Sapic$^+$: protocol verifiers of the world, unite!

Vincent Cheval[1], Charlie Jacomme[2], Steve Kremer[3], and Robert Künnemann[4]

[1]Inria Paris
[2,4]CISPA Helmholtz Center for Information Security
[3]LORIA & Inria Nancy

June 15, 2022

Abstract

Symbolic security protocol verifiers have reached a high degree of automation and maturity. Today, experts can model real-world protocols, but this often requires model-specific encodings and deep insight into the strengths and weaknesses of each of those tools. With Sapic$^+$, we introduce a protocol verification platform that lifts this burden and permits choosing the right tool for the job, at any development stage. We build on the existing compiler from Sapic to Tamarin, and extend it with automated translations from Sapic$^+$ to ProVerif and DeepSec, as well as powerful, protocol-independent optimizations of the existing translation. We prove each part of these translations sound. A user can thus, with a single Sapic$^+$ file, verify reachability and equivalence properties on the specified protocol, either using ProVerif, Tamarin or DeepSec. Moreover, the soundness of the translation allows to directly assume results proven by another tool which allows to exploit the respective strengths of each tool. We demonstrate our approach by analyzing various existing models. This includes a large case study of the 5G authentication protocols, previously analyzed in Tamarin. Encoding this model in Sapic$^+$ we demonstrate the effectiveness of our approach. Moreover, we study four new case studies: the `LAKE` [46] and the `Privacy-Pass` [20] protocols, both under standardization, the `SSH` [47] protocol with the agent-forwarding feature, and the recent `KEMTLS` [45] protocol, a post-quantum version of the main TLS key exchange.

Foreword. This paper is the long version of the version presented at USENIX'22. It is globally more formal and more detailed, especially in Sections 3 to 5. For a first read, we recommend the conference version.

1 Introduction

By leveraging automated reasoning techniques and a symbolic abstraction of cryptographic primitives, protocol verification tools, such as ProVerif and Tamarin, have reached a high degree of maturity in the last decades. Precise models of real-world protocols like TLS 1.3 [25, 15], 5G authentication protocols [11], WPA 2 [26], Noise [35, 30] and Signal [23, 34], show these tools can be used to great effect. However, such case studies tend to only be carried out by experts of the corresponding tool and often at the cost of significant efforts. Further, although multiple tools have very different strength and weaknesses, they have yet to be used in collaboration.

A closer look at the development workflow reveals why. First, even specification languages that are very similar on the surface, e.g., the Sapic front end [37] to Tamarin [44] and ProVerif's applied π calculus [16] require different modeling regimes, e.g., when storing or receiving values, setting locks or parsing network messages. In practice, this makes it very hard to switch between tools, even if they share many syntactic elements.

Second, deciding which tool to target in the first place is even harder. Powerful abstractions deployed by ProVerif generally offer better automation and speed than Tamarin, but may lead in some cases to false attacks or hinder attack reconstruction. In cases where neither a proof nor an attack is found expert knowledge, with a good understanding of ProVerif's internals, is required to guide the proof. Tamarin on the other hand guarantees correctness, hence, no false attacks, in case of termination. Tamarin also provides a GUI to interactively guide the proof search when automated search does not terminate. Experts may also write so-called oracles to automate such guidance. Another aspect that may guide the choice of the tool is the support for cryptographic primitives. Both tools provide the possibility for user-defined primitives, with slightly different, incomparable scopes. In particular, Tamarin has builtin support for several theories that feature an associative, commutative operator and therefore offers a more precise for modeling Diffie-Hellman exponentiation and exclusive or (xor). Given all of these parameters, the choice of the right tool requires not only insight into the protocol, but insight into the modeling strategy.

Third, these requirements change during development. As protocol modeling is an incremental process, it may for instance be desirable to use a simplified Diffie-Hellman theory in the early stages. Throughout the development, it is vital to perform sanity checks, i.e., to find honest traces that show the protocol can execute and thus spot bugs in the model before starting time-consuming proofs. While ProVerif is often faster in finding proofs, it can sometimes struggle to show the existence of 'good' traces. Given the radically different reasoning techniques underlying these two tools it is hard to predict which one will perform better on a given protocol. On a same protocol, one tool may even be capable of proving one property, but not another, while the other tool has the reverse capabilities.

