Symbolic Models for Isolated Execution Environments

Charlie Jacomme, Steve Kremer, Guillaume Scerri

Symbolic Models for Isolated Execution Environments

Charlie Jacomme
ENS Cachan, Université Paris-Saclay

Steve Kremer
LORIA, Inria Nancy-Grand Est & CNRS & Université de Lorraine

Guillaume Scerri
DAVID Lab, Université Versailles Saint-Quentin & Université Paris Saclay & INRIA Saclay

***Abstract*—Isolated Execution Environments (IEEs), such as ARM TrustZone and Intel SGX, offer the possibility to execute sensitive code in isolation from other malicious programs, running on the same machine, or a potentially corrupted OS. A key feature of IEEs is the ability to produce reports binding cryptographically a message to the program that produced it, typically ensuring that this message is the result of the given program running on an IEE. We present a symbolic model for specifying and verifying applications that make use of such features. For this we introduce the SℓAPIC process calculus, that allows to reason about reports issued at given locations. We also provide tool support, extending the SAPIC/TAMARIN toolchain and demonstrate the applicability of our framework on several examples implementing secure outsourced computation (SOC), a secure licensing protocol and a one-time password protocol that all rely on such IEEs.**

1. Introduction

Isolated Execution Environments (IEE) like the ones offered by Intel's Software Guard Extensions (SGX) [1] and ARM Trustzone [2] are designed to allow the execution of a program P on an untrusted machine in an isolated manner. In particular it ensures isolation of a sensitive program from a corrupted OS or other malicious programs running on the same machine. One key component of IEEs is the ability to produce reports binding cryptographically a message to the program that produced it. This capability is intended to provide guarantees on the results produced by a program running on a remote untrusted machine.

An IEE can be viewed as a RAM machine that runs a program P and exposes only a limited interface which allows to provide inputs to and retrieve outputs from P. The isolation guarantee provided by the IEE ensures that the behaviour of the process running in the IEE is only determined by the inputs/outputs communicated through its interfaces (and the semantics of P). This ensures a strict isolation both between different IEEs and between IEEs and the untrusted environment. Additionally IEEs provide (to the program running inside) system calls to a security module executing cryptographic operations. The result of such calls may depend on the content of the calling IEE. A key feature of this security module is a call that when queried with input m returns a report $\mathsf{report}(m, P)$ on both the code of the program P and the message m. This report can then be checked by agents who wish to ascertain that m was indeed produced by an IEE running P. Such reports are typically implemented as digital signatures with a key belonging to the platform providing the IEE and certified by the manufacturer[1]. We will not concentrate on the specific details of the implementation but assume the existence of such a reporting function and look at how applications can make use of it to provide security properties.

As a concrete use case of an IEE based application consider a simplification of the attested computation scheme (and associated security property) proposed by Barbosa et al. [3]. This example will serve as our running example throughout the paper. Intuitively, the scheme provides a means for a user to execute a program remotely and to ensure that the local view of the trace corresponds to a valid execution of the program in a remote IEE even when the adversary controls the network and the remote machine. In the framework proposed in [3], when an agent wants to run P remotely, he compiles P in an instrumented program $P^* = \mathsf{Compile}\ P$. We describe a simplified version of the protocol from [3] in Figure 1. P^* is executed in the remote IEE. When queried an input i, P^* computes the corresponding output o by applying P to i. P^* also keeps track of a list lio of inputs/outputs it has seen so far in its internal state st, and appends the input and the output to lio. Finally, it returns the output together with a report on lio. Locally, the user executes a local verification algorithm Verify to check that the returned results correspond to the correct computation on the provided inputs. For this the user stores the list of queried inputs and received outputs in its local state st_l and checks that the report r provided by the remote IEE was computed on the same list to ensure that the remote and local state are consistent. The expected security goal is to ensure that an adversary cannot have the user accept a sequence of inputs/outputs that were not produced in an IEE running program P^*.

Work was carried while Scerri was a research associate at Bristol University.

1. This is a slight simplification of what SGX provides. Instead of letting IEEs create signatures as reports, SGX exposes to IEEs a call that allows them to produce a MAC tag (under a symmetric platform key) on the output and the content of the IEE. In order to let users provide reports for external agents, it also initializes a specific IEE dedicated to converting such tags into digital signatures with an asymmetric platform key certified by Intel.

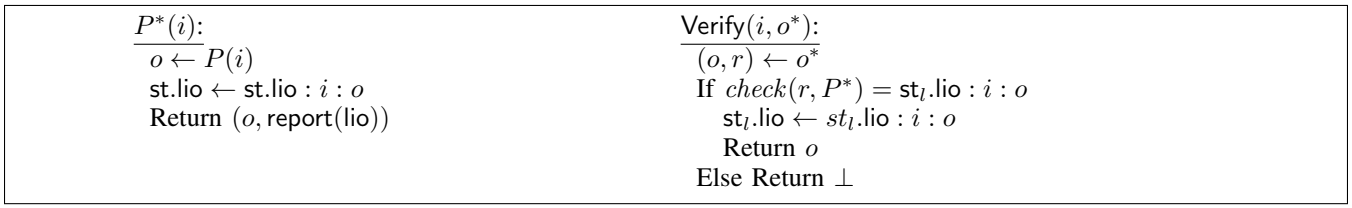

Figure 1. Pseudo code for IEE backed attestation [3]

The release of Intel SGX is likely to significantly increase the number of applications that will rely on IEEs. The complexity of these applications is however also likely to introduce security flaws in the design of these application. Symbolic models, following the seminal ideas of Dolev and Yao [4], abstract away many details of cryptographic implementations and are well suited for reasoning about complex protocols and their security properties. Indeed in such highly concurrent protocol executions, flaws in the underlying protocol logic can often be exploited to break the expected security properties. Symbolic models have been used successfully to find attacks, e.g., on authentication standards [5] and commercial PKCS#11 tokens [6] to name only a few. Applying such verification techniques to IEE based security sensitive applications would provide a valuable tool to evaluate their security. However, existing techniques do not offer direct support for verifying protocols that exploit report functionalities offered by modern IEEs. One possible approach would be to provide a detailed model of the cryptographic protocols underlying the report functionality for a particular IEE. While it is certainly interesting to study these mechanisms, the verification of the resulting models, when composed with complex applications that make use of these mechanisms, is unlikely to scale up. Moreover such models are very specific to a particular IEE. Finally, a direct modelling of the fact that programs are passed as arguments to some system calls offered by the IEEs, requires a modelling languages with support for higher-order value passing, which, to the best of our knowledge, is not supported by any security protocol verification tool.

1.1. Our contributions

A symbolic model for reasoning about IEEs. In this paper we present a symbolic model for reasoning about protocols relying on IEEs. Our aim is to provide a model that directly supports reporting and allows us to reason about applications that make use of this functionality, rather than giving a precise model of its realisation.

In order to model the report capability of IEEs we introduce the SℓAPIC process calculus, that extends SAPIC (Stateful Applied Pi Calculus) [7] with *locations* and a *report instruction*. We introduce locations as an abstraction of programs: rather than letting a report bind a value to a process itself, we bind it to a *location* attached to the given process. This avoids the need for a higher-order calculus and also provides a large amount of flexibility. For instance, if each process is modelled with a distinct location (where locations are terms that may also depend on previously input arguments) binding the result to the process or the location is virtually equivalent. Another important aspect of our model is that, in addition to a full control of the network, the adversary also controls IEE enabled machines. In particular, we model that the adversary can initialise IEEs running arbitrary programs and, through these programs, perform calls to the security module. This is achieved by granting the adversary the ability to directly forge some reports: while reports on trusted locations, i.e., honestly generated programs, may only be produced by the protocol, the adversary may forge reports for any other locations. Technically, we define a predicate H which holds on what we consider to be a trusted location (i.e. code whose properties our security properties rely on) and let the adversary produce any $\mathsf{report}(m, Q)$ with $\neg H(Q)$. This requires the introduction of *guarded function symbols*, i.e., function symbols that an adversary may only apply given that some predicate holds on the function inputs, which we believe to be of independent interest. This over-approximation allows the adversary to directly simulate code running in an IEE.

Tool support and case studies. We also provide tool support for the SℓAPIC calculus by extending the SAPIC/TAMARIN toolchain. In [7] a verification tool for SAPIC is presented which uses the TAMARIN prover [8] as a backend. At the heart of this tool is a highly optimised translation of the SAPIC calculus into multiset rewrite rules, the input format for TAMARIN. We extend this translation to SℓAPIC and prove its correctness. As we will see, the correct encoding of guarded function symbols is not straightforward and requires a slight restriction of the logic for expressing security protocols (the restriction is however minor and does not prevent expressing any relevant security properties).

Finally, we demonstrate the usefulness of the calculus and the effectiveness of the tool by analysing several protocols. We start with a model and analysis of the simplified attested computation protocol [3]. Next, we consider an attested key exchange and show that using this key exchange protocol we can also realize *secure outsourced computation* (SOC) [3]. In addition to attestation, SOC provides confidentiality of the computed values and unicity of the computation. Finally we provide models and an automated analysis of a recent licensing protocol [9] and a one time password protocol [10].

