Automated Analysis of Security Protocols with Global State

Steve Kremer, Robert Künnemann

Automated analysis of security protocols with global state

Steve Kremer
INRIA Nancy - Grand'Est & Loria, France

Robert Künnemann
Department of Computer Science, TU Darmstadt, Germany
INRIA Paris - Rocquencourt, France

Abstract—**Security APIs, key servers and protocols that need to keep the status of transactions, require to maintain a global, non-monotonic state, e.g., in the form of a database or register. However, most existing automated verification tools do not support the analysis of such stateful security protocols – sometimes because of fundamental reasons, such as the encoding of the protocol as Horn clauses, which are inherently monotonic. A notable exception is the recent tamarin prover which allows specifying protocols as multiset rewrite (msr) rules, a formalism expressive enough to encode state. As multiset rewriting is a "low-level" specification language with no direct support for concurrent message passing, encoding protocols correctly is a difficult and error-prone process.**

We propose a process calculus which is a variant of the applied pi calculus with constructs for manipulation of a global state by processes running in parallel. We show that this language can be translated to msr rules whilst preserving all security properties expressible in a dedicated first-order logic for security properties. The translation has been implemented in a prototype tool which uses the tamarin prover as a backend. We apply the tool to several case studies among which a simplified fragment of PKCS#11, the Yubikey security token, and an optimistic contract signing protocol.

I. Introduction

Automated analysis of security protocols has been extremely successful. Using automated tools, flaws have been for instance discovered in the Google Single Sign On Protocol [1], in commercial security tokens implementing the PKCS#11 standard [2], and one may also recall Lowe's attack [3] on the Needham-Schroeder public key protocol 17 years after its publication. While efficient tools such as ProVerif [4], AVISPA [5] or Maude-NPA [6] exist, these tools fail to analyze protocols that require *non-monotonic global state*, i.e., some database, register or memory location that can be read and altered by different parallel threads. In particular ProVerif, one of the most efficient and widely used protocol analysis tools, relies on an abstraction that encodes protocols in first-order Horn clauses. This abstraction is well suited for the monotonic knowledge of an attacker (who never forgets), makes the tool extremely efficient for verifying an unbounded number of protocol sessions and allows to build on existing techniques for Horn clause resolution. However, Horn clauses are inherently monotonic: once a fact is true it cannot be set to false anymore. As a result, even though ProVerif's input language, a variant of the applied pi calculus [7], allows a priori encodings of a global memory, the abstractions performed by ProVerif introduce false attacks. In the ProVerif user manual [8, Section 6.3.3] such an encoding of memory cells and its limitations are indeed explicitly discussed: *"Due to the abstractions performed by ProVerif, such a cell is treated in an approximate way: all values written in the cell are considered as a set, and when one reads the cell, ProVerif just guarantees that the obtained value is one of the written values (not necessarily the last one, and not necessarily one written before the read)."* Some work [9], [10], [11] has nevertheless used ingenious encodings of mutable state in Horn clauses, but these encodings have limitations that we discuss below.

A prominent example where non-monotonic global state appears are security APIs, such as the RSA PKCS#11 standard [12], IBM's CCA [13] or the trusted platform module (TPM) [14]. They have been known to be vulnerable to logical attacks for some time [15], [16] and formal analysis has shown to be a valuable tool to identify attacks and find secure configurations. One promising paradigm for analyzing security APIs is to regard them as a participant in a protocol and use existing analysis tools. However, Herzog [17] already identified not accounting for mutable global state as a major barrier to the application of security protocol analysis tools to verify security APIs. Apart from security APIs many other protocols need to maintain databases: key servers need to store the status of keys, in optimistic contract signing protocols a trusted party maintains the status of a contract, RFID protocols maintain the status of tags and more generally websites may need to store the current status of transactions.

Our contributions: We propose a tool for analyzing protocols that may involve non-monotonic global state, relying on Schmidt *et al.*'s tamarin tool [18], [19] as a backend. We designed a new process calculus that extends the applied pi calculus by defining, in addition to the usual constructs for specifying concurrent processes, constructs for explicitly manipulating global state. This calculus serves as the tool's input language. The heart of our tool is a translation from this extended applied pi calculus to a set of multiset rewrite rules that can then be analyzed by tamarin which we use as a backend. We prove the correctness of this translation and show that it preserves all properties expressible in a dedicated first order logic for expressing security properties.

The full version of this paper including all proofs is available at http://sapic.gforge.inria.fr/

As a result, relying on the tamarin prover, we can analyze protocols without bounding the number of sessions, nor making any abstractions. Moreover it allows to model a wide range of cryptographic primitives by the means of equational theories. As the underlying verification problem is undecidable, tamarin may not terminate. However, it offers an interactive mode with a GUI which allows to manually guide the tool in its proof. Our specification language includes support for private channels, global state and locking mechanisms (which are crucial to write meaningful programs in which concurrent threads manipulate a common memory). The translation has been carefully engineered in order to favor termination by tamarin. We illustrate the tool on several case studies: a simple security API in the style of PKCS#11, a complex case study of the Yubikey security device, as well as several examples analyzed by other tools that aim at analyzing stateful protocols. In all of these case studies we were able to avoid restrictions that were necessary in previous works.

Related work: The most closely related work is the StatVerif tool by Arapinis *et al.* [9]. They propose an extension of the applied pi calculus, similar to ours, which is translated to Horn clauses and analyzed by the ProVerif tool. Their translation is sound but allows for false attacks, limiting the scope of protocols that can be analyzed. Moreover, StatVerif can only handle a finite number of memory cells: when analyzing an optimistic contract signing protocol this appeared to be a limitation and only the status of a single contract was modeled, providing a manual proof to justify the correctness of this abstraction. Finally, StatVerif is limited to the verification of secrecy properties. As illustrated by the Yubikey case study, our work is more general and we are able to analyze complex injective correspondance properties.

Mödersheim [10] proposed a language with support for sets together with an abstraction where all objects that belong to the same sets are identified. His language, which is an extension of the low level AVISPA intermediate format, is compiled into Horn clauses that are then analyzed, e. g., using ProVerif. His approach is tightly linked to this particular abstraction limiting the scope of applicability. Mödersheim also discusses the need for a more high-level specification level which we provide in this work.

There has also been work tailored to particular applications. In [20], Delaune *et al.* show by a dedicated hand proof that for analyzing PKCS#11 one may bound the message size. Their analysis still requires to artificially bound the number of keys. Similarly in spirit, Delaune *et al.* [11] give a dedicated result for analyzing protocols based on the TPM and its registers. However, the number of reboots (which reinitialize registers) needs to be limited.

Guttman [21] also extended the strand space model by adding support for state. While the protocol execution is modeled using the classical strand spaces model, state is modeled by a multiset of facts, and manipulated by multiset rewrite rules. The extended model has been used for analyzing by hand an optimistic contract signing protocol. As of now, protocol analysis in the strand space model with state has not been mechanized yet.

In the goal of relating different approaches for protocol analysis Bistarelli *et al.* [22] also proposed a translation from a process algebra to multiset rewriting: they do however not consider private channels, have no support for global state and assume that processes have a particular structure. These limitations significantly simplify the translation and its correctness proof. Moreover their work does not include any tool support for automated verification.

Obviously any protocol that we are able to analyze can be directly analyzed by the tamarin prover [18], [19] as the rules produced by our translation could have been given directly as an input to tamarin. Indeed, tamarin has already been used for analyzing a model of the Yubikey device [23], the case studies presented with Mödersheim's abstraction, as well as those presented with StatVerif. It is furthermore able to reproduce the aforementioned results on PKCS#11 [20] and the TPM [11] – moreover, it does so without bounding the number of keys, security devices, reboots, etc. Contrary to ProVerif, tamarin sometimes requires additional *typing lemmas* which are used to guide the proof. These lemmas need to be written by hand (but are proved automatically). In our case studies we also needed to provide a few such lemmas manually. In our opinion, an important disadvantage of tamarin is that protocols are modeled as a set of multiset rewrite rules. This representations is very low level and far away from actual protocol implementations, making it very difficult to model a protocol adequately. Encoding private channels, nested replications and locking mechanisms directly as multiset rewrite rules is a tricky and error prone task. As a result we observed that, in practice, the protocol models tend to be simplified. For instance, locking mechanisms are often omitted, modeling protocol steps as a single rule and making them effectively atomic. Such more abstract models may obscure issues in concurrent protocol steps and increase the risk of implicitly excluding attacks in the model that are well possible in a real implementation, e. g., race conditions. Using a more high-level specification language, such as our process calculus, arguably eases protocol specification and overcomes some of these risks.

II. Preliminaries

Terms and equational theories: As usual in symbolic protocol analysis we model messages by abstract terms. Therefore we define an order-sorted term algebra with the sort msg and two incomparable subsorts pub and $fresh$. For each of these subsorts we assume a countably infinite set of names, FN for fresh names and PN for public names. Fresh names will be used to model cryptographic keys and nonces while public names model publicly known values. We furthermore assume a countably infinite set of variables for each sort s, $\mathcal{V}_s$ and let $\mathcal{V}$ be the union of the set of variables for all sorts. We write $u : s$ when the name or variable u is of sort s. Let Σ be a signature, i.e., a set of function symbols, each with an arity. We write f/n when function symbol f is of arity n. We denote by $\mathcal{T}_\Sigma$ the set of well-sorted terms built over Σ, PN, FN and $\mathcal{V}$. For a term t we denote by $names(t)$, respectively $vars(t)$ the set of names, respectively variables, appearing in t. The set of ground terms, i.e., terms without variables, is denoted by $\mathcal{M}_\Sigma$. When Σ is fixed or clear from the context we often omit it and simply write $\mathcal{T}$ for $\mathcal{T}_\Sigma$ and $\mathcal{M}$ for $\mathcal{M}_\Sigma$.

We equip the term algebra with an equational theory E, that is a finite set of equations of the form $M = N$ where $M, N \in \mathcal{T}$. From the equational theory we define the binary relation $=_E$ on terms, which is the smallest equivalence relation containing equations in E that is closed under application of function symbols, bijective renaming of names and substitution of variables by terms of the same sort. Furthermore, we require E to distinguish different fresh names, i. e., $\forall a, b \in FN : a \neq b \Rightarrow a \neq_E b$.

Example. Symmetric encryption can be modelled using a signature

$$\Sigma = \{\, senc/2, sdec/2, encCor/2, true/0 \,\}$$

and an equational theory defined by

$$sdec(senc(m,k),k) = m \quad encCor(senc(x,y),y) = true$$

The last equation allows to check whether a term can be correctly decrypted with a certain key.

For the rest of the paper we assume that E refers to some fixed equational theory and that the signature and equational theory always contain symbols and equations for pairing and projection, i.e., $\{\langle .,. \rangle, \mathsf{fst}, \mathsf{snd}\} \subseteq \Sigma$ and equations $\mathsf{fst}(\langle x,y \rangle) = x$ and $\mathsf{snd}(\langle x,y \rangle) = y$ are in E. We will sometimes use $\langle x_1, x_2, \ldots, x_n \rangle$ as a shortcut for $\langle x_1, \langle x_2, \langle \ldots, \langle x_{n-1}, x_n \rangle \ldots \rangle$.