In this work, we aim at exploiting the strengths of each of the tools. Therefore, we design a common input language, called Sapic$^+$, which may be used as an input for several tools: Tamarin, ProVerif, GSVerif [21], and DeepSec [22]. Sapic$^+$ is an extension of Sapic (which only translated to Tamarin and did not support equivalence-based properties) that can be automatically translated to each of these tools, thus removing the need to choose and making these tools more accessible. While current protocol models only target one of these tools, a common platform encourages the implementation of tool-specific encodings in a model-agnostic manner, which promises consistent verification results even for users who are oblivious to the underlying reasoning technique which is necessary for wide adoption. We believe that this approach has a number of advantages. (1) The distinctive features of each tool are available for the same model while sidestepping many encoding pitfalls (such as subtle differences in the semantics for similar syntactic constructs) that one would encounter when porting manually an existing model to another tool. Moreover, even though we generate the specification for each particular tool, we maintain the possibility for experts to guide the tools. We illustrate this fact on the 5G case study [11]: we are able to use oracles to provide termination for Tamarin. Moreover, unlike Sapic, we support the verification of equivalence properties through ProVerif, and above all DeepSec, a tool that specializes in these properties. (2) An essential part of our work is that we *formally prove* the correctness of the translations, which enables a workflow that can take advantage of each of these tool's strengths throughout model development on the same input file. Lemmas proven in one verifier can be used as assumptions in the other, potentially increasing the scope of verification. Again, on the 5G case study, we show that several attacks can be recovered much faster using ProVerif. On the other hand, we are able to use Tamarin to prove *axioms* assumed by ProVerif, removing the need to prove correctness of these lemmas by hand on some examples. (3) Similar to Isabelle's Sledgehammer, multiple verifiers can be run in parallel, terminating when any of them does. This is especially useful when it is not clear if security is provable or an attack can be found, or when the user is unsure which verifier is most suited for the lemma at hand. (4) Finally, for researchers, Sapic$^+$ provides a convenient target for meta-theories and encodings. Existing results (e.g., distance-bounding [41] or human errors [14]) could cover a wide set of tools rather than being developed for a single tool. Source-to-source encodings in applied-π-like calculi (e.g., [33, 38, 34]) translate with relatively little work.

Contributions We can summarize our contributions as follows.

1. **Translations:** we provide *provably correct* translations from Sapic$^+$ to both ProVerif (supporting stateful reasoning by using ProVerif's GSVerif frontend) and DeepSec and extend the existing translation to Tamarin. Our translations cover both protocol and property specification. In addition, unlike Sapic, we support privacy-type properties expressed as process equivalences in DeepSec (verifying trace equivalence) and ProVerif (verifying diff equivalence, which implies trace equivalence). The translations' correctness proof guarantees that results from one tool carry over to any of the other tools as explained above. Our proofs also relate ProVerif's and Tamarin's security property languages in terms of expressivity, showing subtle differences that even experts of one of the tools may not be aware of.

2. **Protocol-platform:** we automate these translations by integrating Sapic$^+$ (including [37]) directly into Tamarin to improve visual feedback in manual, interactive proofs, and include a(n optional) type system, similar to that of ProVerif, to catch modeling errors in development. This allows to easily use the different tools from a single input file and exploit the strengths of each of the tools, avoiding the time-consuming and potentially error-prone process of carrying over models.

3. **Extensions and optimizations of Sapic:** we extend the scope and efficiency compared to Sapic [37]. We encode support for destructor symbols, let-bindings and macro declarations in Sapic's translation procedure to Tamarin. We also introduce a *compression technique* to reduce the size of the models produced by the translation, prove its correctness and demonstrate its efficiency. Our case studies show that this significantly increases the scope of Sapic$^+$, enabling the verification of larger and more realistic protocol models.

4. **Case studies:** we evaluate the new development workflow on four entirely new protocol models: `KEMTLS`, `Privacy-Pass`, `LAKE` and `SSH` with agent forwarding. We also demonstrate that existing case studies would have benefited from being directly analyzed in Sapic$^+$ without loss of efficiency. In particular, we ported an existing Tamarin model of the complex 5G authentication protocols case study [11] to Sapic$^+$: we observe that the dedicated, handwritten oracles used to automate the proofs in Tamarin carried over straightforwardly, and verification time was preserved, showing the efficiency of the generated model. Moreover, using ProVerif, with a less precise, but attack preserving modeling of xor, we detected the existing attacks in a completely automated way, and much faster.

Related Work The AVISPA [5] and, its successor, AVANTSSAR projects [4] pioneered the protocol platform approach. They provide a front end for modeling protocols that allows to use different tools to verify security properties. However, the expressiveness and efficiency of these back-ends lack in comparison to the more recent Tamarin and ProVerif. Also, contrary to our work, the translations were not formally proven correct, limiting the interaction between the backends.

Other works have taken a different approach by developing front ends for a single tool to make it more expressive (e.g., [43, 3, 13]). Similarly, many other independent tools enable automatic protocol verification (see [9] for an almost extensive presentation), but we believe that they have not yet reached Tamarin's and ProVerif's level of maturity and adoption by the community.

Outline We provide some general background about the symbolic model and the multiple tools we rely on in Section 2. We then define the Sapic$^+$ language in Section 3, before describing the translations and their optimizations for Tamarin in Section 4 and ProVerif in Section 5. We finally showcase the tool and our case studies in Section 6. An understanding of the more formal definitions of Section 3 is neither needed to understand at a high-level the remainder of the paper nor to use the tool.

2 Background

In this section, we provide background on symbolic methods for the verification of cryptographic protocols. We first explain how messages and cryptographic primitives are represented using a *term algebra*. We then review the different verification tools that are relevant for this work.