1.2. Related work

A few works [11], [3] provide security proofs of various IEE based protocols in the computational model. These works consider an abstract model of IEEs: intuitively, the IEE is modelled as a RAM machine with access to some specific cryptographic calls. Our modeling draws from this line of work, casting it in a more abstract symbolic model. While these works provide the inherently stronger security guarantees that come from working in the computational model, they do not appear to be amenable to automation as is.

Various works consider properties of assembly code running in IEEs. Notably in [12] Sinha et al. provide a tool and design methodology ensuring confidentiality of private values and code contained in IEEs. In [13] Patrignani and Clarke give an abstraction of isolation mechanisms at the source code level. These lines of research target a much lower level than ours, and properties of the code running in IEE rather than complex protocols built on top of those.

It has been noted [14] that SGX and other IEEs do not provide protection against side channel, in particular timing, attacks. While modelling this aspect of IEEs is an interesting line of work, it is outside of the scope of Dolev-Yao models and should be dealt with at a lower level.

Symbolic verification techniques provide nowadays good tool support for automated verification of security protocols, see e.g., [15], [16], [17], [8]. However, most tools such as ProVerif [15], Scyther [16] and Maude-NPA [17] do not provide support for global mutable state, i.e., state information that may be accessed and modified by parallel threads. As we will see in Section 4 our case studies however do need such stateful information. StatVerif [18] is an extension of ProVerif that provides such global states in the form of *memory cells*. However, it only provides support for a fixed number of cells, while we require them to be generated dynamically (under a replication). We therefore chose to base our model on the SAPIC/TAMARIN toolchain.

In [19], [20], Delaune et al. analyse parts of the Trusted Platform Module (TPM) [21] instruction set using ProVerif. In [22], Shao et al. model the enhanced authorization mechanism introduced in the TPM 2.0 specification and analyse it using SAPIC. These approaches are however different in the sense that they model some instructions offered by the TPM, while we directly add a report mechanism to the language.

2. Protocol model

In this section we present our formal model for specifying messages, protocols and security properties. This formalism is similar to [8], [7], but introduces guarded function symbols and extends the process calculus with locations and a report instruction.

2.1. Modelling messages as terms

The messages exchanged during protocol executions are modelled as terms in an order-sorted term algebra. In our sort system we consider messages of sort msg and two incomparable subsorts pub and $fresh$. We suppose that each of these subsorts comes with a countably infinite set of names, FN for fresh names and PN for public names. While names in PN model publicly known values, the names in FN model secret (typically randomly drawn) values, such as secret keys and nonces. For each sort s we assume an infinite set of variables, denoted $\mathcal{V}_s$. The set of all variables $\mathcal{V}$ is the union of $\mathcal{V}_s$ for all sorts s. Cryptographic operations are modelled using function symbols. The set Σ of function symbols, coming each with its arity, is called the *signature* and we denote by f/n that f is of arity n.

Given Σ, PN, FN and $\mathcal{V}$ we define the set of terms $\mathcal{T}$ to be the smallest set that contains $PN, FN, \mathcal{V}$ and is closed under application of function symbols. We denote by $names(t)$, respectively $vars(t)$, the function that given a term t returns the set of names, respectively variables, occurring in t. Moreover, we define the set of messages $\mathcal{M} = \{t | t \in \mathcal{T} \text{ and } vars(t) = \emptyset\}$, i.e., $\mathcal{M}$ is the set of ground terms, that is terms without variables.

Example 1. We consider a small signature capturing the report and check functions :

$$\Sigma = \{report/2, check/2\}$$

Then, a term attested by the IEE could be of the form :

$$report(msg, trusted\_location)$$

Next, we equip the term algebra with an equational theory. The equational theory E is defined by a set of equations among terms $t_1 = t_2$, such that $names(t_1) = names(t_2) = \emptyset$ and the set induces an equivalence relation $=_E$. We define $=_E$ to be the smallest equivalence relation which contains all equations in E and is closed under application of function symbols and substitutions of variables by terms.

Example 2. We can equip the previous signature with the following equation :

$$check(report(msg, location), location) = msg$$

Then, given a report and a location, we can verify if the report corresponds to this location.

Facts. We will use *facts* to annotate protocols, using events. Facts will also serve to define multiset rewrite rules (cf Section 3.1). To define facts we assume an unsorted signature Σ_{fact}, disjoint from Σ. Given Σ_{fact}, we define the set of facts as

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

We distinguish *linear* and *persistent* fact symbols: while linear facts may be consumed, persistent facts always remain available. By convention, persistent facts will be denoted by identifiers that start with '!'. When S is a sequence or set of facts we write $lfacts(S)$ for the multiset of all linear facts in S and $pfacts(S)$ for the set of all persistent facts in S. We assume that Σ_{fact} always contains a persistent, unary symbol !K and a linear, unary symbol Fr. $\mathcal{G}$ 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.

Predicates. We assume an unsorted signature Σ_{pred} of predicate symbols that is disjoint from Σ and Σ_{fact}. The set of *predicate formulas* is defined as

$$\mathcal{P} := \{pr(t_1, \ldots, t_k) \mid pr/k \in \Sigma_{pred}, t_i \in \mathcal{T}_\Sigma\}.$$

The semantics of a predicate is defined via a first-order formula over atoms of the form $t_1 \approx t_2$, i.e. the grammar for such formulae is

$$\langle \phi \rangle ::= t_1 \approx t_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi$$

where t_1, t_2 are terms and $x \in \mathcal{V}$. For an n-ary predicate symbol pr, $pr(x_1, ..., x_n)$ is defined by a formula ϕ_{pr} such that $fv(\phi_{pr}) \subseteq x_1, ..., x_n$, where fv denotes the free variables in a formula, i. e., variables $v \in \mathcal{V}$ not bound by $\exists v$. The semantics of the first-order formulae is as usual where we interpret $\approx$ as $=_E$.

Example 3. Suppose $isReport \in \Sigma_{pred}$ is a binary predicate symbol. We can define it as follows, so that it allows to check whether a term x_1 is a report created at location x_2:

$$\phi_{isReport}(x_1, x_2) := \exists m.report(m, x_2) \approx x_1$$

Substitutions. As usual we define a substitution σ to be a partial function from variables to terms. All substitutions are assumed to be well-typed, i.e., they only map variables of sort s to terms of sort s, or of a subsort of s. A substitution that maps x_i to t_i for $1 \leq i \leq n$ will be denoted by $\sigma = \{^{t_1}/_{x_1}, \ldots, ^{t_n}/_{x_n}\}$ and the domain of σ is defined as $\mathbf{D}(\sigma) = \{x_1, \ldots, x_n\}$. We also allow σ to be applied to terms (rather than only variables) by extending substitutions homomorphically and, as usual, write $t\sigma$ rather than $\sigma(t)$ for the application of σ on term t. A substitution σ is said to be grounding for a term t if $t\sigma$ is ground.

Sets, sequences and multisets. Given a set S we assume the following notations for sets and multisets.

- $\mathbb{N}_n$ denotes the set $\{1, \ldots, n\}$;
- S^* denotes the set of finite sequences of elements from S;
- $S^\#$ denotes the set of finite multisets of elements from S;
- $\cup^\#$ denotes multiset union, and similarly we use the superscript $^\#$ on other set operations to denote the corresponding multiset operation;
- $\in_E$ denotes set membership modulo E defined as $e \in_E S$ iff $\exists e' \in S.\ e' =_E e$; $\subset_E$, $\cup_E$, $\setminus_E$ and $=_E$ are also defined for sets in a similar way.

Finally, given a multiset S we denote by $set(S)$ the set of elements in S. We also assume the following notations for sequences.

- $[]$ denotes the empty sequence;
- $[e_1, \ldots, e_n]$ denotes the sequence of elements $e_1, \ldots, e_n$;
- $|S|$ denotes the length of sequence S, i.e., the number of elements of the sequence.

Application of substitutions are lifted to sets, sequences and multisets as expected. By abuse of notation we sometimes interpret sequences as sets or multisets; the applied operators should make the implicit cast clear.

Functions. We interpret functions between terms modulo E. When f is a function from terms to terms we require that if $x =_E y$ then $f(x) =_E f(y)$. Moreover, we let $f(x) = \bot$ when $x \notin_E \mathbf{D}(f)$ and say that g is undefined for x. Given a function g we define the function $f := g[a \mapsto b]$ such that $\mathbf{D}(f) = \mathbf{D}(g) \cup_E \{ a \}$ and $f(x) := b$ when $x =_E a$ and $f(x) := g(x)$ otherwise, i.e., f is defined as g except that it maps a to b.

Frames and deduction. A frame $\nu\tilde{n}.\sigma$ consists of a set of names $\tilde{n}$ of sort fresh and a substitution σ. We will use frames to record the messages output by a protocol during an execution. $\tilde{n}$ is the set of restricted names, that are a priori unknown to the adversary. The adversary might however be able to deduce some of these names. Using deduction rules, the adversary may indeed be able to learn new messages from the messages previously observed. We introduce a slight generalization of the usual deduction definitions. To each function symbol $f/n \in \Sigma$ we associate an n-ary predicate ϕ_f. An adversary may only apply a function symbol f when ϕ_f holds. Standard public and private function symbols are defined by defining ϕ_f to be true, respectively false.