We also use the usual notion of positions for terms. A position p is a sequence of positive integers and $t|_p$ denotes the subterm of t at position p.

Facts: We also assume an unsorted signature Σ_{fact}, disjoint from Σ. The set of *facts* is defined as

$$\mathcal{F} := \{F(t_1, \ldots, t_k) \mid t_i \in \mathcal{T}_\Sigma, F \in \Sigma_{fact} \text{ of arity } k\}.$$

Facts will be used both to annotate protocols, by the means of events, and for defining multiset rewrite rules. We partition the signature Σ_{fact} into *linear* and *persistent* fact symbols. We suppose that Σ_{fact} always contains a unary, persistent symbol $!\mathsf{K}$ and a linear, unary symbol Fr. Given a sequence or set of facts S we denote by $lfacts(S)$ the multiset of all linear facts in S and $pfacts(S)$ the set of all persistent facts in S. By notational convention facts whose identifier starts with '!' will be persistent. $\mathcal{G}$ denotes the set of ground facts, i.e., the set of facts that does not contain variables. For a fact f we denote by $ginsts(f)$ the set of ground instances of f. This notation is also lifted to sequences and sets of facts as expected.

Substitutions: A substitution σ is a partial function from variables to terms. We suppose that substitutions are well-typed, i.e., they only map variables of sort s to terms of sort s, or of a subsort of s. We denote by $\sigma = \{^{t_1}/_{x_1}, \ldots, ^{t_n}/_{x_n}\}$ the substitution whose domain is $\mathbf{D}(\sigma) = \{x_1, \ldots, x_n\}$ and which maps x_i to t_i. As usual we homomorphically extend σ to apply to terms and facts and use a postfix notation to denote its application, e.g., we write $t\sigma$ for the application of σ to the term t. A substitution σ is grounding for a term t if $t\sigma$ is ground. Given function g we let $g(x) = \bot$ when $x \notin \mathbf{D}(x)$. When $g(x) = \bot$ we say that g is undefined for x. We define the function $f := g[a \mapsto b]$ with $\mathbf{D}(f) = \mathbf{D}(g) \cup \{\, a \,\}$ as $f(a) := b$ and $f(x) := g(x)$ for $x \neq a$.

Sets, sequences and multisets: We write $\mathbb{N}_n$ for the set $\{1, \ldots, n\}$. Given a set S we denote by S^* the set of finite sequences of elements from S and by $S^\#$ the set of finite multisets of elements from S. We use the superscript $^\#$ to annotate usual multiset operation, e.g. $S_1 \cup^\# S_2$ denotes the multiset union of multisets S_1, S_2. Given a multiset S we denote by $set(S)$ the set of elements in S. The sequence consisting of elements $e_1, \ldots, e_n$ will be denoted by $[e_1, \ldots, e_n]$ and the empty sequence is denoted by $[]$. We denote by $|S|$ the length, i.e., the number of elements of the sequence. We use $\cdot$ for the operation of adding an element either to the start or to the end, e.g., $e_1 \cdot [e_2, e_3] = [e_1, e_2, e_3] = [e_1, e_2] \cdot e_3$. Given a sequence S, we denote by $idx(S)$ the set of positions in S, i.e., $\mathbb{N}_n$ when S has n elements, and for $i \in idx(S)$ S_i denotes the ith element of the sequence. Set membership modulo E is denoted by $\in_E$ and defined as $e \in_E S$ if $\exists e' \in S.\ e' =_E e$. $\subset_E$ and $=_E$ are defined for sets in a similar way. Application of substitutions are lifted to sets, sequences and multisets as expected. By abuse of notation we sometimes interpret sequences as sets or multisets.

III. A CRYPTOGRAPHIC PI CALCULUS WITH EXPLICIT STATE

A. Syntax and informal semantics

Our calculus is a variant of the applied pi calculus [7]. In addition to the usual operators for concurrency, replication, communication and name creation, it offers several constructs for reading and updating an explicit global state. The grammar for processes is described in Figure 1.

0 denotes the terminal process. $P \mid Q$ is the parallel execution of processes P and Q and $!P$ the replication of P, allowing an unbounded number of sessions in protocol executions. The construct $\nu n; P$ binds the name n in P and

$\langle M,N\rangle$ ::= $x, y, z \in \mathcal{V}$
| $p \in PN$
| $n \in FN$
| $f(M_1,\ldots,M_n)$ ($f \in \Sigma$ of arity n)

$\langle P,Q\rangle$ ::= 0
| $P \mid Q$
| ! P
| νn; P
| out(M, N); P
| in(M, N); P
| if M=N then P [else Q]
| event F ; P ($F \in \mathcal{F}$)
| insert M,N; P
| delete M; P
| lookup M as x in P [else Q]
| lock M; P
| unlock M; P
| $[L] \dashv\!\![A]\!\!\rightarrowtail [R]; P$ ($L, R, A \in \mathcal{F}^*$)

Figure 1. Syntax

models the generation of a fresh, random value. Processes out(M, N); P and in(M, N); P represent the output, respectively input, of message N on channel M. Readers familiar with the applied pi calculus [7] may note that we opted for the possibility of pattern matching in the input construct, rather than merely binding the input to a variable x. The process if M=N then P else Q will execute P if $M =_E N$ and Q otherwise. The event construct is merely used for annotating processes and will be useful for stating security properties. For readability we sometimes omit to write else Q when Q is 0, as well as trailing 0 processes.

The remaining constructs are used for manipulating state and are new compared to the applied pi calculus. We offer two different mechanisms for state. The first construct is *functional* and allows to associate a value to a key. The construct insert M,N binds the value N to a key M. Successive inserts allow to change this binding. The delete M operation simply "undefines" the mapping for the key M. The lookup M as x in P else Q allows to retrieve the value associated to M, binding it to the variable x in P. If the mapping is undefined for M the process behaves as Q. The lock and unlock constructs allow to gain exclusive access to a resource M. This is essential for writing protocols where parallel processes may read and update a common memory. We additionally offer another kind of global state in form of a multiset of ground facts, as opposed to the previously introduced functional store. This multiset can be altered using the construct $[L] \dashv\!\![A]\!\!\rightarrowtail [R]; P$, which tries to match each fact in the sequence L to facts in the current multiset and, if successful, adds the corresponding instance of facts R to the store. The facts A are used as annotations in a similar way to events. The purpose of this construct is to provide access to the underlying notion of state in tamarin, but we stress that it is distinct from the previously introduced functional state, and its use is only advised to expert users. We allow this "low-level" form of state manipulation in addition to the *functional* state, as it offers a great flexibility and has shown useful in one of our case studies. This style of state manipulation is similar to the state extension in the strand space model [21] and the underlying specification language of the tamarin tool [18], [19]. Note that, even though those stores are distinct (which is a restriction imposed by our translation), data can be moved from one to another, for example as follows: lookup 'store1' as x in $[] \dashv\!\![\,]\!\!\rightarrowtail [\mathsf{store2}(x)]$.

In the following example, which will serve as our running example, we model a security API that, even though much simplified, illustrates the most salient issues that occur in the analysis of security APIs such as PKCS#11 [20], [2].

Example. We consider a security device that allows the creation of keys in its secure memory. The user can access the device via an API. If he creates a key, he obtains a handle, which he can use to let the device perform operations on his behalf. For each handle the device also stores an attribute which defines what operations are permitted for this handle. The goal is that the user can never gain knowledge of the key, as the user's machine might be compromised. We model the device by the following process (we use out(m) as a shortcut for out(c, m) for a public channel c):

$$!P_{new} \mid !P_{set} \mid !P_{dec} \mid !P_{wrap}, \text{ where}$$

```
$P_{new}$ := $\nu h$; $\nu k$; event NewKey($h$,$k$);
        insert ⟨'key', $h$⟩, $k$;
        insert ⟨'att', $h$⟩, 'dec'; out($h$)
```

In the first line, the device creates a new handle h and a key k and, by the means of the event NewKey(h, k), logs the creation of this key. It then stores the key that belongs to the handle by associating the pair ⟨'key', h⟩ to the value of the key k. In the next line, ⟨'att', h⟩ is associated to a public constant 'dec'. Intuitively, we use the public constants 'key' and 'att' to distinguish two databases. The process

P_{set} := in(h); insert ⟨'att',h⟩, 'wrap'

allows the attacker to change the attribute of a key from the initial value 'dec' to another value 'wrap'. If a handle has the 'dec' attribute set, it can be used for decryption:

```
$P_{dec}$ := in(⟨$h$,$c$⟩); lookup ⟨'att',$h$⟩ as $a$ in
        if $a$='dec' then
            lookup ⟨'key', $h$⟩ as $k$ in
                if $encCor(c,k)$=$true$ then
                    event DecUsing($k$,$sdec(c,k)$);
                    out($sdec(c,k)$)
```

The first lookup stores the value associated to ⟨'att', h⟩ in a. The value is compared against 'dec'. If the comparison and another lookup for the associated key value k succeeds, we check whether decryption succeeds and, if so, output the plaintext.

If a key has the 'wrap' attribute set, it might be used to encrypt the value of a second key:

$$
\begin{aligned}
P_{wrap} := \ & \mathsf{in}(\langle h_1,h_2\rangle);\ \mathsf{lookup}\ \langle\text{'att'},h_1\rangle\ \mathsf{as}\ a_1\ \mathsf{in}\\
& \quad \mathsf{if}\ \ a_1{=}\text{'wrap'}\ \mathsf{then}\\
& \qquad \mathsf{lookup}\ \langle\text{'key'},h_1\rangle\ \mathsf{as}\ k_1\ \mathsf{in}\\
& \qquad\quad \mathsf{lookup}\ \langle\text{'key'},\ h_2\rangle\ \mathsf{as}\ k_2\ \mathsf{in}\\
& \qquad\qquad \mathsf{event}\ \mathrm{Wrap}(k_1,k_2);\\
& \qquad\qquad \mathsf{out}(senc(k_2,k_1))
\end{aligned}
$$

The bound names of a process are those that are bound by νn. We suppose that all names of sort *fresh* appearing in the process are under the scope of such a binder. Free names must be of sort *pub*. A variable x can be bound in three ways: *(i)* by the construct $\mathsf{lookup}\ M\ \mathsf{as}\ x$, or *(ii)* $x \in vars(N)$ in the construct $\mathsf{in}(M,N)$ and x is not under the scope of a previous binder, *(iii)* $x \in vars(L)$ in the construct $[L] -[A]\rightarrow [R]$ and x is not under the scope of a previous binder. While the construct $\mathsf{lookup}\ M\ \mathsf{as}\ x$ always acts as a binder, the input and $[L] -[A]\rightarrow [R]$ constructs do not rebind an already bound variable but perform pattern matching. For instance in the process

$$P = \mathsf{in}(\mathsf{c},f(x));\ \mathsf{in}(\mathsf{c},g(x))$$

x is bound by the first input and pattern matched in the second. It might seem odd that lookup acts as a binder, while input does not. We justify this decision as follows: as P_{dec} and P_{wrap} in the previous example show, lookups appear often after input was received. If lookup were to use pattern matching, the following process

$$P = \mathsf{in}(c,x);\ \mathsf{lookup}\ \text{'store'}\ \mathsf{as}\ x\ \mathsf{in}\ P'$$

might unexpectedly perform a check if 'store' contains the message given by the adversary, instead of binding the content of 'store' to x, due to an undetected clash in the naming of variables.