2.1 Modeling messages as terms

Terms and substitutions Messages sent in a protocol execution are modeled as *terms*. Fresh values are modeled by constants from an infinite set of names $\mathcal{N}$, divided into public (attacker) names $\mathcal{N}_{\texttt{pub}}$ and secret (protocol) names $\mathcal{N}_{\texttt{priv}}$. We also assume a set of variables $\mathcal{X}$. Terms are then built over names in $\mathcal{N}$, variables in $\mathcal{X}$ and applications of a function symbols in $\mathcal{F}$ on terms. We consider two kinds of function symbols, i.e., $\mathcal{F} = \mathcal{F}_c \uplus \mathcal{F}_d$, for *constructor* and *destructor* function symbols. Consider for example the term $enc(m, k)$ modeling the encryption of another term m with the secret key k: encryption is modeled by the constructor function $enc \in \mathcal{F}_c$, $k \in \mathcal{N}_{\texttt{priv}}$ is a secret name and m is another term.

A substitution σ is a function from variables to terms. We lift the application of substitutions from variables to terms and use postfix notation, i.e., we write $t\sigma$ for the term in which we replace each variable x occurring in the domain of σ and in t by $\sigma(x)$.

Equational theories and rewrite systems Properties of constructor terms, i.e., terms containing no destructor symbol, are expressed by an *equational theory* E, which is defined by a set of equations $(t_1 = t_2)$ on name-free constructor terms. This induces a relation $=_E$ on constructor terms, defined as the smallest equivalence relation $=_E$ that contains all $t_1 = t_2 \in E$, is closed under substitution of variables for constructor terms, and application of function symbols. For example, $dec(enc(x, y), y) = x$ models functional correctness of an encryption scheme. An equational theory for DH exponentiation could include (among others) the equations $\exp(\exp(g, a), b) = \exp(g, a \cdot b)$ and $a \cdot b = b \cdot a$.

The semantics of destructor symbols is defined through a set of *rewrite rules* of the form $d(t_1, \ldots, t_n) \to r$ where $d \in \mathcal{F}_d$ of arity n and the t_is are name-free constructor terms. Terms are rewritten *bottom-up* modulo the equational theory E to ensure that destructors are only applied to constructor terms. We require that the resulting rewrite system is convergent modulo E, meaning that (roughly speaking) any term t has a normal form modulo E, which we denote by $t{\downarrow}$. For example, assuming that $dec \in \mathcal{F}_d$ is a destructor, we can define the rule $dec(enc(x, y), y) \to x$. Then, assuming the equational theory for DH from above, $dec(enc(m, g^{a \cdot b}), g^{b \cdot a})$ rewrites into m, as $g^{a \cdot b} =_E g^{b \cdot a}$. When a term t contains a destructor symbol that does not rewrite, we say that the destructor *fails*, and we write $t \to_E \mathsf{fail}$. Note that the decryption of an invalid ciphertext (here modeled as a random value n) $dec(n, k)$ would fail. Finally, we lift $=_E$ to arbitrary terms and define $t_1 =_E t_2$ to hold if $t_1 \not\to_E \mathsf{fail} \wedge t_2 \not\to_E \mathsf{fail} \wedge t_1{\downarrow} =_E t_2{\downarrow}$.

Patterns We define the set of *patterns* $\mathrm{Pat} \subseteq \mathcal{T}(\mathcal{F}_c, \mathcal{N}, \mathcal{X})$ to be a subset of terms. Patterns define the terms against which we can perform pattern matching in let constructs and protocol inputs. Typically, TAMARIN allows matching against arbitrary terms. PROVERIF, on the other hand, only allows *executable patterns*, that is patterns where the variables to be bound can only directly occur under *data* constructors. A data constructor is a function for which there exist destructors that allow to access each of its arguments. Tuples are a typical example. We denote the set of executable patterns $\mathrm{Pat}_{ex}()$. In the DEEPSEC tool, the only data functions are built-in tuples.

2.2 The Tamarin prover

Protocol specification In TAMARIN protocols are described using multiset rewrite (MSR) rules of the form

$$[lhs] \ \text{--}[\ actions\ ]\!\!\to [rhs].$$

They describe the manipulation of the protocol state represented as a multiset of *facts*. Intuitively, for such a rule to fire, we require the (multiset of) facts on the left-hand side *lhs*. Then these facts are deleted, and the facts on the right-hand side *rhs* are added. The *actions* are facts that *annotate* this rule, and will be used to specify properties. As an example, consider the following 3 rules.

$$\begin{aligned}
\mathtt{R_0} &: [\mathtt{Fr}(\mathtt{lk})] \;-\![]\!\!\rightarrow [!\mathtt{SP}(\mathtt{lk}), !\mathtt{SQ}(\mathtt{lk})] \\
\mathtt{R_P} &: [\mathtt{SP}(\mathtt{lk}), \mathtt{Fr}(\mathtt{k})] \;-\![\ \mathsf{Honest}(\mathtt{k})\ ]\!\!\rightarrow \\
&\qquad\qquad\qquad\qquad [\mathtt{Out}(\mathtt{enc}(\langle \mathtt{key}, '\mathtt{hs}' \rangle, \mathtt{lk}))] \\
\mathtt{R_Q} &: [\mathtt{SQ}(\mathtt{lk}), \mathtt{In}(\mathtt{enc}(\langle \mathtt{k}, '\mathtt{hs}' \rangle, \mathtt{lk}))] \;-\![\ \mathtt{Accept}(\mathtt{k})\ ]\!\!\rightarrow []
\end{aligned}$$