Definition 4 (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 5. Coming back to IEEs and the report function, we want the attacker to be able to use report for any untrusted location. Therefore, if for example our trusted location is called "home", we would simply define the predicate corresponding to report as :

$$\phi_{report}(x_1, x_2) := x_2 \neq \text{“}home\text{”}$$

In a more complex fashion, one may want to trust all the locations that are created from a given seed. For example, we could trust all the locations that have "trusted" as a prefix with the following predicate :

$$\phi_{report}(x_1, x_2) := \neg(\exists z, x_2 = \text{“}trusted\text{”} + z)$$

2.2. A first-order logic for security properties

In the TAMARIN tool, traces are sequences of events. Implicitly the n-th event of a trace occurs at time n. For technical reasons that will be made clear in Section 3.3, we slightly extend this notion of trace, allowing for specifying non-integer timepoints for events in the trace. In a trace $tr = [(E_1, i_1), \ldots, (E_n, i_n)]$ where $i_1, \ldots, i_n \in \mathbb{Q}$ (we always assume $i_1 < \cdots < i_n$) and $E_1, \ldots, E_n$ are multisets of

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

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

Figure 2. Deduction rules.

facts, we define $idx(tr) = \{i_1, \ldots, i_n\}$ and $tr_{i_k} = E_k$. We write $[E_1, \ldots, E_n]$ as a shortcut for $[(E_1, 1), \ldots, (E_n, n)]$. Note that for traces whose indices are not specified, all the notions defined here coincide with the ones from TAMARIN.

In the TAMARIN tool [8] 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 6 (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 analyse a variety of security properties, including complex injective correspondence properties.

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

Definition 7 (Satisfaction relation)**.** The satisfaction relation $(tr, \theta) \vDash \varphi$ between a trace tr, a valuation θ and a trace formula φ is defined as follows:

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

For readability, we define $t_1 \gtrdot t_2$ as $\neg(t_1 \lessdot t_2 \vee t_1 \doteq t_2)$ and $(\lessdot, \not\doteq, \gtrdot)$ as expected. We also use classical notational shortcuts such as $t_1 \lessdot t_2 \lessdot t_3$ for $t_1 \lessdot t_2 \wedge t_2 \lessdot t_3$ and $\forall i \leq j.\ \varphi$ for $\forall i.\ i \leq j \to \varphi$. Moreover, we sometimes use unique existential quantification $\exists!$ as syntactic sugar where $\exists!x.\ \varphi(x)$ stands for $\exists x.\ (\varphi(x) \wedge \forall y.\ (\varphi(y) \Rightarrow x = y))$.

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

Definition 8 (Validity, satisfiability)**.** Let $Tr \subseteq (\mathcal{P}(\mathcal{G}), \mathbb{Q})^*$ 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 $traces^{pi}(P) \vDash^\exists \varphi$.

2.3. A process calculus with support for IEE

Syntax. The SℓAPIC calculus expands the variant of the applied pi calculus from [7]. The complete syntax is provided in Figure 3. Informally, the semantics can be explained as follows. The null process (0) is the terminal process which does nothing. The parallel operator $P|Q$ denotes that processes P and Q are executed in parallel, while the replication $!P$ defines the execution of an unbounded number of parallel copies of P. The calculus provides constructs for communication among parallel processes. out(M, N) denotes the output of message N on channel M. Similarly, in(M, N) denotes the input on channel M of a message that matches N. Variables in N are bound by the input. If the term M is deducible by the adversary, then he may intercept the output, respectively provide the input. The conditional behaves as expected and branches according to the predicate $pr(M_1, \ldots M_n)$. Events are merely used as annotations and referred to when specifying security properties. They do not influence the control flow. Name restriction νn allows to declare new names that are unknown to the attacker. Intuitively, restriction models random number generation and is used for generating fresh keys and nonces. The calculus also contains constructions for manipulating global states. States may be used to model a shared memory, or a simple database accessed by processes, potentially executed in parallel. The command insert M, N associates the value N to M. M may be thought of as a memory cell or the key of a database entry. The delete command allows to undefine M and lookup allows to retrieve the latest value associated to M; if the lookup fails, i.e., no value is associated to M, the else branch is taken. Finally, the lock and unlock commands allow implementation of mutually exclusive access to memory cells, which is essential to avoid

race conditions and write correct processes. We highlight the two constructs that differ from the original SAPIC calculus.

We first introduce locations for processes. P running at location ℓ is denoted by $(P)@\ell$ where ℓ is a term. In the context of IEEs, this construction is intended to represent an IEE running some particular program represented by ℓ. We do not enforce a one to one mapping between locations and processes running at said location, and let the user ensure this. In an actual IEE, the behaviour of a call to the security module would depend on the code of P instead of a user defined location, however by allowing more freedom in the allocation of locations, we avoid having to deal with a higher order calculus.

The other construct we introduce here is the call to the report function, denoted by let $x = \mathsf{report}(M)$ in P. It is intended to model the call to the security module that returns a certificate that program has produced output M.

Example 9. We give here a simplified example of the remote process presented in Section 1. This process simply takes an input, applies the free function prog and returns a certificate that the output has been produced at location l. Location l is assumed to be an honest location only used for this process.

$$\big(\mathrm{in}(c,i);\ \text{let } x = \mathsf{report}(\mathrm{prog}(i)) \text{ in } \mathrm{out}(c,\langle \mathrm{prog}(i), x\rangle)\big)@l$$

Operational semantics. We are now ready to define the operational semantics of our calculus. The semantics is defined by a labelled transition relation between process configurations. A *process configuration* is a 5-tuple $(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$ where

- $\mathcal{E} \subseteq FN$ is the set of fresh names generated by the processes;
- $\mathcal{S} : \mathcal{M} \to \mathcal{M}$ is a partial function modelling the store;
- $\mathcal{P}$ is a multiset of ground processes with their location representing the processes executed in parallel; implicitly, if no location is specified, we give the process the default location $\bot$;
- σ is a ground substitution modelling the messages output to the environment;
- $\mathcal{L} \subseteq \mathcal{M}$ 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 $\stackrel{f}{\Rightarrow}$ for $\to^* \xrightarrow{f} \to^*$.

Most rules are identical to the SAPIC semantics if we ignore locations. Our first extension is the rule for setting locations: $\mathcal{P} \cup^{\#} \{(P@\mathsf{loc}_2, \mathsf{loc}_1)\} \longrightarrow \mathcal{P} \cup^{\#} \{(P, \mathsf{loc}_2)\}$ (only showing the change in the multiset of processes). Intuitively this rule ensures that the apparent location of a process is the innermost one specified in its syntax. This is in contrast to notions of location attached to the network [23], [24] which tend to see locations as trees designating the path to a process. However, this modelling fits the propagation of identities for IEEs: if an IEE initializes another IEE, the identity of the second IEE only depends on the code passed explicitly at the initialization step and does not retain information on the initializing IEE. In other words the identity of an IEE only depends on its content and not on how it was created. Our second addition is the report rule which, when restricted to the multiset of processes, is $\mathcal{P} \cup^{\#} \{(\text{let } x = \mathsf{report}(M) \text{ in } P, \mathsf{loc})\} \longrightarrow \mathcal{P} \cup^{\#} \{P\{^{report(M,\mathsf{loc})}/_{x}\}\}$. This rule simulates a call to the security module, fetches the identity of the calling process, and produces a certificate $report(M, \mathsf{loc})$ that message M was reported on by a process with identity loc. We also sometimes use syntactic sugar for assignment and write let $x = t$ in P for $P\{^{t}/_{x}\}$.

We can now define the set of traces, i.e., possible executions that a process admits.

Definition 10 (Traces of P)**.** Given a ground process P we define the *set of traces of* P as

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

3. SℓAPIC in SAPIC

In this section we show how we can extend the SAPIC tool to verify SℓAPIC processes. We begin by shortly recalling the multiset rewrite rule (msr) framework (used by the TAMARIN prover) and SAPIC. Next we show how to encode reports and locations, as well as the treatment of *guarded* function symbols.

3.1. TAMARIN and SAPIC

SAPIC [7] is a verification tool for verifying protocols written in a stateful extension of the applied pi calculus (the calculus presented in Section 2.3 without support for locations and reports). At the heart of SAPIC is an encoding of this calculus in multiset rewrite rules (msr) that can then be analysed by the TAMARIN prover [8] which is used as a verification backend.

MSR rules. We now give a short overview of the msr rule framework. Detailed definitions are given in the long version [25] . An MSR rule is a triple (l, a, r), generally written $l -[\ a\ ]\to r$, where l, a, r are multisets of facts. The idea is that given a set of facts, we can consume some of the facts to create others according to the rules. We can then consider all the possible reduction from a set for a specific set of rules and have a semantic.