A process is ground if it does not contain any free variables. We denote by $P\sigma$ the application of the homomorphic extension of the substitution σ to P. As usual we suppose that the substitution only applies to free variables. We sometimes interpret the syntax tree of a process as a term and write $P|_p$ to refer to the subprocess of P at position p (where $|$, if and lookup are interpreted as binary symbols, all other constructs as unary).

B. Semantics

Frames and deduction: Before giving the formal semantics of our calculus we introduce the notions of frame and deduction. A *frame* consists of a set of fresh names $\tilde{n}$ and a substitution σ and is written $\nu\tilde{n}.\sigma$. Intuitively a frame represents the sequence of messages that have been observed by an adversary during a protocol execution and secrets $\tilde{n}$ generated by the protocol, a priori unknown to the adversary. Deduction models the capacity of the adversary to compute new messages from the observed ones.

Definition 1 (Deduction)**.** *We define the deduction relation $\nu\tilde{n}.\sigma \vdash t$ as the smallest relation between frames and terms defined by the deduction rules in Figure 2.*

Example. If one key is used to wrap a second key, then, if the intruder learns the first key, he can deduce the second. For $\tilde{n} = k_1, k_2$ and $\sigma = \{ {}^{senc(k_2,k_1)}/_{x_1}, {}^{k_1}/_{x_2} \}$, $\nu\tilde{n}.\sigma \vdash k_2$, as witnessed by the proof tree given in Figure 3.

Operational semantics: We can now define the operational semantics of our calculus. The semantics is defined by a labelled transition relation between process configurations. A *process configuration* is a 6-tuple $(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\mathrm{MS}}, \mathcal{P}, \sigma, \mathcal{L})$ where

- $\mathcal{E} \subseteq FN$ is the set of fresh names generated by the processes;
- $\mathcal{S} : \mathcal{M}_\Sigma \to \mathcal{M}_\Sigma$ is a partial function modeling the functional store;
- $\mathcal{S}^{\mathrm{MS}} \subseteq \mathcal{G}^\#$ is a multiset of ground facts and models the multiset of stored facts;
- $\mathcal{P}$ is a multiset of ground processes representing the processes executed in parallel;
- σ is a ground substitution modeling the messages output to the environment;
- $\mathcal{L} \subseteq \mathcal{M}_\Sigma$ is the set of currently acquired locks.

The transition relation is defined by the rules described in Figure 4. Transitions are labelled by sets of ground facts. For readability we omit empty sets and brackets around singletons, i.e., we write $\to$ for $\xrightarrow{\emptyset}$ and $\xrightarrow{f}$ for $\xrightarrow{\{f\}}$. We write $\to^*$ for the reflexive, transitive closure of $\to$ (the transitions that are labelled by the empty sets) and write $\overset{f}{\Rightarrow}$ for $\to^* \xrightarrow{f} \to^*$. We can now define the set of traces, i.e., possible executions, that a process admits.

Definition 2 (Traces of P)**.** *Given a ground process P we define the* set of traces of P *as*

$$
\begin{aligned}
traces^{pi}(P) = \Big\{ & [F_1, \ldots, F_n] \mid (\emptyset,\emptyset,\emptyset,\{P\},\emptyset,\emptyset)\\
& \overset{F_1}{\Longrightarrow} (\mathcal{E}_1, \mathcal{S}_1, \mathcal{S}_1^{\mathrm{MS}}, \mathcal{P}_1, \sigma_1, \mathcal{L}_1)\\
& \overset{F_2}{\Longrightarrow} \ldots \overset{F_n}{\Longrightarrow} (\mathcal{E}_n, \mathcal{S}_n, \mathcal{S}_n^{\mathrm{MS}}, \mathcal{P}_n, \sigma_n, \mathcal{L}_n) \Big\}
\end{aligned}
$$

Example. In Figure 5 we display the transitions that illustrate how the first key is created on the security device in our running example and witness that $[\mathrm{NewKey}(h', k')] \in traces^{pi}(P)$.

IV. Labelled multiset rewriting

We now recall the syntax and semantics of labelled multiset rewriting rules, which constitute the input language of the tamarin tool [18].

$$\frac{a \in FN \cup PN \quad a \notin \tilde{n}}{\nu\tilde{n}.\sigma \vdash a}\ \textsc{DName} \qquad \frac{\nu\tilde{n}.\sigma \vdash t \quad t =_E t'}{\nu\tilde{n}.\sigma \vdash t'}\ \textsc{DEq}$$

$$\frac{x \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash x\sigma}\ \textsc{DFrame} \qquad \frac{\nu\tilde{n}.\sigma \vdash t_1 \cdots \nu\tilde{n}.\sigma \vdash t_n \quad f \in \Sigma^k}{\nu\tilde{n}.\sigma \vdash f(t_1, \ldots, t_n)}\ \textsc{DAppl}$$

Figure 2. Deduction rules.

$$\frac{\dfrac{\dfrac{x_1 \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash \mathit{senc}(k_2, k_1)} \quad \dfrac{x_2 \in \mathbf{D}(\sigma)}{\nu\tilde{n}.\sigma \vdash k_1}}{\nu\tilde{n}.\sigma \vdash \mathit{sdec}(\mathit{senc}(k_2, k_1), k_1)} \quad \mathit{sdec}(\mathit{senc}(k_2, k_1), k_1) =_E k_2}{\nu\tilde{n}.\sigma \vdash k_2}$$

Figure 3. Proof tree witnessing that $\nu\tilde{n}.\sigma \vdash k_2$

Standard operations:

$$\begin{array}{rcl}
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{0\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P}, \sigma, \mathcal{L}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P|Q\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P, Q\}, \sigma, \mathcal{L}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{!P\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{!P, P\}, \sigma, \mathcal{L}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\nu a; P\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E} \cup \{a'\}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\{a'/a\}\}, \sigma, \mathcal{L}) \\
 & & \hfill \text{if } a' \text{ is fresh} \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P}, \sigma, \mathcal{L}) & \xrightarrow{K(M)} & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P}, \sigma, \mathcal{L}) \quad \text{if } \nu\mathcal{E}.\sigma \vdash M \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{out}(M, N); P\}, \sigma, \mathcal{L}) & \xrightarrow{K(M)} & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\}, \sigma \cup \{^{N}/_{x}\}, \mathcal{L}) \\
 & & \hfill \text{if } x \text{ is fresh and } \nu\mathcal{E}.\sigma \vdash M \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{in}(M, N); P\}, \sigma, \mathcal{L}) & \xrightarrow{K(\langle M, N\tau\rangle)} & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\tau\}, \sigma, \mathcal{L}) \\
 & \multicolumn{2}{r}{\text{if } \exists\tau.\ \tau \text{ is grounding for } N, \nu\mathcal{E}.\sigma \vdash M, \nu\mathcal{E}.\sigma \vdash N\tau} \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{out}(M, N); P, \text{in}(M', N'); Q\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup \{P, Q\tau\}, \sigma, \mathcal{L}) \\
 & \multicolumn{2}{r}{\text{if } M =_E M' \text{ and } \exists\tau.\ N =_E N'\tau \text{ and } \tau \text{ grounding for } N'} \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup \{\text{if } M = N \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup \{P\}, \sigma, \mathcal{L}) \quad \text{if } M =_E N \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup \{\text{if } M = N \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup \{Q\}, \sigma, \mathcal{L}) \quad \text{if } M \neq_E N \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup \{\text{event}(F);\ P\}, \sigma, \mathcal{L}) & \xrightarrow{F} & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup \{P\}, \sigma, \mathcal{L})
\end{array}$$

Operations on global state:

$$\begin{array}{rcl}
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{insert } M, N;\ P\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}[M \mapsto N], \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{delete } M;\ P\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}[M \mapsto \bot], \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q\ \}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\{V/x\}\}, \sigma, \mathcal{L}) \\
 & & \hfill \text{if } \mathcal{S}(N) =_E V \text{ is defined and } N =_E M \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q\ \}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{Q\}, \sigma, \mathcal{L}) \\
 & & \hfill \text{if } \mathcal{S}(N) \text{ is undefined for all } N =_E M \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{lock } M;\ P\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \cup \{\ M\ \}) \\
 & & \hfill \text{if } M \notin_E \mathcal{L} \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{\text{unlock } M;\ P\}, \sigma, \mathcal{L}) & \longrightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \setminus \{\ M' \mid M' =_E M\ \}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}}, \mathcal{P} \cup^{\#} \{[l \,\text{-}[a]\!\!\rightarrow r];\ P\}, \sigma, \mathcal{L}) & \xrightarrow{a'} & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\text{MS}} \setminus \mathit{lfacts}(l') \cup^{\#} r', \mathcal{P} \cup^{\#} \{\ P\tau\ \}, \sigma, \mathcal{L}) \\
\text{if } \exists\tau, l', a', r'. & \multicolumn{2}{l}{\tau \text{ grounding for } l \,\text{-}[a]\!\!\rightarrow r, l' \,\text{-}[a']\!\!\rightarrow r' =_E (l \,\text{-}[a]\!\!\rightarrow r)\tau,} \\
 & \multicolumn{2}{l}{\mathit{lfacts}(l') \subseteq^{\#} \mathcal{S}^{\text{MS}}, \mathit{pfacts}(l') \subset \mathcal{S}^{\text{MS}}}
\end{array}$$

Figure 4. Operational semantics

$$(\emptyset,\emptyset,\emptyset,\{\,\underbrace{!P_{new}, !P_{set}\ |!P_{dec}\ |!P_{wrap}}_{=:\mathcal{P}'}\,\}^{\#},\emptyset,\emptyset) \to (\emptyset,\emptyset,\emptyset,\{\,P_{new}\,\}^{\#}\cup^{\#}\mathcal{P}',\emptyset,\emptyset)$$

$$\to\ (\emptyset,\emptyset,\emptyset,\{\,\nu h;\nu k;\text{event NewKey}(h,k);\dots\}^{\#}\cup^{\#}\mathcal{P}',\emptyset,\emptyset)$$

$$\to^{*}(\{\,h',k'\,\},\emptyset,\emptyset,\{\,\text{event NewKey}(h',k');\dots\}^{\#}\cup^{\#}\mathcal{P}',\emptyset,\emptyset)$$

$$\xrightarrow{\text{NewKey}(h',k')}\ (\{\,h',k'\,\},\emptyset,\emptyset,\{\,\text{insert }\langle\text{'key'},\ h'\rangle,\ k';\dots\}^{\#}\cup^{\#}\mathcal{P}',\emptyset,\emptyset)$$

$$\to^{*}(\{\,h',k'\,\},\mathcal{S},\emptyset,\{\,\text{out}(h');0\,\}^{\#}\cup^{\#}\mathcal{P}',\emptyset,\emptyset)\to^{*}\ (\{\,h',k'\,\},\mathcal{S},\emptyset,\mathcal{P}',\{\,{}^{h'}/_{x_1}\,\},\emptyset)$$

$$\text{where } \mathcal{S}(\langle\text{'key'},h'\rangle)=k' \text{ and } \mathcal{S}(\langle\text{'att'},h'\rangle)=\text{'dec'}.$$