In rule $\mathtt{R_0}$ we require a fresh long-term key $\mathtt{lk}$, shared between P and Q, and move these two parties into the states $\mathsf{SP}(\mathtt{lk})$ and $\mathsf{SQ}(\mathtt{lk})$, respectively. Whenever P is in state SP and we have a fresh session key k, then P may output the term $\mathtt{enc}(\langle \mathtt{k}, '\mathtt{hs}' \rangle, \mathtt{lk})$ where $'\mathtt{hs}'$ is a constant. During this transition, the key $\mathtt{k}$ is tagged as an honest key. Q on the other hand, when in state $\mathtt{SQ}(\mathtt{lk})$ and with an available input matching $\mathtt{enc}(\langle \mathtt{k}, '\mathtt{hs}' \rangle, \mathtt{lk})$, can accept key k. The facts $\mathtt{Fr}(\cdot), \mathsf{In}(\cdot)$ and $\mathsf{Out}(\cdot)$ are built-in facts. The fresh fact $\mathtt{Fr}(\cdot)$ is always available, but guarantees that its argument is instantiated with a *fresh* name, i.e., a name that did not appear elsewhere, modeling the generation of a random key. $\mathtt{In}(\cdot)$ and $\mathtt{Out}(\cdot)$ facts provide the network interface. As we assume that the network is controlled by the attacker, creating a fact $\mathtt{Out}(\mathtt{t})$ adds $\mathtt{t}$ to the attacker knowledge; conversely, $\mathtt{In}(\mathtt{t})$ requires the attacker to construct a term that matches $\mathtt{t}$. When we generated the facts $\mathsf{SP}(\mathtt{lk})$ and $\mathsf{SQ}(\mathtt{lk})$ we preceded them by '!', marking them as permanent. They are available an unbounded number of times, modeling that P and Q can execute an arbitrary number of sessions with the long-term key $\mathtt{lt}$, generating a fresh session key $\mathtt{k}$ each time.

Equational theories In Tamarin, cryptographic primitives can be specified by an arbitrary equational theory that is convergent and has the finite variant property. Introducing these notions goes beyond the scope of this paper and we refer the reader to [44, 28]. In addition, Tamarin offers built-in support for several other operations, including DH exponentiation and exclusive or. Tamarin allows to pattern match any term in its inputs, but does not support destructors.

Property specification Tamarin uses a temporal logic to express security properties about the possible protocol executions, which are modeled as traces. Traces are simply the sequence of actions triggered by MSR rules. We can write lemmas that must be valid on all traces. For instance, to express in our previous example that, whenever Q accepts a key $\mathtt{k}$, $\mathtt{k}$ must have been honestly generated by P. This can be written as:

```
All k #i.Accept(k)@i ⟹ Ex #j.Honest(k)@j & j < i
```

More precisely, we require that if an `Accept` event was raised for `k` at any time point `i` of the trace, then the `Honest` event must have been raised for the same value `k` at a previous time point `j`. We can also write *restrictions*, which constrain the set of traces considered when proving security properties. In our example, it could be used to only focus on traces with at most one successful session of Q by writing:

```
All k l #i #j.Accept(k)@i & Accept(l)@j ⟹ #i=#j
```

Automation To automatically find proofs, Tamarin tries to refute the existence of a counter-example by negating the property and exploring in a backward search all paths that would lead to this negated formula. The reasoning is done symbolically, based on a dedicated constraint solving algorithm. If the tool can prove that no such path exists, the property is valid. The proof search may, however, not terminate. In such cases, the user can guide the backward search in interactive mode, or (requiring more advanced knowledge) specify tailored heuristics through so-called oracles.

```
 1 let P('lk, k) =
 2     event Honest(k);
 3     out(c, enc(<k,'hs'>,'lk))
 4
 5 let Q('lk) =
 6    in(c, cipher);
 7    let <key,'hs'>=dec(cipher,sk) in
 8      event Accept(key);
 9      out(c, 'accept')
10    else
11      out(c, 'abort')
12
13 !new 'lk; (!new k; P('lk,k) | !Q('lk))
```

Figure 1: Protocol example in the applied π calculus

2.3 ProVerif

Protocol specification In ProVerif protocols are expressed in a dialect of the applied π calculus. As an example consider the protocol described in Figure 1, similar to the example given for Tamarin. We define two agents P and Q, parameterized by a long-term secret key `'lk` and for P a fresh session key `k` as well. The main process is described on line 13: the `!` operator allows to spawn an unbound number of sessions, and in each session we sample a fresh value `k` thanks to the `new` instruction. On line 3, P outputs (on channel c) an encryption with `'lk` of the key on the network. `!` may input a ciphertext, and check if decryption succeeds and whether the plaintext is of the expected form `<k,'hs'>`. If so, it raises a success event and sends a success message. Else, it outputs an error message.