Formally, given a set R of msr rules we define a transition relation among multisets of facts: $S \xrightarrow{a}_R S'$ if there is an msr rule $l -[\ a\ ]\to r$ which is a ground instance of a rule in R such that

- $l \subseteq^{\#} S$, and
- S' is defined as $(S \setminus^{\#} lfacts(l)) \cup^{\#} r$

$\langle P, Q \rangle$ ::=

0	*null process*
$P \mid Q$	*parallel composition*
$!P$	*replication*
$\mathrm{out}(M, N);\ P$	*output of N on channel M*
$\mathrm{in}(M, N);\ P$	*input*
if $pr(M_1, \ldots M_n)$ then P else Q	*conditionnal*
event $F;\ P$	*event*
$\nu n;\ P$	*binding of a fresh name*
insert $M, N;\ P$	*set value of state M to N*
delete $M;\ P$	*delete state M*
lookup M as v in P else Q	*read the state*
lock $M;\ P$	*lock a state*
unlock $M;\ P$	*unlock a state*
$(P)@M$	*localised process*
let $x = \mathsf{report}(M)$ in P	*reporting according to location*

Figure 3. Syntax of our process calculus

Modelling *freshness* of names requires some care. We suppose no rule may create a name of sort fresh, except the distinguished rule

$$\textsc{Fresh} : [] \dashrightarrow [\mathsf{Fr}(x : \mathit{fresh})]$$

which we suppose to always be part of the set of msr rules we consider. Then an *execution* of R is a sequence $\emptyset \xrightarrow{A_1}_R \ldots \xrightarrow{A_n}_R S_n$ where for a given name a of sort *fresh* the fact $\mathsf{Fr}(a)$ is introduced at most once, i.e. for all $i \neq j$, we have that $(S_{i+1} \setminus^{\#} S_i) = \{\mathsf{Fr}(a)\}$ implies that $(S_{j+1} \setminus^{\#} S_j) \neq \{\mathsf{Fr}(a)\}$.

Given a set of msr rules R we define the set of traces $\mathit{traces}^{msr}(R)$ to contain all sequences $[A_1, \ldots, A_n]$ such that $\emptyset \xrightarrow[R]{\emptyset}{}^{*} \xrightarrow[R]{A_1} \xrightarrow[R]{\emptyset}{}^{*} \ldots \xrightarrow[R]{\emptyset}{}^{*} \xrightarrow[R]{A_n} \xrightarrow[R]{\emptyset}{}^{*} S_n \in \mathit{exec}^{msr}(R)$ is an execution of R.

The TAMARIN prover also defines a set of msr rules MD that model message deduction by the adversary. The rules are displayed in Figure 5. The facts $\mathsf{Out}(t)$ and $\mathsf{In}(t)$ model that a protocol outputs, respectively inputs the term t. An output adds knowledge to the adversary (MDOUT), modelled by the fact symbol $!\mathsf{K}$ while an input requires the message to be known (MDIN). An adversary may know public names (MDPUB), generate fresh names (MDFRESH) and apply function symbols to known terms (MDAPPL).

SAPIC. In [7], Kremer and Künnemann define a translation from a SAPIC process to a set of msr rules. Given a process P, the corresponding set of msr rules is denoted $[\![P]\!]$. For example, the translation of an output action in SAPIC correspond directly to adding something to the knowledge of the attacker.

One important point of the translation is how conditionals, lookups and locks are handled. A conditional

$$\text{if } pr(M_1, \ldots, M_n) \text{ then } P \text{ else } Q$$

is simply translated to two possible transitions with labels $\mathrm{Pred}_{pr}(M_1, \ldots, M_n)$ for the then branch and $\mathrm{Pred\_not}_{pr}(M_1, \ldots, M_n)$ for the else branch. The check whether $pr(M_1, \ldots, M_n)$ holds or not is then encoded in the formula by adding an *axioms* of the form:

$$\forall x_1, \ldots, x_k, i.\ \mathrm{Pred}_{pr}(x_1, \ldots, x_k)@i \implies \phi_{pr}$$

and

$$\forall x_1, \ldots, x_k, i.\ \mathrm{Pred\_not}_{pr}(x_1, \ldots, x_k)@i \implies \neg(\phi_{pr})$$

The conjunction of all these axioms is denoted α and is used to filter out any invalid traces. It is shown in [7] that the proposed translation is sound and complete for well-formed processes and formulas. (A process and a formula are basically well-formed if they do not use any reserved facts, used for the translation.)

Theorem 1 ([7]). *Given a well-formed ground process P and a well-formed trace formula φ we have that*

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

where $\star$ is either $\forall$ or $\exists$, $[\![\varphi]\!]_{\forall} = \alpha \implies \varphi$ and $[\![\varphi]\!]_{\exists} = \alpha \implies \varphi$.

3.2. Encoding locations and reports

The syntax of SℓAPIC allows two additional constructs: $(P)@M$ and let $x = \mathsf{report}(M)$ in P. However, these two constructs can be seen as syntactic sugar and a SℓAPIC process can be easily rewritten into a SAPIC process by replacing let $x = \mathsf{report}(M)$ in P with $\{P = \{^{report(M,\ell)}/_{x}\}\}$ when P is in the scope of the location ℓ.

Definition 11. We define the function rw_0 as

$$rw_0((P)@\ell', \ell) = rw_0(P, \ell')$$
$$rw_0(\text{let } x = \mathsf{report}(y) \text{ in } P, \ell) = rw_0(P\{^{report(y,\ell)}/_{x}\}, \ell)$$

and the function rw to be the homomorphic extension of rw_0 to all processes.

The following proposition directly follows from the operational semantics.

Standard operations:

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(0,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P|Q,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P,\mathsf{loc}),(Q,\mathsf{loc})\},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(!P,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(!P,\mathsf{loc}),(P,\mathsf{loc})\},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\nu a;P,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E}\cup\{a'\},\mathcal{S},\mathcal{P}\cup^{\#}\{(P\{a'/a\},\mathsf{loc})\},\sigma,\mathcal{L}) \quad \text{if } a' \text{ is fresh}$$

$$(\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) \xrightarrow{K(M)} (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) \quad \text{if } \nu\mathcal{E}.\sigma\vdash M$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\mathrm{out}(M,N);P,\mathsf{loc})\},\sigma,\mathcal{L}) \xrightarrow{K(M)} (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P,\mathsf{loc})\},\sigma\cup\{^{N}/_{x}\},\mathcal{L}) \quad \text{if } x \text{ is fresh and } \nu\mathcal{E}.\sigma\vdash M$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\mathrm{in}(M,N);P,\mathsf{loc})\},\sigma,\mathcal{L}) \xrightarrow{K(\langle M,N\tau\rangle)} (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P\tau,\mathsf{loc})\},\sigma,\mathcal{L}) \quad \text{if } \nu\mathcal{E}.\sigma\vdash M,\nu\mathcal{E}.\sigma\vdash N\tau \text{ and } \tau \text{ is grounding for } N$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\mathrm{out}(M,N);P,\mathsf{loc}_1),(\mathrm{in}(M',N');Q,\mathsf{loc}_2)\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mathsf{loc}_1),(Q\tau,\mathsf{loc}_2)\},\sigma,\mathcal{L}) \quad \text{if } M=_E M' \text{ and } N=_E N'\tau \text{ and } \tau \text{ grounding for } N'$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\text{if } pr(M_1,\ldots,M_n) \text{ then } P \text{ else } Q,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mathsf{loc})\},\sigma,\mathcal{L}) \quad \text{if } \phi_{pr}\{^{M_1}/_{x_1},\ldots,^{M_n}/_{x_n}\} \text{ is satisfied}$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\text{if } pr(M_1,\ldots,M_n) \text{ then } P \text{ else } Q,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(Q,\mathsf{loc})\},\sigma,\mathcal{L}) \quad \text{if } \phi_{pr}\{^{M_1}/_{x_1},\ldots,^{M_n}/_{x_n}\} \text{ is not satisfied}$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(\text{event}(F);P,\mathsf{loc})\},\sigma,\mathcal{L}) \xrightarrow{F} (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{(P,\mathsf{loc})\},\sigma,\mathcal{L})$$

Operations on global state:

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\text{insert } M,N;\ P,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S}[M\mapsto N],\mathcal{P}\cup^{\#}\{(P,\mathsf{loc})\},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\text{delete } M;\ P,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S}[M\mapsto\bot],\mathcal{P}\cup^{\#}\{(P,\mathsf{loc})\},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q\ ,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P\{V/x\},\mathsf{loc})\},\sigma,\mathcal{L}) \text{if } \mathcal{S}(M)=_E V$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q,\mathsf{loc})\ \},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(Q,\mathsf{loc})\},\sigma,\mathcal{L}) \text{if } \mathcal{S}(M) \text{ is undefined}$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\text{lock } M;\ P,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P,\mathsf{loc})\},\sigma,\mathcal{L}\cup\{\,M\,\}) \quad \text{if } M\notin_E\mathcal{L}$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\text{unlock } M;\ P,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P,\mathsf{loc})\},\sigma,\mathcal{L}\setminus_E\{\,M\,\})$$

Operations for locations and reporting:

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P@\mathsf{loc}_2,\mathsf{loc}_1)\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(P,\mathsf{loc}_2)\},\sigma,\mathcal{L})$$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{(\text{let } x=\mathsf{report}(M) \text{ in } P,\mathsf{loc})\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{^{\mathsf{report}(M,\mathsf{loc})}/_{x}\}\},\sigma,\mathcal{L})$$

Figure 4. Operational semantics

$$\begin{array}{rcll}
\mathsf{Out}(x) & \multimap[\ ]\to & !\mathsf{K}(x) & (\text{MDOut})\\
!\mathsf{K}(x) & \multimap[\ K(x)\ ]\to & \mathsf{In}(x) & (\text{MDIn})\\
 & \multimap[\ ]\to & !\mathsf{K}(x:pub) & (\text{MDPub})\\
\mathsf{Fr}(x:fresh) & \multimap[\ ]\to & !\mathsf{K}(x:fresh) & (\text{MDFresh})\\
!\mathsf{K}(x_1),\ldots,!\mathsf{K}(x_k) & \multimap[\ ]\to & !\mathsf{K}(f(x_1,\ldots,x_k)) \text{ for } f\in\Sigma^k & (\text{MDAppl})
\end{array}$$

Figure 5. The set of rules MD.