Figure 5. Example of transitions modelling the creation of a key on a PKCS#11-like device

Definition 3 (Multiset rewrite rule)**.** *A labelled multiset rewrite rule ri is a triple (l,a,r), $l,a,r\in\mathcal{F}^*$, written $l\ {-}[a]{\rightarrow}\ r$. We call $l=prems(ri)$ the premises, $a=actions(ri)$ the actions, and $r=conclusions(ri)$ the conclusions of the rule.*

Definition 4 (Labelled multiset rewriting system)**.** *A labelled multiset rewriting system is a set of labelled multiset rewrite rules R, such that each rule $l\ {-}[a]{\rightarrow}\ r\in R$ satisfies the following conditions:*

- *l,a,r do not contain fresh names*
- *r does not contain* Fr*-facts*

A labelled multiset rewriting system is called well-formed, if additionally

- *for each $l'\ {-}[a']{\rightarrow}\ r'\in_E ginsts(l\ {-}[a]{\rightarrow}\ r)$ we have that $\cap_{r''=_E r'}names(r'')\cap FN\subseteq\cap_{l''=_E l'}names(l'')\cap FN$.*

We define one distinguished rule FRESH which is the only rule allowed to have Fr-facts on the right-hand side

$$\text{FRESH}:[]\ {-}[]{\rightarrow}\ [\mathsf{Fr}(x:\mathit{fresh})]$$

The semantics of the rules is defined by a labelled transition relation.

Definition 5 (Labelled transition relation)**.** *Given a multiset rewriting system R we define the* labeled transition relation *$\to_R\subseteq\mathcal{G}^{\#}\times\mathcal{P}(\mathcal{G})\times\mathcal{G}^{\#}$ as*

$$S\xrightarrow{a}_R((S\setminus^{\#}lfacts(l))\cup^{\#}r)$$

if and only if $l\ {-}[a]{\rightarrow}\ r\in_E ginsts(R\cup\text{FRESH})$, $lfacts(l)\subseteq^{\#}S$ and $pfacts(l)\subseteq S$.

Definition 6 (Executions)**.** *Given a multiset rewriting system R we define its set of executions as*

$$exec^{msr}(R)=\Big\{\emptyset\xrightarrow{A_1}_R\dots\xrightarrow{A_n}_R S_n\ |\ \forall a,i,j\colon 0\le i\ne j<n.\ (S_{i+1}\setminus^{\#}S_i)=\{\mathsf{Fr}(a)\}\Rightarrow(S_{j+1}\setminus^{\#}S_j)\ne\{\mathsf{Fr}(a)\}\Big\}$$

The set of executions consists of transition sequences that respect freshness, i. e., for a given name a the fact $\mathsf{Fr}(a)$ is only added once, or in other words the rule FRESH is at most fired once for each name. We define the set of traces in a similar way as for processes.

Definition 7 (Traces)**.** *The set of traces is defined as*

$$traces^{msr}(R)=\Big\{[A_1,\dots,A_n]\ |\ \forall\,0\le i\le n.\ A_i\ne\emptyset \text{ and } \emptyset\overset{A_1}{\Longrightarrow}_R\dots\overset{A_n}{\Longrightarrow}_R S_n\in exec^{msr}(R)\Big\}$$

where $\overset{A}{\Longrightarrow}_R$ is defined as $\xrightarrow{\emptyset}{}^{*}_R\xrightarrow{A}_R\xrightarrow{\emptyset}{}^{*}_R$.

Note that both for processes and multiset rewrite rules the set of traces is a sequence of sets of facts.

V. Security Properties

In the tamarin tool [18] security properties are described in an expressive two-sorted first-order logic. The sort $temp$ is used for time points, $\mathcal{V}_{temp}$ are the temporal variables.

Definition 8 (Trace formulas)**.** *A trace atom is either false $\bot$, a term equality $t_1\approx t_2$, a timepoint ordering $i\lessdot j$, a timepoint equality $i\doteq j$, or an action $F@i$ for a fact $F\in\mathcal{F}$ and a timepoint i. A trace formula is a first-order formula over trace atoms.*

As we will see in our case studies this logic is expressive enough to analyze a variety of security properties, including complex injective correspondence properties.

To define the semantics, let each sort s have a domain $\mathbf{D}(s)$. $\mathbf{D}(temp)=\mathcal{Q}$, $\mathbf{D}(msg)=\mathcal{M}$, $\mathbf{D}(fresh)=FN$, and $\mathbf{D}(pub)=PN$. A function $\theta:\mathcal{V}\to\mathcal{M}\cup\mathcal{Q}$ is a valuation if it respects sorts, that is, $\theta(\mathcal{V}_s)\subset\mathbf{D}(s)$ for all sorts s. If t is a term, $t\theta$ is the application of the homomorphic extension of θ to t.

Definition 9 (Satisfaction relation)**.** *The satisfaction relation $(tr,\theta)\vDash\varphi$ between trace tr, valuation θ and trace formula*

φ is defined as follows:

$$\begin{array}{lll}
(tr,\theta) \vDash \bot & \textit{never} & \\
(tr,\theta) \vDash F@i & \textit{iff} & \theta(i) \in idx(tr) \textit{ and } F\theta \in_E tr_{\theta(i)} \\
(tr,\theta) \vDash i \lessdot j & \textit{iff} & \theta(i) < \theta(j) \\
(tr,\theta) \vDash i \doteq j & \textit{iff} & \theta(i) = \theta(j) \\
(tr,\theta) \vDash t_1 \approx t_2 & \textit{iff} & t_1\theta =_E t_2\theta \\
(tr,\theta) \vDash \neg\varphi & \textit{iff} & \textit{not } (tr,\theta) \vDash \varphi \\
(tr,\theta) \vDash \varphi_1 \wedge \varphi_2 & \textit{iff} & (tr,\theta) \vDash \varphi_1 \textit{ and } (tr,\theta) \vDash \varphi_2 \\
(tr,\theta) \vDash \exists x : s.\varphi & \textit{iff} & \textit{there is } u \in \mathbf{D}(s) \textit{ such that} \\
& & (tr, \theta[x \mapsto u]) \vDash \varphi
\end{array}$$

When φ is a ground formula we sometimes simply write $tr \vDash \varphi$ as the satisfaction of φ is independent of the valuation.

Definition 10 (Validity, satisfiability)**.** *Let $Tr \subseteq (\mathcal{P}(\mathcal{G}))^*$ be a set of traces. A trace formula φ is said to be* valid *for Tr, written $Tr \vDash^\forall \varphi$, if for any trace $tr \in Tr$ and any valuation θ we have that $(tr,\theta) \vDash \varphi$.*

A trace formula φ is said to be satisfiable *for Tr, written $Tr \vDash^\exists \varphi$, if there exist a trace $tr \in Tr$ and a valuation θ such that $(tr,\theta) \vDash \varphi$.*

Note that $Tr \vDash^\forall \varphi$ iff $Tr \nvDash^\exists \neg\varphi$. Given a multiset rewriting system R we say that φ is valid, written $R \vDash^\forall \varphi$, if $traces^{msr}(R) \vDash^\forall \varphi$. We say that φ is satisfied in R, written $R \vDash^\exists \varphi$, if $traces^{msr}(R) \vDash^\exists \varphi$. Similarly, given a ground process P we say that φ is valid, written $P \vDash^\forall \varphi$, if $traces^{pi}(P) \vDash^\forall \varphi$, and that φ is satisfied in P, written $P \vDash^\exists \varphi$, if $traces^{pi}(P) \vDash^\exists \varphi$.

Example. The following trace formula expresses secrecy of keys generated on the security API, which we introduced in Section III.

$$\neg(\exists h,k\colon msg,\ i,j\colon temp.\ \mathrm{NewKey}(h,k)@i \wedge \mathrm{K}(k)@j)$$

VI. A TRANSLATION FROM PROCESSES INTO MULTISET REWRITE RULES

In this section we define a translation from a process P into a set of multiset rewrite rules $[\![P]\!]$ and a translation on trace formulas such that $P \models^\forall \varphi$ if and only if $[\![P]\!] \models^\forall [\![\varphi]\!]$. Note that the result also holds for satisfiability, as an immediate consequence. For a rather expressive subset of trace formulas (see [18] for the exact definition of the fragment), checking whether $[\![P]\!] \models^\forall [\![\varphi]\!]$ can then be discharged to the tamarin prover that we use as a backend.

A. Definition of the translation of processes

To model the adversary's message deduction capabilities, we introduce the set of rules MD defined in Figure 6.

In order for our translation to be correct, we need to make some assumptions on the set of processes we allow. These assumptions are however, as we will see, rather mild and most of them without loss of generality. First we define a set of reserved variables that will be used in our translation and whose use we therefore forbid in the processes.

Definition 11 (Reserved variables and facts)**.** *The set of reserved variables is defined as the set containing the elements n_a for any $a \in FN$ and $lock_l$ for any $l \in \mathbb{N}$.*

The set of reserved facts $\mathcal{F}_{res}$ is defined as the set containing facts $f(t_1,\dots,t_n)$ where $t_1,\dots,t_n \in \mathcal{T}$ and $f \in \{$ Init, Insert, Delete, IsIn, IsNotSet, state, Lock, Unlock, Out, Fr, In, Msg, ProtoNonce, Eq, NotEq, Event, InEvent $\}$.

Similar to [9], for our translation to be sound, we require that for each process, there exists an injective mapping assigning to every $\mathsf{unlock}\ t$ in a process a $\mathsf{lock}\ t$ that precedes it in the process' syntax tree. Moreover, given a process $\mathsf{lock}\ t; P$ the corresponding unlock in P may not be under a parallel or replication. These conditions allow us to annotate each corresponding pair $\mathsf{lock}\ t$, $\mathsf{unlock}\ t$ with a unique label l. The annotated version of a process P is denoted $\overline{P}$. The formal definition of $\overline{P}$ is given in Appendix A. In case the annotation fails, i.e., P violates one of the above conditions, the process $\overline{P}$ contains $\bot$.

Definition 12 (well-formed)**.** *A ground process P is well-formed if*

- *no reserved variable nor reserved fact appear in P,*
- *any name and variable in P is bound at most once and*
- *$\overline{P}$ does not contain $\bot$.*
- *For each action $l \dashv\![a]\!\mapsto r$ that appears in the process, the following holds: for each $l' \dashv\![a']\!\mapsto r' \in_E ginsts(l \dashv\![a]\!\mapsto r)$ we have that $\cap_{r''=_E r'} names(r'') \cap FN \subseteq \cap_{l''=_E l'} names(l'') \cap FN$.*

A trace formula φ is well-formed if no reserved variable nor reserved fact appear in φ.