Equational theories As for Tamarin, cryptographic primitives can be specified using arbitrary convergent equational theories that have the finite variant, but in ProVerif, no associative-commutative symbols are allowed, such as required for DH exponentiation or exclusive. However, ProVerif additionally allows for so-called linear equations (where each variable appears at most once on the left-hand and on the right-hand side). This allows to approximate DH exponentiation by the equation $(g^x)^y = (g^y)^x$ for a constant g. Moreover, ProVerif supports the definition of destructor symbols, hence allowing a convenient way to model that some function applications may fail.

Property specification Properties on traces can be expressed by means of (injective) correspondence queries. E.g.,

```
query  event(Accept(x)  ⟹  event(Honest(x))
```

expresses that whenever (an instance of) the event `Accept(x)` executes, the event `Honest(x)` must have been executed before (with the same value for `x`). If the query is specified to be *injective* we require that each `Accept(x)` can be matched by a distinct `Honest(x)`.

Moreover, ProVerif offers support for properties expressed in terms of indistinguishability, i.e., by specifying that two protocols cannot be distinguished by an attacker. Continuing our example, we could specify *strong secrecy* of `k` as a non-interference property, encoded as an equivalence. ProVerif can then be used to prove this equivalence:

```
let scen1 = in(c, <k1,k2>); !new sk; (P(sk, k1)|Q(sk)).
let scen2 = in(c, <k1,k2>); !new sk; (P(sk, k2)|Q(sk)).
equivalence scen1 scen2
```

Intuitively, even if the attacker chooses two session keys `k1` and `k2`, it is unable to distinguish which one was used. Similarly, we could model unlinkability: can the attacker distinguish the case where all sessions of `P` and `Q` have the same shared secret key from the scenario where each pair of sessions has a distinct key?

```
let scen1 =  new sk; (!new key; P(sk,key) | !Q(sk)).
let scen2 = !new sk; (!new key; P(sk,key) | !Q(sk)).
equivalence scen1 scen2.
```

This property does, in fact, not hold. Moreover, ProVerif is unable to conclude, returning `cannot be proved`. This comes in particular from the fact that ProVerif tries to prove a very strong form of equivalence, dubbed *diff-equivalence* which is not satisfied on this example.

Automation Internally, ProVerif translates the protocol specification into a particular form of first-order logic formulas, called Horn clauses. It then uses a resolution algorithm that simplifies these clauses in such a way that the security property can be verified on this simplified form. The verification in ProVerif is generally much faster than in Tamarin. This is in particular due to the fact that the translation from the applied π calculus into Horn clauses introduces (sound) abstractions: in addition to nontermination the tool may sometimes fail and find neither a proof nor an attack. This problem occurs in particular when protocols maintain a global, mutable state, i.e., different protocol sessions update this state. Recently, the GSVerif front end significantly improved on this limitation building on the following simple, but effective idea: rather than verifying the formula ϕ, GSVerif verifies whether '*ϕ or some action that should occur only once occurred twice*' by automatically adding protocol annotations and transforming queries accordingly.

2.4 DeepSec

DeepSec specializes in indistinguishability properties, notably *trace equivalence*. Protocols are described in basically the same language as ProVerif, but without replication, hence limiting verification to a bounded number of sessions. On the flip side, DeepSec provides a decision procedure, ensuring termination (in theory—in practice the tool may run out of resources on large instances). Therefore, when ProVerif does not terminate or is inconclusive, we can use DeepSec, but we must bound the number of replications. Continuing our example, this would correspond to:

```
let scen1 = new sk; !^3(new key; P(sk,key) | Q(sk)).
let scen2 = !^3(new sk;(new key; P(sk,key) | Q(sk))).
query trace_equiv(scen1,scen2).
```

where we check the equivalence for three sessions. DeepSec is indeed able to reconstruct an attack trace. Note that ProVerif may sometimes be unable to prove the equivalence even when the processes are equivalent, as it verifies a stricter form of equivalence than DeepSec. This happens for instance on the Basic Access Control (BAC) protocol implemented in the electronic passport [29].

Regarding equational theories, DeepSec provides support for a class of subterm convergent destructor theories, a class strictly included in those of ProVerif.

2.5 Sapic

In Sapic, protocols are described in a stateful dialect of the applied π calculus, including constructs for manipulating global, mutable state and for acquiring *locks* to manipulate this state concurrently. These processes are translated into Tamarin. The property specification language is exactly that of Tamarin. As a result, Sapic inherits the strengths of Tamarin, but also its limitations. In particular, Sapic does neither support destructors nor equivalence properties. Moreover, the generated MSR rules sometimes add a performance overhead.

3 The Sapic$^+$ language

We describe here in more detail the Sapic$^+$ language (which builds on Sapic [36, 37]). The added features include the possibility to define destructors, i.e., function symbols whose application may fail (see below), let bindings with pattern matching and else branches and a slightly modified semantics suitable for expressing equivalence properties.

Before defining SAPIC$^+$'s language more formally, we give a short overview of its main features.

3.1 Overview

Protocol specification SAPIC$^+$'s syntax for specifying protocols is an applied π calculus similar to PROVERIF and SAPIC; Fig. 1 is actually a valid SAPIC$^+$ example. Compared to SAPIC, the added features include the definition of destructors, i.e., function symbols whose application may fail (see below), let bindings with pattern matching and else branches.