Proposition 12. *Let p be a ground process. We have that $traces^{pi}(P) = traces^{pi}(rw(P,\bot))$.*

An example of rewriting a process is given in figure 6. In the following we suppose that we always apply the rw function as a first step and simply write $[\![P]\!]$ for $[\![rw(P,\bot)]\!]$.

3.3. Extending SAPiC with guarded functions

In our model we consider a more general type of function symbols than usual: function symbols may be guarded by a predicate that is evaluated to check whether the attacker is allowed to apply the function on the given arguments. SAPiC and Tamarin directly support public and private symbols, but not the more general guarded functions introduced here.

Guarded function symbols can be added to SAPiC and Tamarin in a rather elegant way: we simply modify the set of message deduction rules MD (Figure 5) by annotating the MDAppl rule as follows:

$$!\mathsf{K}(x_1),\ldots,!\mathsf{K}(x_k) \multimap[\ \mathrm{Pred}_{\phi_f}(x_1,\ldots,x_k)\ ]\to !\mathsf{K}(f(x_1,\ldots,x_k)) \quad \text{for } f\in\Sigma^k \qquad (\text{MDAppl}_{\mathcal{I}})$$

Figure 6. Original and rewritten process

We denote by $\text{MD}_{\mathcal{I}}$ the set of rules obtained by replacing (MDAPPL) in MD with $\text{MD}_{\mathcal{I}}$ and by $[\![\ ]\!]_{\mathcal{I}}$ the translation where MD has been replaced by $\text{MD}_{\mathcal{I}}$. Adding an axiom

$$\alpha_f = \forall x_1, \ldots, x_k, i.\, \text{Pred}_{\phi_f}(x_1, \ldots, x_k)@i \Rightarrow \phi_f(x_1, \ldots, x_k)$$

guarantees that all traces where the attacker applies a function symbol that is not allowed are then discarded. The adaptation of the SAPIC translation is easily shown to be correct.

Proposition 13. *Given a well-formed ground process P and a well-formed trace formula φ we have that*

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

where $\star$ is either $\forall$ or $\exists$, $[\![\varphi]\!]_{\forall} = (\alpha \wedge \bigwedge_{f\in\Sigma} \alpha_f) \Longrightarrow \varphi$ and $[\![\varphi]\!]_{\exists} = (\alpha \wedge \bigwedge_{f\in\Sigma} \alpha_f) \Longrightarrow \varphi$.

The proof requires the generalization of a Lemma in [7] that shows that the intruder deduction in SAPIC and in TAMARIN coincide (see the long version [25]). Unfortunately, even though the $[\![\ ]\!]_{\mathcal{I}}$ translation is correct, TAMARIN does not allow to write msr rules which directly manipulate !K facts as these are reserved for internal use. As this part of the prover is highly optimized and very sensitive we instead have to declare guarded function symbols as private and encode the rule $\text{MDAPPL}_{\mathcal{I}}$ by the following one

$$\mathsf{In}(x_1), \ldots, \mathsf{In}(x_k) \dashv[\ \text{Pred}_{\phi_f}(x_1, \ldots, x_k)\ ]\mapsto \mathsf{Out}(f(x_1, \ldots, x_k))$$

$$\text{for } f \in \Sigma^k \qquad (\text{MDAPPL}_{\mathcal{R}})$$

We denote the resulting set of message deduction rules by $\text{MD}_{\mathcal{R}}$ and denote by $[\![\ ]\!]_{\mathcal{R}}$ the translation replacing MD with $\text{MD}_{\mathcal{R}}$. Any application of the rule $\text{MDAPPL}_{\mathcal{I}}$ $S \dashv[\ \phi_f(t_1, \ldots, t_k)\ ]\mapsto S'$ can of course be simulated by k successive applications of MDIN, an application of $\text{MDAPPL}_{\mathcal{R}}$ and an application ofMDOUT:

$$S \dashv[\ K(t_1)\ ]\mapsto \ldots \dashv[\ K(t_k)\ ]\mapsto S \cup \{\mathsf{In}(t_1), \ldots \mathsf{In}(t_k)\}$$

$$\dashv[\ \text{Pred}_{\phi_f}(t_1, \ldots, t_k)\ ]\mapsto \dashv[\ ]\mapsto S'$$

This results into an execution among the same multisets S and S', but generates k extra trace facts $K(t_i)$. Hence these two translations will not satisfy the same formulas. Consider for instance the following msr rule r

$$\mathsf{In}(f(x)) \dashv[\ ev(x)\ ]\mapsto$$

which simply raises the event $ev(x)$ when a term of the form $f(x)$ is received. Such a rule is not directly generated by the translation but illustrates the underlying difference between $[\![\ ]\!]_{\mathcal{I}}$ and $[\![\ ]\!]_{\mathcal{R}}$. We easily see that

$$\text{MDAPPL}_{\mathcal{R}} \cup^{\#} \{r\}^{\#} \models^{\forall} \forall x, i.\, \exists j < i.\, ev(x)@i \Longrightarrow K(t)@j$$

and

$$\text{MDAPPL}_{\mathcal{I}} \cup^{\#} \{r\}^{\#} \not\models^{\forall} \forall x, i.\, \exists j < i.\, ev(x)@i \Longrightarrow K(t)@j$$

We therefore restrict the class of formulae we consider to a class for which the additional $K(t)$ facts preserve satisfaction. When verifying validity claims we consider the class of formulae in which, when in negation normal form, the knowledge atoms always appear under a negation. By duality, when checking satisfiability, we require that no knowledge atom appears under a negation when in negation normal form.

Definition 14. We denote by $\mathcal{L}$ the set of traces formulas where all terms of sort $temp$ belong to $\mathcal{V}_{temp}$. We denote by $\mathcal{L}^+$ (resp. $\mathcal{L}^-$) the subsets of $\mathcal{L}$ where all the occurrences of the fact K are positive (resp. negative), defined inductively as follows:

- If ψ is atomic then $\psi \in \mathcal{L}^+$
- If ψ is atomic and $\psi \neq K(t)@i$ then $\psi \in \mathcal{L}^-$
- If $\psi_1, \psi_2 \in \mathcal{L}^+$ (resp. $\psi_1, \psi_2 \in \mathcal{L}^-$), then $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ belong to $\mathcal{L}^+$ (resp. $\mathcal{L}^-$)
- If $\psi \in \mathcal{L}^+$ (resp. $\psi \in \mathcal{L}^-$), then $\exists x.\psi, \forall x.\psi \in \mathcal{L}^+$ (resp. $\exists x.\psi, \forall x.\psi \in \mathcal{L}^-$)
- If $\psi \in \mathcal{L}^+$ (resp. $\psi \in \mathcal{L}^-$), then $\neg\psi \in \mathcal{L}^-$ (resp. $\neg\psi \in \mathcal{L}^+$)

This class is general enough to tackle most trace properties, as demonstrated by our examples. Indeed, it seems natural that when checking a validity claim (a security property) we wish to ensure that on all traces the attacker does not know a certain term, while for satisfiability (an attack trace) we wish to check whether on some trace the attacker does know a term.

We can now show that for the translation $[\![\_]\!]_{\mathcal{R}}$ is correct for this restricted set of formulae.

Theorem 2. *Given a well-formed ground process P and a well-formed trace formula φ we have that*

$$\forall \psi \in \mathcal{L}^+.\, traces^{pi}(P) \models^{\exists} \psi \Leftrightarrow traces^{msr}([\![P]\!]_{\mathcal{R}}) \models^{\exists} [\![\psi]\!]_{\exists}$$

$$\forall \psi \in \mathcal{L}^-.\, traces^{pi}(P) \models^{\forall} \psi \Leftrightarrow traces^{msr}([\![P]\!]_{\mathcal{R}}) \models^{\forall} [\![\psi]\!]_{\forall}$$

where $[\![\varphi]\!]_{\forall} = (\alpha \wedge \bigwedge_{f\in\Sigma} \alpha_f) \Longrightarrow \varphi$ and $[\![\varphi]\!]_{\exists} = (\alpha \wedge \bigwedge_{f\in\Sigma} \alpha_f) \Longrightarrow \varphi$.