The two first restrictions of well-formed processes are not a loss of generality as processes and formulas can be consistently renamed to avoid reserved variables and α-converted to avoid binding names or variables several times. Also note that the second condition is not necessarily preserved during an execution, e.g. when unfolding a replication, $!P$ and P may bind the same names. We only require this condition to hold on the initial process for our translation to be correct.

The annotation of locks restricts the set of protocols we can translate, but allows us to obtain better verification results, since we can predict which unlock is "supposed" to close a given lock. This additional information is helpful for tamarin's backward reasoning. We think that our locking mechanism captures all practical use cases. Using our calculus' "low-level" multiset manipulation construct, the user is also free to implement locks himself, e.g., as

$$[\mathrm{NotLocked}()] \to [];\, code;\, [] \to [\mathrm{NotLocked}()]$$

(In this case the user does not benefit from the optimisation we put into the translation of locks.) Obviously, locks can be

$$\begin{array}{rcll} \mathsf{Out}(x) & -[\,]\!\!\rightarrow & !\mathsf{K}(x) & (\text{MDO}\textsc{ut}) \\ !\mathsf{K}(x) & -[K(x)]\!\!\rightarrow & \mathsf{In}(x) & (\text{MDI}\textsc{n}) \\ & -[\,]\!\!\rightarrow & !\mathsf{K}(x:pub) & (\text{MDP}\textsc{ub}) \\ \mathsf{Fr}(x:\mathit{fresh}) & -[\,]\!\!\rightarrow & !\mathsf{K}(x:\mathit{fresh}) & (\text{MDF}\textsc{resh}) \\ !\mathsf{K}(x_1),\ldots,!\mathsf{K}(x_k) & -[\,]\!\!\rightarrow & !\mathsf{K}(f(x_1,\ldots,x_k)) \text{ for } f \in \Sigma^k & (\text{MDA}\textsc{ppl}) \end{array}$$

Figure 6. The set of rules MD.

modelled both in tamarin's multiset rewriting calculus (this is actually what the translation does) and Mödersheim's set rewriting calculus [10]. However, protocol steps typically consist of a single input, followed by several database lookups, and finally an output. In practice, they tend to be modelled as a single rule, and are therefore atomic. Real implementations are however different, as several entities might be involved, database lookups could be slow, etc. In this case, such simplified models could, e. g., miss race conditions. To the best of our knowledge, StatVerif is the only comparable tool that models locks explicitly and it has stronger restrictions.

Definition 13. *Given a well-formed ground process P we define the labelled multiset rewriting system $[\![P]\!]$ as*

$$\mathrm{MD} \cup \{\textsc{Init}\} \cup [\![\overline{P}, [], []]\!]$$

- *where the rule* $\textsc{Init}$ *is defined as*

$$\textsc{Init} : [] \;-[\mathrm{Init}()]\!\!\rightarrow [\mathsf{state}_{[]}()]$$

- $[\![P, p, \tilde{x}]\!]$ *is defined inductively for process P, position $p \in \mathbb{N}^*$ and sequence of variables $\tilde{x}$ in Figure 7.*
- *For a position p of P we define* state_p *to be persistent if $P|_p = \,!Q$ for some process Q; otherwise* state_p *is linear.*

In the definition of $[\![P, p, \tilde{x}]\!]$ we intuitively use the family of facts state_p to indicate that the process is currently at position p in its syntax tree. A fact state_p will indeed be true in an execution of these rules whenever some instance of P_p (i.e. the process defined by the subtree at position p of the syntax tree of P) is in the multiset $\mathcal{P}$ of the process configuration. The translation of the zero-process, parallel and replication operators merely use state_p-facts. For instance $[\![P \mid Q, p, \tilde{x}]\!]$ defines the rule

$$[\mathsf{state}_p(\tilde{x})] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x}), \mathsf{state}_{p\cdot 2}(\tilde{x})]$$

which intuitively states that when a process is at position p (modelled by the fact $\mathsf{state}_p(\tilde{x})$ being true) then the process is allowed to move both to P (putting $\mathsf{state}_{p\cdot 1}(\tilde{x})$ to true) and Q (putting $\mathsf{state}_{p\cdot 2}(\tilde{x})$ to true). The translation of $[\![P \mid Q, p, \tilde{x}]\!]$ also contains the set of rules $[\![P, p\cdot 1, \tilde{x}]\!] \cup [\![Q, p\cdot 2, \tilde{x}]\!]$ expressing that after this transition the process may behave as P and Q, i.e., the processes at positions $p \cdot 1$, respectively $p \cdot 2$, in the process tree. Also note that the translation of $!P$ results in a persistent fact as $!P$ always remains in $\mathcal{P}$. The translation of the construct $\nu\, a$ translates the name a into a variable n_a, as msr rules must not contain fresh names. Any instantiation of this rule will substitute n_a by a fresh name, which the Fr-fact in the premise guarantees to be new. This step is annotated with a (reserved) action *ProtoNonce*, used in the proof of correctness to distinguish adversary and protocol nonces. Note that the fact $\mathsf{state}_{p\cdot 1}$ in the conclusion carries n_a, so that the following protocol steps are bound to the fresh name used to instantiate n_a. The first rules of the translation of out and in model the communication between the protocol and the adversary, and vice versa. In the case of out, the adversary must know the channel M, modelled by the fact $\mathsf{In}(M)$ in the rule's premisse, and learns the output message, modelled by the fact $\mathsf{Out}(N)$ in the conclusion. In the case of in, the knowledge of the message N is additionally required and the variables of the input message are added to the parameters of the state fact to reflect that these variables are bound. The second and third rules of the translations of out and in model an internal communication, which is synchronous. For this reason, when the second rule of the translation of out is fired, the state-fact is substituted by an intermediate, *semi-state* fact, $\mathsf{state}^{\mathsf{semi}}$, reflecting that the sending process can only execute the next step if the message was successfully received. The fact $\mathsf{Msg}(M, N)$ models that a message is present on the synchronous channel. Only with the acknowledgement fact $\mathsf{Ack}(M, N)$, resulting from the second rule of the translation of in, is it possible to advance the execution of the sending process, using the third rule in the translation of out, which transforms the semi-state *and* the acknowledgement of receipt into $\mathsf{state}_{p\cdot 1}(\ldots)$. Only now the next step in the execution of the sending process can be executed. The remaining rules essentially update the position in the state facts and add labels. Some of these labels are used to restrict the set of executions. For instance the label $\mathrm{Eq}(M,N)$ will be used to indicate that we only consider executions in which $M =_E N$. As we will see in the next section these restrictions will be encoded in the trace formula.

Example. Figure 8 illustrates the above translation by presenting the set of msr rules $[\![!P_{new}]\!]$ (omitting the rules in MD already shown in Figure 6).

A graph representation of an example trace, generated by the tamarin tool, is depicted in Figure 9. Every box in this picture stands for the application of a multiset rewrite

$$
\begin{aligned}
\llbracket 0, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \rightarrow []\}\\
\llbracket P \mid Q, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x}), \mathsf{state}_{p\cdot 2}(\tilde{x})]\}\\
&\quad \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket \cup \llbracket Q, p\cdot 2, \tilde{x} \rrbracket\\
\llbracket !P, p, \tilde{x} \rrbracket &= \{[!\mathsf{state}_p(\tilde{x})] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket\\
\llbracket \nu a; P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x}), \mathsf{Fr}(n_a : \mathit{fresh})] \;-[\mathit{ProtoNonce}(n_a : \mathit{fresh})]\!\rightarrow\\
&\quad [\mathsf{state}_{p\cdot 1}(\tilde{x}, n_a : \mathit{fresh})]\} \cup \llbracket P, p\cdot 1, (\tilde{x}, n_a : \mathit{fresh}) \rrbracket\\
\llbracket \mathsf{Out}(M,N); P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x}), \mathsf{In}(M)] \;-[\mathrm{InEvent}(M)]\!\rightarrow [\mathsf{Out}(N), \mathsf{state}_{p\cdot 1}(\tilde{x})],\\
&\quad [\mathsf{state}_p(\tilde{x})] \rightarrow [\mathsf{Msg}(M,N), \mathsf{state}_p^{\mathsf{semi}}(\tilde{x})],\\
&\quad [\mathsf{state}_p^{\mathsf{semi}}(\tilde{x}), \mathsf{Ack}(M,N)] \rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket\\
\llbracket \mathrm{In}(M,N); P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x}), \mathsf{In}(\langle M, N\rangle)] \;-[\mathrm{InEvent}(\langle M, N\rangle)]\!\rightarrow\\
&\quad [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(N))], [\mathsf{state}_p(\tilde{x}), \mathsf{Msg}(M,N)] \rightarrow\\
&\quad [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(N)), \mathsf{Ack}(M,N)]\}\\
&\quad \cup \llbracket P, p\cdot 1, \tilde{x} \cup \mathit{vars}(N) \rrbracket\\
\llbracket \text{if } M = N \text{ then } P \text{ else } Q, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \;-[\mathrm{Eq}(M,N)]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})],\\
&\quad [\mathsf{state}_p(\tilde{x})] \;-[\mathrm{NotEq}(M,N)]\!\rightarrow [\mathsf{state}_{p\cdot 2}(\tilde{x})]\}\\
&\quad \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket \cup \llbracket Q, p\cdot 2, \tilde{x} \rrbracket\\
\llbracket \text{event } F; P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \;-[\mathrm{Event}(), F]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket\\
\llbracket \text{insert } s, t; P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \;-[\mathrm{Insert}(s,t)]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket\\
\llbracket \text{delete } s; P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \;-[\mathrm{Delete}(s)]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket\\
\llbracket \text{lookup } M \text{ as } v \text{ in } P \text{ else } Q, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \;-[\mathrm{IsIn}(M,v)]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{M}, v)],\\
&\quad [\mathsf{state}_p(\tilde{x})] \;-[\mathrm{IsNotSet}(M)]\!\rightarrow [\mathsf{state}_{p\cdot 2}(\tilde{x})]\}\\
&\quad \cup \llbracket P, p\cdot 1, (\tilde{x}, v) \rrbracket \cup \llbracket Q, p\cdot 2, \tilde{x} \rrbracket\\
\llbracket \text{lock}^l\ s; P, p, \tilde{x} \rrbracket &= \{[\mathrm{Fr}(\mathrm{lock}_l), \mathsf{state}_p(\tilde{x})] \;-[\mathrm{Lock}(\mathit{lock}_l, s)]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x}, \mathit{lock}_l)]\}\\
&\quad \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket\\
\llbracket \text{unlock}^l\ s; P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x})] \;-[\mathrm{Unlock}(\mathit{lock}_l, s)]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x})]\} \cup \llbracket P, p\cdot 1, \tilde{x} \rrbracket\\
\llbracket [l \;-[a]\!\rightarrow r]; P, p, \tilde{x} \rrbracket &= \{[\mathsf{state}_p(\tilde{x}), l] \;-[\mathrm{Event}(), a]\!\rightarrow [r, \mathsf{state}_{p\cdot 1}(\tilde{x} \cup \mathit{vars}(l))]\}\\
&\quad \cup \llbracket P, p\cdot 1, \tilde{x} \cup \mathit{vars}(l) \rrbracket
\end{aligned}
$$

Figure 7. Translation of processes: definition of $\llbracket P, p, \tilde{x} \rrbracket$

rule, where the premises are at the top, the conclusions at the bottom, and the actions (if any) in the middle. Every premise needs to have a matching conclusion, visualized by the arrows, to ensure the graph depicts a valid msr execution. (This is a simplification of the dependency graph representation tamarin uses to perform backward-induction [18], [19].) Note that the machine notation for $\mathsf{state}_p()$ predicates omits brackets [ ] in the position p and denotes the empty sequence by '0'. We also note that in the current example $!\mathsf{state}_{[1]}()$ is persistent and can therefore be used multiple times as a premise. As $\mathsf{Fr}(\,)$ facts are generated by the FRESH rule which has an empty premise and action, we omit instances of FRESH and leave those premises, but only those, disconnected.