Equational theories Through its exports, SAPIC$^+$ supports the union of the theories supported by TAMARIN and PROVERIF. When exporting a theory like DH exponentation to PROVERIF (which only has partial support for this theory), we export an abstraction of the theory, that can still be useful.

Property specification SAPIC$^+$ supports reachability properties expressed in the same logic as SAPIC and TAMARIN, but additionally translates and exports them to PROVERIF's query language. Further, SAPIC$^+$ supports equivalence properties that can be expressed similarly to PROVERIF and DEEPSEC.

Automation SAPIC$^+$ automatically translates to PROVERIF, TAMARIN and DEEPSEC, and thus benefits from the automation of each of those backends. In contrast to SAPIC, which was a stand-alone preprocessor, SAPIC$^+$ is integrated into TAMARIN. This improves user interaction, as now SAPIC$^+$-generated rules display as process actions in interactive proofs and error reporting is more precise. Moreover, it aids future development, as the SAPIC$^+$ module interfaces directly with TAMARIN's term theory and parser, benefiting from new developments. Most importantly, we significantly improved on SAPIC's level of automation, first through several (provably correct) optimizations for the translation to TAMARIN and second with the addition of translations to PROVERIF and DEEPSEC. Both improvements are illustrated by our case-studies, notably 5G-AKA.

Extra features SAPIC$^+$ naturally inherits previous extensions to SAPIC (e.g., a local-progress semantics, needed to show liveness-like properties [6]). Direct encodings in SAPIC (e.g., SGX reports [33] or accountability [38]) translate *for free*, and become available in PROVERIF for the first time.

3.2 Modeling messages as terms

Terms and substitutions. We model the messages sent in a protocol execution as *terms*. Terms are built using a set of function symbols $\mathcal{F}$, each with an arity n. Moreover, we suppose that $\mathcal{F} = \mathcal{F}_c \uplus \mathcal{F}_d$ for constructor and destructor functions. Fresh values are modeled by constants from an infinite set of names $\mathcal{N}$, divided into public (attacker) names $\mathcal{N}_{\mathtt{pub}}$ and secrets (protocol) names $\mathcal{N}_{\mathtt{priv}}$. We also assume a set of variables $\mathcal{X}$. Given $\mathsf{F} \subseteq \mathcal{F}$, $\mathsf{N} \subseteq \mathcal{N}$ and $\mathsf{X} \subseteq \mathcal{X}$ we denote by $\mathcal{T}(\mathsf{F}, \mathsf{N}, \mathsf{X})$ the set of terms built over functions in F, names in N and variables in X.

A substitution $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{N}, \mathcal{X})$ is a function from variables to terms. We homomorphically lift the application of substitutions from variables to terms and use postfix notation, i.e., we write $t\sigma$ for the term in which we replace each variable x occurring in the domain of σ and in t by $\sigma(x)$.

Equational theories and rewrite systems Properties of constructor terms, i.e., terms containing no destructor symbol, are expressed by an *equational theory* E, defined by a set of equations $(t_1 = t_2)$ such that $t_1, t_2 \in \mathcal{T}(\mathcal{F}_c, \emptyset, \mathcal{X})$. This induces a relation $=_E$ on constructor terms, defined as the smallest equivalence relation $=_E$ that contains all $t_1 = t_2 \in E$, is closed under substitution of variables for constructor terms, and application of function symbols. An equational theory for DH exponentiation could include (among others) the equations $\exp(\exp(g, a), b) = \exp(g, a \cdot b)$ and $a \cdot b = b \cdot a$.

The semantics of destructor symbols is defined through a set of *rewrite rules* E_D of the form $d(t_1, \ldots, t_n) \to r$ where $d \in \mathcal{F}_d$ of arity n and $t_i \in \mathcal{T}(\mathcal{F}_c, \emptyset, \mathcal{X})$. As usual, we denote that t_1 rewrites to t_2 by $t_1 \to t_2$ and write $t \to_E u$ when t contains a single destructor d and $t = d(t_1, \ldots, t_n)$ rewrites to u modulo E, i.e., $t \to_E u$ iff $d(v_1, \ldots, v_n) \to v$ for some constructor terms $v_1 =_E t_1, \ldots, v_n =_E t_n$ and $v =_E u$. Terms with several destructors are rewritten *bottom-up* to ensure that we only rewrite destructors applied to constructor terms. We require that the resulting rewrite system is convergent modulo the equational theory E and denote by $t{\downarrow}$ the normal form of t (up to E). When a term contains a destructor symbol that does not rewrite, we say that the destructor *fails*. We suppose that protocols only send valid messages which we capture using the Msg predicate: we define $\mathsf{Msg}(t)$ to hold if for any subterm u of t we have that $u{\downarrow} \in \mathcal{T}(\mathcal{F}_c, \mathcal{N}, \mathcal{X})$. For example, a symmetric encryption scheme that provides ciphertext integrity could be modeled by the rule $dec(enc(m, k), k) \to m$. Note that the decryption of an invalid ciphertext (here modeled as a random value n) $dec(n, k)$ would fail.