Let us define, for any multiset rewrite rules system R, $T_{\mathcal{I}}(R) = traces^{msr}(R \cup MD_{\mathcal{I}})$ and $T_{\mathcal{R}}(R) = traces^{msr}(R \cup MD_{\mathcal{R}})$. The core of the proof of Theorem 2

is the fact that a formula ψ in $\mathcal{L}^+$ is satisfied on $T_{\mathcal{R}}(R)$ if and only if it is satisfied on $T_{\mathcal{I}}(R)$. We will now sketch the main steps of the proof. Details are postponed to the long version [25] .

We first show that the traces of $(R \cup MD_{\mathcal{R}})$ subsume the traces of $(R \cup MD_{\mathcal{I}})$.

Lemma 15. *Let R be a set of multiset rewrite rules. We have that* $T_{\mathcal{R}}(R) \subset T_{\mathcal{I}}(R)$.

Intuitively, every time we apply $MD_{\mathcal{R}}$ in a trace we could also apply $MD_{\mathcal{I}}$, because when we have $\mathsf{In}(t_k)$ we also have $!K(t_k)$. This means that every trace of $T_{\mathcal{R}}(R)$ is also in $T_{\mathcal{I}}(R)$. The Lemma yields that if ψ is satisfied on $T_{\mathcal{R}}(R)$, i.e., there exists a trace in $T_{\mathcal{R}}(R)$ that satisfies ψ, it is also satisfied on $T_{\mathcal{I}}(R)$. With Propositions 13, we then have the first part of the Theorem.

To prove the converse, given a trace $t \in T_{\mathcal{I}}(R)$ that witnesses the satisfaction of ψ, we build a witness trace $t' \in T_{\mathcal{R}}(R)$ which also satisfy ψ. We construct t' by adding the relevant knowledge facts (corresponding to MDIN applications) to allow for the application of the $\text{MDAPPL}_{\mathcal{R}}$ rule instead of the $\text{MDAPPL}_{\mathcal{I}}$ rule. The idea is that because we are in the $\mathcal{L}^+$ fragment of the logic, adding K facts to the trace t will not invalidate the formula ψ. Then, as we can construct a valid trace of $T_{\mathcal{R}}(R)$ which satisfies ψ, we conclude with Propositions 13.

4. Implementation and Case studies

We validate our approach experimentally by checking IEE based protocols existing in the literature, starting from the relatively straightforward attested computation protocol from [3], and following with attested key exchange and secure outsourced computation from the same paper. We also verify the licensing protocol from [9] and a one time password protocol proposed in [10].

We have extended the SAPIC tool to handle all additional constructs of SℓAPIC: the protocol specification language and the underlying translation now support locations, reports and guarded function symbols. All our case studies are proven automatically by the SℓAPIC tool, sometimes with the help of a few additional lemmas. Except for one example (the OTP protocol), the lemmas are merely used to speed up the proof as lemmas can be reused several times avoiding repeated re-computation of parts of the proof. The lemmas are generally a direct translation of the properties we expect for the protocol. For instance, most of the lemmas state that shared secret keys are indeed secret and match; others verify the matching of sessions, and in some specific protocol (OTP and licensing), we need to check the unique usage of a token. The implementation and models are part of the tamarin-prover repository[2].

2. https://github.com/tamarin-prover/tamarin-prover

4.1. Attested computation

4.1.1. Definition. Abstracting from [3], we now formally define what is an attested computation (AC) protocol and its desired property. Intuitively an AC protocol consists of two parts. The first part is a compilation mechanism (denoted later as A), that takes as input a program P and returns an instrumented program P^* which is to be run in an IEE. The second part is a verification program, denoted later by B, that checks whether outputs were indeed produced by the compiled program, running in an IEE.

Given a program P, that we see here as a ternary function symbol (with arguments current state, input, randomness), we model the instrumented version P^* as a sequential process (without parallel composition nor replications) $A(P, i, lio, r, st)$, where

- i is the next input to be processed,
- lio is the current list of inputs/outputs,
- r is the next randomness to passed to P and
- st the current state of the program.

P^* is expected to return the output $o = P(st, i, r)$ and also sends the attested output on a public channel.

We model the verifier as a sequential process $B(i, o^*, st)$. We assume that B assigns variables b and o. Given an input i and an attested output o^*, b is a Boolean value representing whether the verifier accepts the attested output and o is the corresponding (decoded) output.

Informally, the AC security definition from [3] aims at ensuring the following property.

> *"If a trace t is accepted by the verifier, then t is indeed a valid trace of program P executed on inputs passed to the instrumented version P^* running in an IEE."*

In order to model this security property we define two contexts R and V for bookkeeping messages passed to the remote process and to the local verifier. A context is simply a process with a hole, denoted by _. Given a context $C[\_]$ and a process P we write $C[P]$ for the process obtained by syntactically replacing _ by P.

```
R(P,l)[_] =
!(ν st; insert st, init;
  !(lock st; lookup st as lio;
    in(i); ν r;
    _ ;
    insert st,<i,o,lio>;
    unlock(st);
    )
  )@l


V(l)[_] =
  !(ν st; insert st, init;
    !(in(i,o*); lock st;
      lookup st as lio in
      _ ;
      if b then
        event Local(<i,o,lio>,l);
```

```
        insert st,<i,o,lio>;
        unlock(st)
)
```

The context R starts by creating a new store st (initialized to the constant *init*) which saves the list of inputs/outputs processed so far. For each invocation, it retrieves the list of inputs/outputs in the state st, receives an input i, draws a new random value r for this invocation. Finally it executes its argument (the hole _), which is expected to be filled in by A. We suppose that A logs the current expected trace (i.e., the updated sequence of inputs/outputs) of this remote instance using the event `Remote(<`i, o, lio`>,`l`)`. The context V does a similar bookkeeping for the verifier process. The main difference is that it executes the verifier process and checks the verification outcome (the Boolean b) before it logs the extended trace through the event `Local(<`i, o, lio`>,l)`.

We let

- $A(P, i, lio, r, st)$ be a sequential process that models the instrumented version of P and assigns the output computed by P to the variable o, and
- $B(i, o^*, st)$ be a sequential verifier process which assigns values to variables b and o, P be a function symbol of arity 3 and l a location.

The tuple (A, B, l, P) guarantees attested computation if and only if :

$$!V(l)[B(i, o^*, st)] \mid !R(P, l)[A(P, i, lio, r, st)] \models^{\forall}$$
$$\forall i, m.\ Local(m, l)@i \Rightarrow \exists j < i.\ Remote(m, l)@j$$

Intuitively, R runs the program P on a new input and uses A to attest the output. Then, V receives both the input and a value corresponding to the attestation, and asks B to check if the attestation is valid. If B sets b to true, it means that all checks succeeded, and V may raise the corresponding event. The protocol is indeed an attestation protocol if the list of inputs and outputs of the verifier is a prefix of the one of the providers. Therefore, as $Local$ and $Remote$ contain the corresponding list of inputs and outputs, every $Local$ event must be preceded by a matching $Remote$ event.

4.1.2. Modeling. We use SℓAPIC to model the AC protocol from [3]. The main idea of this protocol is that every output produced by P will be reported upon, together with the list of inputs/outputs received so far. The verifier then checks consistency of the list of inputs/outputs.

The SℓAPIC processes for A and B are described in Figure 7. The complete system wraps these processes into the bookkeeping contexts R and V and composes them in parallel. Note that the only honest location here is l_P (modelled by the predicate ϕ_{report}). Indeed, in this modelling, we ensure that only the compiled program runs at l_P. The adversary controls all other locations. Checking in SℓAPIC that the resulting protocol guarantees attested computation, we prove security of this AC protocol in 5 seconds.

4.1.3. Attacks on weak versions. We studied two weaker versions of the AC protocol where the AC property is not satisfied. The two attacks we present here are found automatically using SℓAPIC.

The first attack occurs when instead of sending out a report on the previous list of inputs/outputs together with the result, the process A sends out a session identifier only. More precisely, on first activation, A starts by deriving a fresh session identifier s, and then proceeds as presented as in Figure 7 but sets x to be $\mathsf{report}(s, i, o)$. Protocol B of Figure 7 is modified accordingly to check for the session identifier. This version is refered to as sid-AC. In this weak version, the adversary is able to force the verifier to accept the second input/output of a trace of length 2 without having previously submitted the first one. This attack is found automatically when trying to prove the AC property in 1 minute and 30 seconds.

The second attack occurs when A reports on a sequence number only, instead of the whole list of inputs/outputs. This weak version of AC is refered to as counter-AC. In this case the adversary is able to force the verifier to accept (i_1, o_1) followed by (i'_2, o'_2) where (i'_2, o'_2) belongs to a remote trace $(i'_1, o'_1), (i'_2, o'_2)$ with $(i'_1, o'_1) \neq (i_1, o_1)$. This attack is found automatically using SℓAPIC when trying to prove the AC property for traces of length 2 in 45 seconds.