Remark 1. *One may note that, while for all other operators, the translation produces well-formed multiset rewriting rules (as long as the process is well-formed itself), this is not the case for the translation of the* lookup *operator, i. e., it violates the well-formedness condition from Definition 4. Tamarin's constraint solving algorithm requires all rules, with the exception of* FRESH*, to be well-formed. We show however that, under these specific conditions, the solution procedure is still correct. See Appendix C in the full version for the proof.*

B. Definition of the translation of trace formulas

We can now define the translation for formulas.

Definition 14. *Given a well-formed trace formula φ we*

$$
\begin{array}{rcl}
[\,] & -[\mathrm{Init}()]\rightarrow & [\mathsf{state}_{[]}()] \\
[\mathsf{state}_{[]}()] & -[\;]\rightarrow & [!\mathsf{state}_{[1]}()] \\
[!\mathsf{state}_{[1]}(), \mathsf{Fr}(h)] & -[\;]\rightarrow & [\mathsf{state}_{[11]}(h)] \\
[\mathsf{state}_{[11]}(h), \mathsf{Fr}(k)] & -[\;]\rightarrow & [\mathsf{state}_{[111]}(k,h)] \\
[\mathsf{state}_{[111]}(k,h)] & -[\mathrm{Event}(), \mathrm{NewKey}(h,k)]\rightarrow & [\mathsf{state}_{[1111]}(k,h)] \\
[\mathsf{state}_{[1111]}(k,h)] & -[\mathrm{Insert}(\langle\text{'key'}, h\rangle, k)]\rightarrow & [\mathsf{state}_{[11111]}(k,h)] \\
[\mathsf{state}_{[11111]}(k,h)] & -[\mathrm{Insert}(\langle\text{'att'}, h\rangle, \text{'dec'})]\rightarrow & [\mathsf{state}_{[111111]}(k,h)] \\
[\mathsf{state}_{[111111]}(k,h)] & -[\;]\rightarrow & [\mathsf{Out}(h), \mathsf{state}_{[1111111]}(k,h)]
\end{array}
$$

Figure 8. The set of multiset rewrite rules $[\![!P_{new}]\!]$ (omitting the rules in MD)

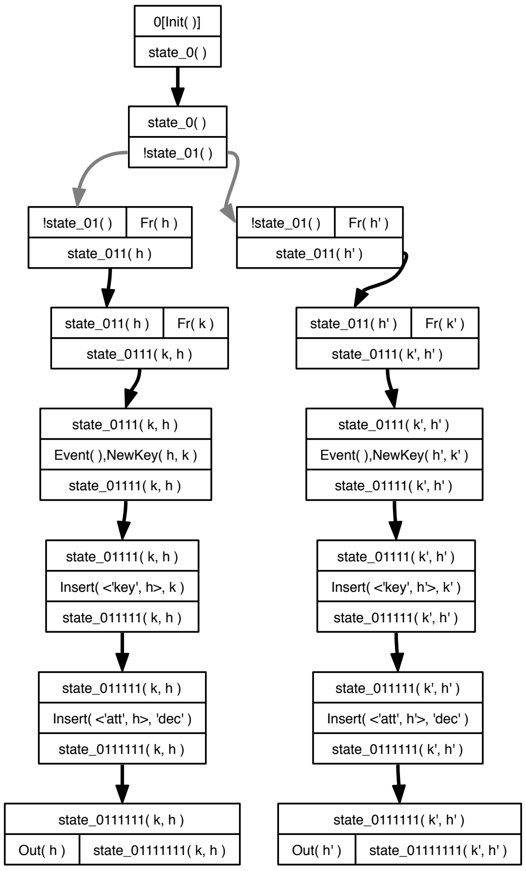

Figure 9. Example trace for the translation of $!P_{new}$.

define

$$[\![\varphi]\!]_\forall := \alpha \Rightarrow \varphi \qquad \textit{and} \qquad [\![\varphi]\!]_\exists := \alpha \wedge \varphi$$

where α is defined in Figure 10.

The formula α uses the actions of the generated rules to filter out executions that we wish to discard:

- α_{init} ensures that the init rule is only fired once.
- α_{eq} and α_{noteq} ensure that we only consider traces where all (dis)equalities hold.
- α_{in} and α_{notin} ensure that a successful lookup was preceded by an insert that was neither revoked nor overwritten while an unsuccessful lookup was either never inserted, or deleted and never re-inserted.
- α_{lock} checks that between each two matching locks there must be an unlock. Furthermore, between the first of these locks and the corresponding unlock, there is neither a lock nor an unlock.
- α_{inev} ensures that whenever an instance of MDIN is required to generate an In-fact, it is generated as late as possible, i. e., there is no visible event between the action $K(t)$ produced by MDIN, and a rule that requires $\mathsf{In}(t)$.

We also note that $Tr \vDash^\forall [\![\varphi]\!]_\forall$ iff $Tr \nvDash^\exists [\![\neg\varphi]\!]_\exists$.

The axioms in the translation of the formula are designed to work hand in hand with the translation of the process into rules. They express the correctness of traces with respect to our calculus' semantics, but are also meant to guide tamarin's constraint solving algorithm. α_{in} and α_{notin} illustrate what kind of axioms work well: when a node with the action IsIn is created, by definition of the translation, this corresponds to a lookup command. The existential translates into a graph constraint that postulates an insert node for the value fetched by the lookup, and three formulas assuring that *a*) this insert node appears before the lookup, *b*) is uniquely defined, i. e., it is the last insert to the corresponding key, and *c*) there is no delete in between. Due to these conditions, α_{notin} only adds one Insert node per IsIn node – the case where an axiom postulates a node, which itself allows for postulating yet another node needs to be avoided, as tamarin runs into loops otherwise. Similarly, a naïve way of implementing locks using an axiom would postulate that every lock is preceeded by an unlock and no lock or unlock in between, unless it is the first lock. This again would cause tamarin to loop, because an unlock is typically preceeded by yet another lock. The axiom α_{lock} avoids this caveat because it only applies to pairs of locks carrying the same annotations.

The interaction between the α_{lock} axiom and tamarin's constraint solving algorithm is described in more detail in the full version.

C. Correctness of the translation

The correctness of our translation is stated by the following theorem.

$\alpha := \alpha_{init} \wedge \alpha_{eq} \wedge \alpha_{noteq} \wedge \alpha_{in} \wedge \alpha_{notin} \wedge \alpha_{lock} \wedge \alpha_{inev}$ and

$$
\begin{aligned}
\alpha_{init} &:= \forall i, j. && \text{Init}()@i \wedge \text{Init}()@j \implies i = j \\
\alpha_{eq} &:= \forall x, y, i. && \text{Eq}(x, y)@i \implies x \approx y \\
\alpha_{noteq} &:= \forall x, y, i. && \text{NotEq}(x, y)@i \implies \neg(x \approx y) \\
\alpha_{in} &:= \forall x, y, t_3. && \text{IsIn}(x, y)@t_3 \implies \exists t_2.\ \text{Insert}(x, y)@t_2 \wedge t_2 \lessdot t_3 \\
& && \qquad \wedge \forall t_1, y.\ \text{Insert}(x, y)@t_1 \implies (t_1 \lessdot t_2 \vee t_1 \doteq t_2 \vee t_3 \lessdot t_1) \\
& && \qquad \wedge \forall t_1. \quad \text{Delete}(x)@t_1 \implies (t_1 \lessdot t_2 \vee t_3 \lessdot t_1) \\
\alpha_{notin} &:= \forall x, y, t_3. && \text{IsNotSet}(x)@t_3 \implies (\forall t_1, y.\ \text{Insert}(x, y)@t_1 \implies t_3 \lessdot t_1) \vee \\
& && \qquad (\exists t_1.\ \text{Delete}(x)@t_1 \wedge t_1 \lessdot t_3 \\
& && \qquad\quad \wedge \forall t_2, y.\ (\text{Insert}(x, y)@t_2 \wedge t_2 \lessdot t_3) \implies t_2 \lessdot t_1) \\
\alpha_{lock} &:= \forall x, l, l', i, j. && \text{Lock}(l, x)@i \wedge \text{Lock}(l', x)@j \wedge i \lessdot j \\
& && \quad \implies \exists k.\ \text{Unlock}(l, x)@k \wedge i \lessdot k \wedge k \lessdot j \\
& && \qquad \wedge (\forall l', m.\ \text{Lock}(l', x)@m \implies \neg(i \lessdot m \wedge m \lessdot k)) \\
& && \qquad \wedge (\forall l', m.\ \text{Unlock}(l', x)@m \implies \neg(i \lessdot m \wedge m \lessdot k)) \\
\alpha_{inev} &:= \forall t, i. && \text{InEvent}(t)@i \implies \exists j.\ \text{K}(t)@j \wedge (\forall k.\ \text{Event}()@k \implies (k \lessdot j \vee i \lessdot k)) \\
& && \qquad \wedge (\forall k, t'.\ \text{K}(t')@k \implies (k \lessdot j \vee i \lessdot k \vee k \approx j))
\end{aligned}
$$

Figure 10. Definition of α.

Theorem 1. *Given a well-formed ground process P and a well-formed trace formula φ we have that*

$$\mathit{traces}^{pi}(P) \vDash^{\star} \varphi \text{ iff } \mathit{traces}^{msr}([\![P]\!]) \vDash^{\star} [\![\varphi]\!]_{\star}$$

where $\star$ is either $\forall$ or $\exists$.

We here give an overview of the main propositions and lemmas needed to prove Theorem 1. To show the result we need two additional definitions. We first define an operation that allows to restrict a set of traces to those that satisfy the trace formula α as defined in Definition 14.

Definition 15. *Let α be the trace formula as defined in Definition 14 and Tr a set of traces. We define*

$$\mathit{filter}(Tr) := \{tr \in Tr \mid \forall \theta.(tr, \theta) \vDash \alpha\}$$

The following proposition states that if a set of traces satisfies the translated formula then the filtered traces satisfy the original formula.

Proposition 1. *Let Tr be a set of traces and φ a trace formula. We have that*

$$Tr \vDash^{\star} [\![\varphi]\!]_{\star} \text{ iff } \mathit{filter}(Tr) \vDash^{\star} \varphi$$

where $\star$ is either $\forall$ or $\exists$.