Finally, we lift $=_E$ to arbitrary terms and define $t_1 =_E t_2$ to hold if $\mathsf{Msg}(t_1) \wedge \mathsf{Msg}(t_2) \wedge t_1{\downarrow} =_E t_2{\downarrow}$.

Patterns Finally we define the set of *patterns* $\mathrm{Pat} \subseteq \mathcal{T}(\mathcal{F}_c, \mathcal{N}, \mathcal{X})$ to be a subset of terms. Patterns define the terms against which we can perform pattern matching in let constructs. Typically, Tamarin allows matching against arbitrary terms. ProVerif, on the other hand, only allows *executable patterns*, that is patterns where the variables to be bound can only directly occur under *data* constructors. A data constructor is a function f of arity n for which there exist n destructors d_i defined by $d_i(f(x_1, \ldots, x_n)) \to x_i$ for $1 \leq i \leq n$, i.e., data constructors allow to access each of its arguments. A typical example are tuples. Denoting the set of data constructors by $\mathcal{F}_{\mathsf{data}}$, we denote the set of executable patterns $\mathrm{Pat}_{ex}(\mathcal{F}_{\mathsf{data}})$. In the DeepSec tool the only data functions are built-in tuples.

Conditional formulae Following [37], conditional branchings in protocols are described by first-order formulae over equalities on terms of the form $t_1 = t_2$, possibly containing variable quantifiers. The semantics of formulae is as usual where we interpret $=$ as $=_E$.

Facts We define a set of fact symbols $\mathcal{F}_f$ each with an arity over which we construct the set of facts Facts by applying fact symbols on constructor terms. These facts will be used to annotate processes and log events. In particular, $\mathcal{F}_f$ will contain the facts $K, \mathsf{In}, \mathsf{Out}$ to log attacker knowledge, input and output facts.

3.3 Protocols

3.3.1 Syntax

We present the syntax of Sapic$^+$ in Fig. 2 where t, t_i range over arbitrary terms in $\mathcal{T}(\mathcal{F}, \mathcal{N}, \mathcal{X})$, n ranges over private names in $\mathcal{N}_{\mathtt{priv}}$, x ranges over variables in $\mathcal{X}$ and p ranges over patterns in Pat. *Elementary processes* model finite protocols that simply sample fresh values with the new construct, perform input and output, based on some control flow. *Extended processes* can model unbounded protocols, that can also raise events that provide a convenient way to model reachability properties. Finally, we also define *stateful processes* that can manipulate global shared states.

3.3.2 Semantics

We equip Sapic$^+$ with operational semantics that will be suitable for expressing both equivalence and reachability properties. A *configuration* $(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$ defines the set of currently bound

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

(elementary processes)		*(extended processes)*	
0	*null process*	$\mid$ $\mathsf{event}\ F;\ P$	*event*
$\mid$ $\mathsf{new}\ n;\ P$	*binding of a fresh name*	$\mid$ $!P$	*replication*
$\mid$ $P \mid Q$	*parallel composition*	*(stateful processes)*	
$\mid$ $\mathsf{out}(t_1, t_2);\ P$	*output of t_2 on channel t_1*	$\mid$ $\mathsf{insert}\ t_1, t_2;\ P$	*set state t_1 to t_2*
$\mid$ $\mathsf{in}(t, x);\ P$	*input on channel t a term bound to x*	$\mid$ $\mathsf{delete}\ t;\ P$	*delete state t*
$\mid$ $\mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ Q$	*conditional*	$\mid$ $\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P\ \mathsf{else}\ Q$	*read state t*
$\mid$ $\mathsf{let}\ p = t\ \mathsf{in}\ P\ \mathsf{else}\ Q$	*let binding*	$\mid$ $\mathsf{lock}\ t;\ P$	*lock a state*
		$\mid$ $\mathsf{unlock}\ t;\ P$	*unlock a state*

Figure 2: Syntax of our process calculus

$$
\begin{array}{rcl}
(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) & \xrightarrow{K(t)} & (\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) \\
& & \quad \text{if } t =_E R\sigma \text{ for some } R \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{\mathsf{pub}}, \mathcal{AX}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{out}(t_1, t_2); P\}, \sigma, \mathcal{L}) & \xrightarrow{\mathsf{Out}(R), K(t_1)} & (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\}, \sigma \cup \{\mathtt{att}_n \mapsto t_2\}, \mathcal{L}) \\
& & \quad \text{if } t_1 =_E R\sigma \text{ for some } R \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{\mathsf{pub}}, \mathcal{AX}), \mathsf{Msg}(t_2) \text{ and } n = |\sigma| + 1 \\
(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{in}(t, x); P\}, \sigma, \mathcal{L}) & \xrightarrow{\mathsf{In}(R, R'), K(\langle t, R'\sigma \rangle)} & (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\{x \mapsto R'\sigma\}\}, \sigma, \mathcal{L}) \\
& & \quad \text{if } t =_E R\sigma, \mathsf{Msg}(R'\sigma) \text{ for some } R, R' \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{\mathsf{pub}}, \mathcal{AX}) \\
(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{let}\ p = t\ \mathsf{in}\ P\ \mathsf{else}\ Q\}, \sigma, \mathcal{L}) & \rightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\tau\}, \sigma, \mathcal{L}) \\
& & \quad \text{if } p\tau =_E t \text{ and } \tau \text{ is grounding for } p \\
(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{let}\ p = t\ \mathsf{in}\ P\ \mathsf{else}\ Q\}, \sigma, \mathcal{L}) & \rightarrow & (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{Q\}, \sigma, \mathcal{L}) \\
& & \quad \text{if for all } \tau, p\tau \neq_E t
\end{array}
$$