4.2. Attested key exchange

In [3], the main tool for building stronger guarantees on top of attestation is an attested key exchange. Intuitively this key exchange uses the attestation guarantees to establish a shared key between a user and the IEE running the remote part of the key-exchange. The attested key exchange presented in [3] is depicted in Figure 8.

This key exchange is extremely simple: Alice provides her public key to the IEE running the remote part of the key exchange, and expects as an answer a fresh symmetric key, encrypted with her public key. Obviously, such a naive key exchange protocol is not secure against active adversaries. However, as it is shown in [3], if the remote part is executed under AC, and Alice verifies every message she receives as prescribed by the AC protocol, then this key exchange is secure. Intuitively, this is entailed by the fact that Alice gets, from the AC security definition, the assurance that the encrypted message she receives originates from an IEE honestly executing the remote part of the key exchange for her public key.

To match the definition of [3] where the public key is hardcoded in the code running remotely, instead of defining one trusted location, we define a set of trusted locations: locations of the form $\langle l, x \rangle$ correspond to IEEs running the remote part of the key exchange using x as the public key of the intended receiving party. In order to model this protocol, we let the adversary execute any number of IEEs for any public key of his choice. The precise SℓAPIC model is provided in Figure 9. We have verified an injective agreement property (between remote and local sessions) and

$$\phi_{report}(x, y) \Leftrightarrow y \neq l_P$$

```
let A(P,i,l_io,r,st)=
  let o = P(st,i,r) in
  event Remote(<i,o,lio>,l);
  let x = report(l_io,i,o) in
  out(o,x);
```

```
let B(i,o*,st) =
  let o,r = o* in
  let b = ((st,i,o) = check(l,r)) in
```

$$!R(P,l)\,[A(P,i,o,l_{io},r,st)] \;\mid\; (\; !V(l)\;\; [B(i,o^*,st)]\;)$$

Figure 7. Attested computation implementation

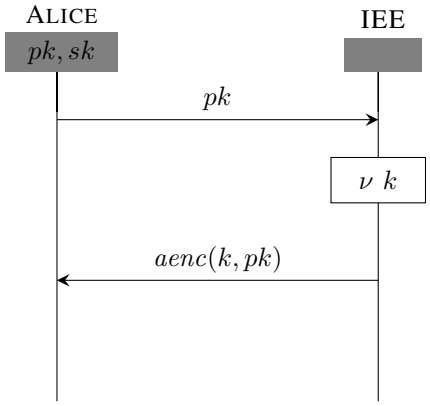

Figure 8. Basic key exchange

secrecy of the derived key using SℓAPIC. The verification of this property took about 1 second.

4.3. Secure outsourced computation

As defined in [3], a secure outsourced computation (SOC) protocol is an attested computation protocol in which, additionally, we have the secrecy of the input/output trace between the agent running the protocol and the IEE. In [3], the authors build such a protocol on top of an attested key exchange. Intuitively, this protocol first establishes a key between the agent and the remote machine, then uses this key to establish a secure channel.

With the same notations as in Section 4.1, we formally define a secure SOC protocol.

Definition 16. The tuple (A, B, l, P) guarantees secure outsourced computation if and only if

$$Proc \models^{\forall}$$
$$\forall i, m, l.\ Local(m, l)@i \Rightarrow \exists !j < i.Remote(m, l)@j$$
$$\wedge\ \forall i, m, l, l_{io}, t_1.\ Local(\langle i, P(m), l_{io}\rangle, l)@t_1 \Rightarrow \neg(\exists t_2, K(P(m))@t2)$$

where

$$Proc = !V(l)[B(i, o, st)]|!R(P, l)[A(P, i, o, lio, r, st)]$$

Using SℓAPIC, we prove that the SOC protocol given in [3] is indeed a secure outsourced computation protocol according to this definition.

4.4. Licensing

Licensing is a concept used by program developers to guarantee that their code is only used by legitimate users. The idea is that a vendor would like to authorize the use of a software only for users who have bought the software and therefore possess a valid license. In [9] Costea and Warinschi present a licensing scheme based on IEEs. We describe this protocol in Figure 10. Intuitively, the vendor and the user share a token which was established when the user bought the software. The vendor and an enclave run an attested key exchange to create a secure channel. Then, the vendor provides the program to the enclave through the secure channel. Now, the user chooses an input on which he would like to run the program and gives it through its direct access to the enclave along with the token. The enclave asks the vendor if the token is valid and if so, returns the outcome of the desired computation to the user. For simplicity we present here the version where one token corresponds to one execution of the program, but there also exists versions where a token corresponds to a fixed number n or an unlimited number of computations.

As defined in [9], the desired property of secure licensing is an injective mapping between the computations obtained by the user and the tokens produced by the vendor, i.e., any computation corresponds to a distinct token issued by the vendor. We modelled the above protocol in SℓAPIC and were able to verify the security property.

4.5. One time password

In essence, classical password based protocols allow an adversary to completely impersonate the user if he was able to intercept or break the password, e.g. using an offline dictionary attack. When using a one time password (OTP) protocol a different password is generated for every authentication request and this password is valid only once. As these passwords may not be reused a second time intercepting or breaking a password is basically useless. We consider

$\phi_{report}(x, y) \Leftrightarrow \forall z.y \neq \langle l, z \rangle$

```
let R =
  in(x); // set the public key
  !( // initiate any number of IEE for this key
    ν k; //generate the fresh key
    event SessionP(x, k);
    let r = report(aenc(k, x)) in
      out(⟨aenc(k, x), r⟩);
  )@⟨l, x⟩


let L =
  ν sk_V;
  event HonestP(pk(sk_V));
  out(pk(sk_V)); // send out public key
  in(⟨aenc(x_k, pk(sk_V)), x_r⟩); //receive encrypted key
  if  aenc(x_k, pk(sk_V)) = check(⟨l, pk(sk_V)⟩, x_r) then
    event SessionV(pk(sk_V), x_k); //session established


  ( (!R) | (!L) )
```

Figure 9. Attested key exchange

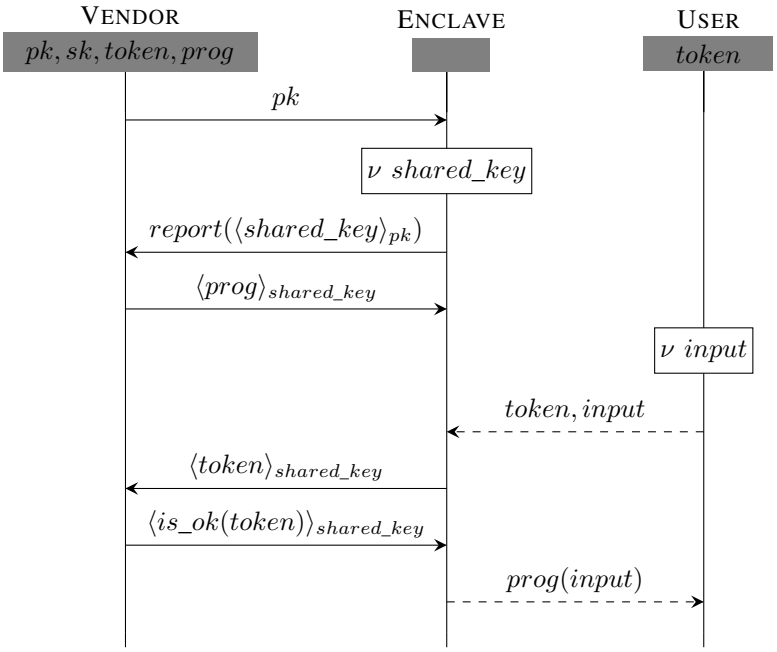

Figure 10. Licensing Protocol

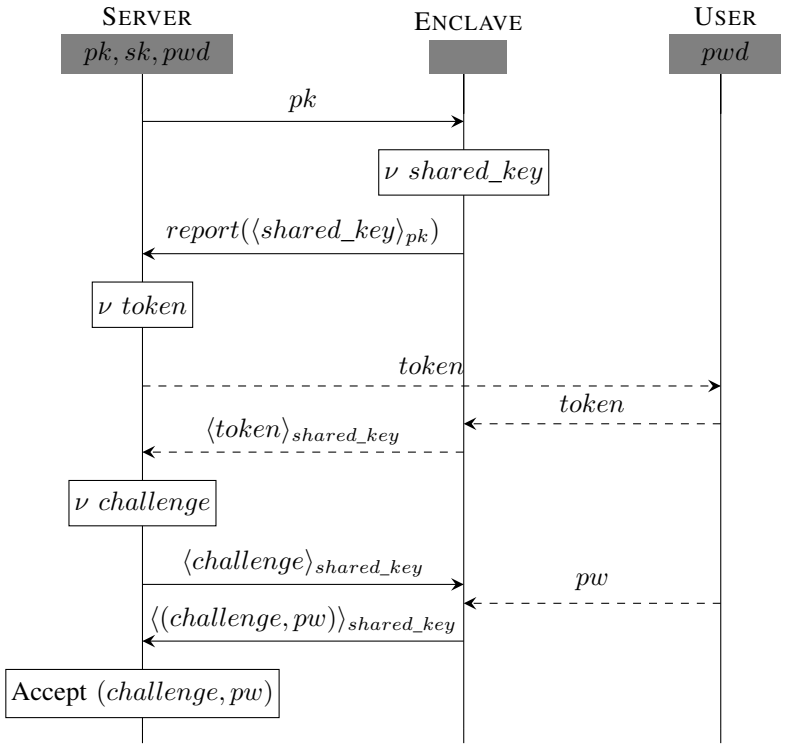

Figure 11. One time password protocol

here the one time password scheme presented in [10] and described in Figure 11.

