMan'05*, 2005.