The proof (detailed in the full version) follows directly from the definitions. Next we define the *hiding* operation which removes all reserved facts from a trace.

Definition 16 (*hide*)**.** *Given a trace tr and a set of facts F we inductively define $\mathit{hide}([]) = []$ and*

$$\mathit{hide}(F \cdot tr) := \begin{cases} \mathit{hide}(tr) & \text{if } F \subseteq \mathcal{F}_{res} \\ (F \setminus \mathcal{F}_{res}) \cdot \mathit{hide}(tr) & \text{otherwise} \end{cases}$$

Given a set of traces Tr we define $\mathit{hide}(Tr) = \{\mathit{hide}(t) \mid t \in Tr\}$.

As expected well-formed formulas that do not contain reserved facts evaluate the same whether reserved facts are hidden or not.

Proposition 2. *Let Tr be a set of traces and φ a well-formed trace formula. We have that*

$$Tr \vDash^{\star} \varphi \text{ iff } \mathit{hide}(Tr) \vDash^{\star} \varphi$$

where $\star$ is either $\forall$ or $\exists$.

We can now state our main lemma which is relating the set of traces of a process P and the set of traces of its translation into multiset rewrite rules (proven in the full version).

Lemma 1. *Let P be a well-formed ground process. We have that*

$$\mathit{traces}^{pi}(P) = \mathit{hide}(\mathit{filter}(\mathit{traces}^{msr}([\![P]\!]))).$$

Our main theorem can now be proven by applying Lemma 1, Proposition 2 and Proposition 1.

Proof of Theorem 1:

$$\begin{array}{lr} \mathit{traces}^{pi}(P) \models^\star \varphi & \\ \Leftrightarrow \mathit{hide}(\mathit{filter}(\mathit{traces}^{msr}(\llbracket P \rrbracket))) \models^\star \varphi & \text{by Lemma 1} \\ \Leftrightarrow \mathit{filter}(\mathit{traces}^{msr}(\llbracket P \rrbracket)) \models^\star \varphi & \text{by Proposition 2} \\ \Leftrightarrow \mathit{traces}^{msr}(\llbracket P \rrbracket) \models^\star \llbracket \varphi \rrbracket_\star & \text{by Proposition 1} \end{array}$$

■

VII. Case studies

In this section we briefly overview some case studies we performed. These case studies include a simple security API similar to PKCS#11 [12], the Yubikey security token, the optimistic contract signing protocol by Garay, Jakobsson and MacKenzie (GJM) [24] and a few other examples discussed in Arapinis et al. [9] and Mödersheim [10]. The results are summarized in Figure 11. For each case study we provide the number of typing lemmas that were needed by the tamarin prover and whether manual guidance of the tool was required. In case no manual guidance is required we also give execution times. We do not detail all the formal models of the protocols and properties that we studied, and sometimes present slightly simplified versions. All files of our prototype implementation and our case studies are available at http://sapic.gforge.inria.fr/.

Example	Typing Lemmas	Automated Run*
Security API à la PKCS#11	1	yes ($51s$)
Yubikey Protocol [23], [25]	3	no
GJM protocol [9], [24]	0	yes ($36s$)
Mödersheim's example (locks/inserts) [10]	0	no**
Mödersheim's example (embedded msr rules) [10]	0	yes ($1s$)
Security Device [9]	1	yes ($21s$)
Needham-Schroeder-Lowe [3]	1	yes ($5s$)

* (Running times on Intel Core2 Duo 2.66Ghz with 4GB RAM)
** (little interaction: 7 manual rule selections)

Figure 11. Case studies.

A. Security API à la PKCS#11

This example illustrates how our modelling might be useful for the analysis of Security APIs in the style of the PKCS#11 standard [12]. We expect studying a complete model of PKCS#11, such as in [20], to be a straightforward extension of this example. In addition to the processes presented in the running example in Section III the actual case study models the following two operations: *(i) encryption:* given a handle and a plain-text, the user can request an encryption under the key the handle points to. *(ii) unwrap* given a ciphertext $senc(k_2, k_1)$, and a handle h_1, the user can request the ciphertext to be *unwrapped*, i.e. decrypted, under the key pointed to by h_1. If decryption is successful the result is stored on the device, and a handle pointing to k_2 is returned. Moreover, contrary to the running example, at creation time keys are assigned the attribute 'init', from which they can move to either 'wrap', or 'unwrap', see the following snippet:

```
1 in($\langle$'set_dec',$h\rangle$); lock $\langle$'att',$h\rangle$;
2   lookup $\langle$'att',$h\rangle$ as $a$ in
3     if $a$='init' then
4       insert $\langle$'att',$h\rangle$, 'dec'; unlock $\langle$'att',$h\rangle$
```

Note that, in contrast to the running example, it is necessary to encapsulate the state changes between lock and unlock. Otherwise an adversary can stop the execution after line 3, set the attribute to 'wrap' in a concurrent process and produce a wrapping. After resuming operation at line 4, he can set the key's attribute to 'dec', even though the attribute is set to 'wrap'. Hence, the attacker is allowed to decrypt the wrapping he has produced and can obtain the key. Such subtleties can produce attacks that our modeling allows to detect. If locking is handled correctly, we show secrecy of keys produced on the device, proving the property introduced in Example 5. If locks are removed the attack described before is found.

B. Yubikey

The Yubikey [25] is a small hardware device designed to authenticate a user against network-based services. Manufactured by Yubico, a Swedish company, the Yubikey itself is a low cost ($25), thumb-sized USB device. In its typical configuration, it generates one-time passwords based on encryptions of a secret value, a running counter and some random values using a unique AES-128 key contained in the device. The Yubikey authentication server accepts a one-time password only if it decrypts under the correct AES key to a valid secret value containing a counter larger than the last counter accepted. The counter is thus used as a means to prevent replay attacks. To date, over a million Yubikeys have been shipped to more than 30,000 customers including governments, universities and enterprises, e.g. Google, Microsoft, Agfa and Symantec [26].

Besides the counter values used in the one-time password, the Yubikey stores three additional pieces of information: the public id pid that is used to identify the Yubikey, a secret id $secretid$ that is transmitted as part of the one-time password and only known to the server and the Yubikey, as well as the AES key k, which is also shared with the server. The following process $P_{Yubikey}$ models a single Yubikey, as well as its initial configuration, where an entry in the server's database for the public id pid is created. This entry contains a tuple consisting of the Yubikey's secret id, AES key, and an initial counter value.

$P_{Yubikey} =$
$\nu\ k;\ \nu\ pid;\ \nu\ \ secretid\ ;$

```
insert ⟨'Server',$pid$⟩, ⟨ $secretid$ , $k$, 'zero'⟩;
insert ⟨'Yubikey', $pid$⟩, 'zero'+'one';
out($pid$);
$!P_{Plugin}$ | $!P_{ButtonPress}$
```

Here, the processes $!P_{Plugin}$ and $!P_{ButtonPress}$ model the Yubikey being unplugged and plugged in again (possibly on a different computer), and the emission of the one-time password. We will only discuss $P_{ButtonPress}$ here. When the user presses the button on the Yubikey, the device outputs a one-time password consisting of a counter tc, the secret id $secretid$ and additional randomness npr encrypted using the AES key k.

```
$P_{ButtonPress}$ =
lock $pid$;
  lookup ⟨'Yubikey',$pid$⟩ as $tc$ in
    insert ⟨'Yubikey',$pid$⟩, $tc$ + 'one';
    $\nu$ $nonce$; $\nu$ $npr$;
    event YubiPress($pid$, $secretid$ ,$k$, $tc$ );
    out(⟨$pid$,$nonce$,$senc$(⟨ $secretid$ , $tc$ ,$npr$⟩,$k$)⟩);
unlock $pid$
```

The one-time password $senc(\langle secretid, tc, npr\rangle, k)$ can be used to authenticate against a server that shares the same secret key, which we model in the process P_{Server}. The process receives the encrypted one-time password along with the public id pid of a Yubikey and a $nonce$ that is part of the protocol, but is irrelevant for the authentication of the Yubikey on the server.

The server looks up the secret id and the AES key associated to the public id, i. e., to the Yubikey sending the request, as well as the last recorded counter value otc. If the key and secret id used in the request match the values retrieved from the database, then the event $\mathrm{Smaller}(otc, tc)$ is logged along with the event $\mathrm{Login}(pid, k, tc)$, which marks a successful login of the Yubikey pid with key k for the counter value tc. Afterwards, the old tuple $\langle secretid, k, otc\rangle$ is replaced by $\langle secretid, k, tc\rangle$, to update the latest counter value received.

```
$P_{Server}$ =
! in(⟨$pid$,$nonce$,$senc$(⟨ $secretid$ , $tc$ ,$npr$⟩,$k$)⟩);
lock $pid$;
lookup ⟨'Server',$pid$⟩ as $tuple$ in
  if $fst$ ( $tuple$ )=$secretid$ then
    if $fst$ ($snd$( $tuple$ ))=$k$ then
      event Smaller($snd$($snd$( $tuple$ )), $tc$ )
      event Login($pid$,$k$, $tc$ );
      insert ⟨'Server',$pid$⟩, ⟨ $secretid$ ,$k$, $tc$⟩;
unlock $pid$
```

Note that, in our modelling, the server keeps one lock per public id, which means that it is possible to have several active instances of the server thread in parallel as long as all requests concern different Yubikeys.

An important part of the modelling of the protocol is to determine whether one counter value is smaller than another. To this end, our modelling employs a feature added to the development version of tamarin as of October 2012, a union operator $\cup^{\#}$ for multisets of message terms. The operator is denoted with a plus sign ("+"). We model the counter as a multiset only consisting of the symbols "one" and "zero". The multiplicity of 'one' in the multiset is the value of the counter. A counter value is considered smaller than another one, if the first multiset is included in the second. A test $a < b$ is included by adding the event $\mathrm{Smaller}(a, b)$ and an axiom that requires that a is a subset of b:

$$\alpha_{Smaller} := \forall i : temp, a, b : msg.\ \mathrm{Smaller}(a, b)@i \\ \Rightarrow \exists z : msg.\ a + z = b$$

We incorporate this axiom into the security properties just like in Definition 14. Intuitively, we are only interested in traces where a is indeed smaller than b.

The process we analyse models a single authentication server (that may run arbitrary many threads) and an arbitrary number of Yubikeys, i. e., $P_{Server} \mid !P_{Yubikey}$. Among other properties, we show by the means of an injective correspondence property that an attacker that controls the network cannot perform replay attacks, and that each successful login was preceded by a user "pressing the button", formally:

$$\forall\, pid, k, x, t_2.\mathrm{Login}(pid, k, x)@t_2 \Rightarrow \\ \exists sid, t_1.\mathrm{YubiPress}(pid, sid, k, x)@t_1 \wedge t_1 \lessdot t_2 \\ \wedge \forall t_3.\mathrm{Login}(pid, k, x)@t_3 \Rightarrow t_3 = t_2$$

Besides injective correspondence, we show the absence of replay attacks and the property that a successful login invalidates previously emitted one-time passwords. All three properties follow more or less directly from a stronger invariant, which itself can be proven in 295 steps. To find theses steps, tamarin needs some additional human guidance, which can be provided using the interactive mode. This mode still allows the user to complement his manual efforts with automated backward search. The example files contain the modelling in our calculus, the complete proof, and the manual part of the proof which can be verified by tamarin without interaction.