We assume that the user has previously registered on a server and shares a long-time password pwd with the server. To authenticate to the server, the user launches an enclave which performs a key-exchange with the server. Next, the server sends a token to the user through an out-of-band channel (e.g, phone call, sms, ...). This token will ensure that the enclave belongs to the user. Next, the user forwards this token to the enclave which in turns gives it back to the server. Finally, when the user wants to authenticate to the server, he gives its password to the enclave and the enclave receives from the server a unique challenge. The enclave then creates the one time password, a combination of the challenge and the password, and sends it to the server over the secure channel. If the challenge and the password match the server accepts the authentication.

The desired security property is an injective mapping between authentication requests accepted on the server and the requests issued by the enclave. We implemented the protocol in SℓAPIC and were able to prove this property with the tool.

4.6. Summary

We summarise the verification results for our case studies in Table 1. The table indicates for each case study the execution time, the number of proof steps computed by TAMARIN and the number of helping lemmas. The lemmas are used to guide TAMARIN and proven before the main one, so they can be reused in the final proof. The lemmas were actually not always necessary to achieve termination but greatly improve the performances. For instance, without the additional lemmas the proof of the licensing protocol requires 30 minutes and 414 steps, and the SOC protocol around a day with 47297 steps. For the OTP protocol, without additional lemmas, we could not witness termination after about two days of computation exhausting the computer's memory.

Protocol	Lemmas	Time	Number of steps	Result
AC	0	5s	184 steps	proof
sid-AC	0	82s	32 steps	attack
counter-AC	0	32m	30 steps	attack
AKE	0	1s	7 steps	proof
SOC	4	9s	195 steps	proof
Licensing	2	6s	112 steps	proof
OTP	7	30s	440 steps	proof

Table 1. SUMMARY OF VERIFICATION RESULTS ON CASE STUDIES

5. Conclusion and future work

In this paper we introduced the process calculus SℓAPIC which allows us to reason about applications that make use of report functionalities such as those offered by recent IEEs. We extend the SAPIC/TAMARIN toolchain in order to provide tool support and use SℓAPIC to analyse several

protocols that rely on such reports. The applications include protocols for SOC, secure licensing and a OTP protocol.

As a next step we plan to design composition results for our calculus, in the style of [26]. Such composition results would allow us to analyse complex protocols in a modular way and better scale to large applications: typically, we would want to separately analyse the attested key exchange and use it as a building block for a SOC protocol without the need to analyse the whole system. Another direction for future work would be to study the sealing mechanisms offered by IEEs and integrate such a mechanism in the calculus.

Acknowledgments. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 645865-SPOOC) and the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE). The authors also wish to thank Bogdan Warinschi for the useful discussions in the early stages of this work.

References

[1] *Software Guard Extensions Programming Reference*, Intel, 2014, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[2] *ARM Security Technology - Building a Secure System using TrustZone® Technology*, ARM, 2009, http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[3] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi, "Foundations of hardware-based attested computation and application to SGX," in *Proc. 1st IEEE European Symposium on Security and Privacy (EuroS&P'16)*. IEEE Comp. Soc. Press, 2016, pp. 245–260.

[4] D. Dolev and A. Yao, "On the security of public key protocols," in *Proc. 22nd Symp. on Foundations of Computer Science (FOCS'81)*. IEEE Comp. Soc. Press, 1981, pp. 350–357.

[5] D. A. Basin, C. Cremers, and S. Meier, "Provably repairing the ISO/IEC 9798 standard for entity authentication," *Journal of Computer Security*, vol. 21, no. 6, pp. 817–846, 2013.

[6] 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.

[7] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," *Journal of Computer Security*, vol. 18, no. 6, pp. 1211–1245, Nov. 2016.

[8] 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 Comp. Soc. Press, 2012, pp. 78–94.

[9] S. Costea and B. Warinschi, "Secure software licensing: Models, constructions, and proofs," in *Proc. 29th IEEE Computer Security Foundations Symposium (CSF'16)*. IEEE Comp. Soc. Press, 2016, pp. 78–94.

[10] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proc. 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*. ACM Press, 2013, p. 11.

[11] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: trustworthy data analytics in the cloud using SGX," in *Proc. 36th IEEE Symposium on Security and Privacy (S&P'15)*. IEEE Comp. Soc. Press, 2015, pp. 38–54.

[12] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. K. Rajamani, S. A. Seshia, and K. Vaswani, "A design and verification methodology for secure isolated regions," in *Proc. 37th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'16)*. ACM Press, 2016, pp. 665–681.

[13] M. Patrignani and D. Clarke, "Fully abstract trace semantics for low-level isolation mechanisms," in *Proc. 29th ACM Symposium on Applied Computing (SAC'14)*. ACM Press, 2014, pp. 1562–1569.

[14] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. 36th IEEE Symposium on Security and Privacy (S&P'15)*. IEEE Comp. Soc. Press, 2015, pp. 640–656.

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

[16] C. J. Cremers, "The Scyther Tool: Verification, falsification, and analysis of security protocols," in *Proc. 20th International Conference on Computer Aided Verification (CAV'08)*, ser. Lecture Notes in Computer Science, vol. 5123. Springer, 2008, pp. 414–418.

[17] S. Escobar, C. Meadows, and J. Meseguer, "Maude-NPA: Cryptographic protocol analysis modulo equational properties," in *Foundations of Security Analysis and Design V (FOSAD'09)*, ser. Lecture Notes in Computer Science, vol. 5705. Springer, 2009, pp. 1–50.

[18] 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.

[19] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, "A formal analysis of authentication in the TPM," in *Revised Selected Papers of the 7th International Workshop on Formal Aspects in Security and Trust (FAST'10)*, ser. Lecture Notes in Computer Science, vol. 6561. Springer, 2010, pp. 111–125.

[20] ——, "Formal analysis of protocols based on TPM state registers," in *Proc. 24th IEEE Computer Security Foundations Symposium (CSF'11)*. IEEE Comp. Soc. Press, 2011, pp. 66–82.

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

[22] J. Shao, Y. Qin, D. Feng, and W. Wang, "Formal analysis of enhanced authorization in the TPM 2.0," in *Proc. 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*. ACM Press, 2015, pp. 273–284.

[23] C. Bodei, P. Degano, R. Focardi, and C. Priami, "Primitives for authentication in process algebras," *Theoretical Computer Science*, vol. 283, pp. 271–304, 2002.

[24] D. Hoshina, E. Sumii, and A. Yonezawa, "A typed process calculus for fine-grained resource access control in distributed computation," in *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS'01)*, ser. Lecture Notes in Computer Science, vol. 2215. Springer, 2001, pp. 64–81.

[25] C. Jacomme, S. Kremer, and G. Scerri, "Symbolic models for isolated execution environments," Cryptology ePrint Archive, Report 2017/070, 2017, http://eprint.iacr.org/2017/070.

[26] M. Arapinis, V. Cheval, and S. Delaune, "Composing security protocols: From confidentiality to privacy," in *Proc. 4th International Conference on Principles of Security and Trust (POST'15)*, ser. Lecture Notes in Computer Science, vol. 9036. Springer, 2015, pp. 324–343.

Appendix

We recall below the formal definitions of the syntax and semantics of multiset rewrite rules which serve as the input language of the TAMARIN prover [8].

Definition 17 (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 $l = \mathit{prems}(ri)$ the premises, $a = \mathit{actions}(ri)$ the actions, and $r = \mathit{conclusions}(ri)$ the conclusions of the rule.

Definition 18 (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 and
- r does not contain Fr-facts.

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

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

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

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

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

Definition 20 (Executions)**.** Given a multiset rewriting system R we define its set of executions as

$$\mathit{exec}^{msr}(R) = \Big\{ \emptyset \xrightarrow{A_1}_R \ldots \xrightarrow{A_n}_R S_n \mid \forall a, i, j \colon 0 \leq i \neq j < n.$$

$$(S_{i+1} \setminus^\# S_i) = \{\mathsf{Fr}(a)\} \Rightarrow (S_{j+1} \setminus^\# S_j) \neq \{\mathsf{Fr}(a)\} \Big\}$$

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

$$\mathit{traces}^{msr}(R) = \Big\{ [A_1, \ldots, A_n] \mid \; \forall\, 0 \leq i \leq n.\; A_i \neq \emptyset$$

$$\text{and } \emptyset \stackrel{A_1}{\Longrightarrow}_R \ldots \stackrel{A_n}{\Longrightarrow}_R S_n \in \mathit{exec}^{msr}(R) \Big\}$$

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