Our analysis makes three simplifications: First, in P_{Server}, we use pattern matching instead of decryption as demonstrated in the process P_{dec} we introduced in Section III. Second, we omit the CRC checksum and the time-stamp that are part of the one-time password in the actual protocol, since they do not add to the security of the protocol in the symbolic setting. Third, the Yubikey has actually two counters instead of one, a session counter, and a token counter. We treat the session and token counter on the Yubikey as a single value, which we justify by the fact that the Yubikey either increases the session counter and resets the token counter, or increases only the token counter, thereby implementing a complete lexicographical order on the pair $(session\ counter, token\ counter)$.

A similar analysis has already been performed by Künnemann and Steel, using tamarin's multiset rewriting calculus [23]. However, the model in our new calculus is more fine-grained and we believe more readable. Security-relevant operations like locking and tests on state are written out in detail, resulting in a model that is closer to the real-life operation of such a device. The modeling of the Yubikey takes approximately 38 lines in our calculus, which translates to 49 multiset rewrite rules. The model of [23] contains only four rules, but they are quite complicated, resulting in 23 lines of code. More importantly, the gap between their model and the actual Yubikey protocol is larger – in our calculus, it becomes clear that the server can treat multiple authentication requests in parallel, as long as they do not claim to stem from the same Yubikey. An implementation on the basis of the model from Künnemann and Steel would need to implement a global lock accessible to the authentication server and all Yubikeys. This is however unrealistic, since the Yubikeys may be used at different places around the world, making it unlikely that there exist means of direct communication between them. While a server-side global lock might be conceivable (albeit impractical for performance reasons), a real global lock could not be implemented for the Yubikey as deployed.

C. Further Case Studies

We also investigated the case study presented by Mödersheim [10], a key-server example. We encoded two models of this example, one using the insert construct, the other manipulating state using the embedded multiset rewrite rules. For this example the second model turned out to be more natural and more convenient allowing for a direct automated proof without any additional typing lemma.

We furthermore modeled the contract signing protocol by Garay et al. [24] and a simple security device which both served as examples in [9]. In the contract signing protocol a trusted party needs to maintain a database with the current status of all contracts (aborted, resolved, or no decision has been taken). In our calculus the status information is naturally modelled using our insert and lookup constructs. The use of locks is indispensable to avoid the status to be changed between a lookup and an insert. Arapinis et al. [9] showed the crucial property that the same contract can never be both aborted and resolved. However, due to the fact that StatVerif only allows for a finite number of memory cells, they have shown this property for a single contract and provide a manual proof to lift the result to an unbounded number of contracts. We directly prove this property for an unbounded number of contracts. Finally we also illustrate the tool's ability to analyze classical security protocols, by analyzing the Needham Schroeder Lowe protocol [3].

VIII. Conclusion

We present a process calculus which extends the applied pi calculus with constructs for accessing a global, shared memory together with an encoding of this calculus in labelled msr rules which enables automated verification using the tamarin prover as a backend. Our prototype verification tool, automating this translation, has been successfully used to analyze several case studies. As future work we plan to increase the degree of automation of the tool by automatically generating helping lemmas. To achieve this goal we can exploit the fact that we generate the msr rules, and hence control their form. We also plan to use the tool for more complex case studies including a complete model of PKCS#11 and a study of the TPM 2.0 standard, currently in public review. Finally, we wish to investigate how our constructs for manipulating state can be used to encode loops, needed to model stream protocols such as TESLA.

Acknowledgements: The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 258865, project ProSecure and was supported by CASED (http://www.cased.de).

References

[1] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. T. Abad, "Formal analysis of saml 2.0 web browser single sign-on: Breaking the saml-based single sign-on for google apps," in *Proc. 6th ACM Workshop on Formal Methods in Security Engineering (FMSE'08)*, 2008, pp. 1–10.

[2] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel, "Attacking and fixing PKCS#11 security tokens," in *Proc. 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM Press, 2010, pp. 260–269.

[3] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in *Proc. 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96)*, ser. LNCS, vol. 1055. Springer, 1996, pp. 147–166.

[4] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *Proc. 14th Computer Security Foundations Workshop (CSFW'01)*. IEEE Press, 2001, pp. 82–96.

[5] A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The AVISPA tool for the automated validation of internet security protocols and applications." in *Proc. 17th International Conference on Computer Aided Verification (CAV'05)*, ser. LNCS. Springer, 2005, pp. 281–285.

[6] S. Escobar, C. Meadows, and J. Meseguer, "Maude-npa: Cryptographic protocol analysis modulo equational properties," in *Foundations of Security Analysis and Design V*, ser. LNCS, vol. 5705. Springer, 2009, pp. 1–50.

[7] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. 28th ACM Symp. on Principles of Programming Languages (POPL'01)*. ACM Press, 2001, pp. 104–115.

[8] B. Blanchet, B. Smyth, and V. Cheval, *ProVerif 1.88: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2013.

[9] M. Arapinis, E. Ritter, and M. Ryan, "Statverif: Verification of stateful processes." in *Proc. 24th IEEE Computer Security Foundations Symposium (CSF'11)*. IEEE Press, 2011, pp. 33–47.

[10] S. Mödersheim, "Abstraction by set-membership: verifying security protocols and web services with databases," in *Proc. 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, 2010, pp. 351–360.

[11] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, "Formal analysis of protocols based on TPM state registers," in *Proc. 24th IEEE Computer Security Foundations Symposium (CSF'11)*. IEEE Press, 2011, pp. 66–82.

[12] *PKCS #11: Cryptographic Token Interface Standard.*, RSA Security Inc., v2.20, June 2004.

[13] *CCA Basic Services Reference and Guide*, Oct. 2006, available online.

[14] Trusted Computing Group, "TPM Specification version 1.2. Parts 1–3, revision 103," http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2007.

[15] D. Longley and S. Rigby, "An automatic search for security flaws in key management schemes," *Computers and Security*, vol. 11, no. 1, pp. 75–89, March 1992.

[16] M. Bond and R. Anderson, "API level attacks on embedded systems," *IEEE Computer Magazine*, pp. 67–75, October 2001.

[17] J. Herzog, "Applying protocol analysis to security device interfaces," *IEEE Security & Privacy Magazine*, vol. 4, no. 4, pp. 84–87, July-Aug 2006.

[18] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *Proc. 25th IEEE Computer Security Foundations Symposium (CSF'12)*. IEEE Press, 2012, pp. 78–94.

[19] ——, "The tamarin prover for the symbolic analysis of security protocols," in *Proc. 25th International Conference on Computer Aided Verification (CAV'13)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 696–701.

[20] S. Delaune, S. Kremer, and G. Steel, "Formal analysis of PKCS#11 and proprietary extensions," *Journal of Computer Security*, vol. 18, no. 6, pp. 1211–1245, Nov. 2010. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/DKS-jcs09.pdf

[21] J. D. Guttman, "State and progress in strand spaces: Proving fair exchange," *J. Autom. Reasoning*, vol. 48, no. 2, pp. 159–195, 2012.

[22] S. Bistarelli, I. Cervesato, G. Lenzini, and F. Martinelli, "Relating multiset rewriting and process algebras for security protocol analysis," *Journal of Computer Security*, vol. 13, no. 1, pp. 3–47, 2005.

[23] R. Künnemann and G. Steel, "YubiSecure? Formal security analysis results for the Yubikey and YubiHSM," in *Proc. 8th Workshop on Security and Trust Management (STM'12)*, ser. LNCS, vol. 7783, 2012, pp. 257–272.

[24] J. A. Garay, M. Jakobsson, and P. D. MacKenzie, "Abuse-free optimistic contract signing," in *Advances in Cryptology—Crypto'99*, ser. LNCS, vol. 1666. Springer, 1999, pp. 449–466.

[25] *The YubiKey Manual - Usage, configuration and introduction of basic concepts (Version 2.2), available at: http://www.yubico.com/documentation*, Yubico AB, Kungsgatan 37, 111 56 Stockholm Sweden, June 2010.

[26] Yubico AB, "Yubico customer list," 2013, accessed: Wed 17 Jul 2013 11:40:50 CEST. [Online]. Available: https://www.yubico.com/about/reference-customers/

Appendix

Definition 17 (Process annotation)**.** *Given a ground process P we define the annotated ground process $\overline{P}$ as follows:*

$$\begin{aligned}
\overline{0} &:= 0\\
\overline{P|Q} &:= \overline{P}|\overline{Q}\\
\overline{!P} &:= !\overline{P}\\
\overline{\begin{array}{r}\textit{if } t_1 = t_2 \textit{ then } P\\ \textit{else } Q\end{array}} &:= \textit{if } t_1 = t_2 \textit{ then } \overline{P} \textit{ else } \overline{Q}\\
\overline{\begin{array}{r}\textit{lookup } M \textit{ as } x\\ \textit{in } P \textit{ else } Q\end{array}} &:= \begin{array}{l}\textit{lookup } M \textit{ as } x\\ \textit{in } \overline{P} \textit{ else } \overline{Q}\end{array}\\
\overline{\alpha; P} &:= \alpha; \overline{P}\\
&\quad \textit{where } \alpha \notin \{\, \mathrm{lock}\ t, \mathrm{unlock}\ t \colon t \in \mathcal{T}\,\}\\
\overline{\mathrm{lock}\ t; P} &:= \mathrm{lock}^l\ t; \overline{au(P,t,l)}\\
&\quad \textit{where } l \in \mathbb{N} \textit{ is a fresh label}\\
\overline{\mathrm{unlock}^l\ t; P} &:= \mathrm{unlock}^l\ t; \overline{P}\\
\overline{\mathrm{unlock}\ t; P} &:= \bot
\end{aligned}$$

where $au(P,t,l)$ annotates the first unlock that has parameter t with the label l, i. e.:

$$\begin{aligned}
au(P|Q,t,l) &:= \bot\\
au(!P,t,l) &:= \bot\\
\begin{array}{r}au(\textit{if } t_1 = t_2 \textit{ then}\\ P \textit{ else } Q,t,l)\end{array} &:= \begin{array}{l}\textit{if } t_1 = t_2 \textit{ then } au(P,t,l)\\ \textit{else } au(Q,t,l)\end{array}\\
\begin{array}{r}au(\textit{lookup } M \textit{ as } x\\ \textit{in } P \textit{ else } Q,t,l)\end{array} &:= \begin{array}{l}\textit{lookup } M \textit{ as } x \textit{ in}\\ au(P,t,l) \textit{ else } au(Q,t,l)\end{array}\\
au(\alpha; P,t,l) &:= \alpha; au(P,t,l)\\
&\qquad\qquad \textit{where } \alpha \neq \textit{unlock } t\\
au(\mathrm{unlock}\ t; P,t,l) &:= \mathrm{unlock}^l\ t; P\\
au(0,t,l) &:= 0
\end{aligned}$$