Figure 3: Operational semantics (excerpt)

names $\mathcal{E}$, the current values of the states $\mathcal{S}$, the set of executable processes $\mathcal{P}$, the current attacker knowledge inside the substitution σ, and the active locks $\mathcal{L}$. We assume an infinite set of variables $\mathcal{AX} = \{\mathtt{att}_i\}_{i \in \mathbb{N}}$, that will be mapped to the attacker knowledge, and σ is thus mapping variables in $\mathcal{AX}$ to the sequence of outputs of the protocol.

The semantics is a labeled transition system between configurations, partially defined through its transition relation $\xrightarrow{l}$ (where l may be empty, or otherwise is a set of facts) in Fig. 3 (see Fig. 9 in App. G for the full version).

The labels contain both the terms corresponding to the values sent over the network ($K(\langle t_1, t_2 \rangle)$), but also the *recipes* used by the attacker to compute the protocol inputs ($\mathsf{In}(R, R')$), as well as the recipes used by the attacker to show that it can compute the channel of an output ($\mathsf{Out}(R)$). This allows us to define both reachability and equivalence based properties.

We define $\mathcal{C}_i \xRightarrow{l_1 \cdot \dots \cdot l_n} \mathcal{C}_f$ to hold when $\mathcal{C}_i \rightarrow^* \xrightarrow{l_1} \rightarrow^* \dots \rightarrow^* \xrightarrow{l_n} \rightarrow^* \mathcal{C}_f$ and, for ease of reading, we write $P \xRightarrow{l_1 \cdot \dots \cdot l_n} \mathcal{C}$ instead of $(\emptyset, \emptyset, \{P\}, \emptyset, \emptyset) \xRightarrow{l_1 \cdot \dots \cdot l_n} \mathcal{C}$. Such a sequence of labels, together with the attacker knowledge σ of the resulting configuration defines a *trace* and denote the set of traces of a process P as

$$traces^{pi}(P) = \{(tr, \sigma) \mid \exists \mathcal{E}, \mathcal{S}, \mathcal{P}, \mathcal{L}.\ P \xRightarrow{tr} (\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})\}$$

Moreover, we denote by $\xRightarrow{tr}_{\vdash}$ the relation where In and Out facts are removed from the labels, and $\xRightarrow{tr}_{\approx}$ the relation where we only keep the labels that are In and Out facts. Those relations yield the corresponding traces set $traces^{pi}_{\vdash}(P)$ and $traces^{pi}_{\approx}(P)$. When the attacker knowledge is not important, by abuse of notation, we sometimes refer to a trace simply as tr instead of (tr, σ).

3.4 Security properties

3.4.1 Reachability properties

To express reachability properties, we use the same temporal logic as the one of Tamarin [44]. In the Tamarin tool, security properties are described in an expressive two-sorted first-order logic: we distinguish the sort *msg* that ranges over terms and the sort *temp* used for time points.

Definition 1 (Trace formulas)**.** *An atomic trace formula is either false* $\bot$*, a term equality* $t_1 \approx t_2$*, a time point ordering* $i \lessdot j$*, a time point equality* $i \doteq j$*, or an action* $F@i$ *for a fact* $F \in \mathrm{Facts}$ *and a time point* i*. A trace formula is a first-order formula over atomic trace formulas.*

To define the semantics, let each sort s have a domain $\mathsf{dom}(s)$: $\mathsf{dom}(temp) = \mathcal{Q}$ and $\mathsf{dom}(msg) = \mathcal{T}(\mathcal{F}, \mathcal{N})$. Moreover, we denote by $\mathcal{X}_{temp}$ the set of temporal variables and $\mathcal{X}_{msg}$ the set of message variables. A function $\theta : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{N}) \cup \mathcal{Q}$ is a valuation if it respects sorts, i.e., $\theta(\mathcal{X}_s) \subset \mathsf{dom}(s)$ for all sorts s. If t is a term, $t\theta$ is the application of the homomorphic extension of θ to t. Moreover, given a trace tr, we denote by $idx(tr)$ the set of positions in the trace and by tr_i the multiset of facts of tr at position i.

Definition 2 (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:*

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

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

For a set of traces Tr we also write $Tr \vDash \varphi$ if φ holds on any trace in Tr and any valuation, and even $P \vDash \varphi$ when $traces^{pi}_{\vdash}(P) \vDash \varphi$. Actually Sapic$^+$ (and Tamarin) also allows checking satisfiability, i.e., the existence of a satisfying trace and valuation, denoted $\vDash^{\exists}$: this is the dual notion and $Tr \vDash \varphi$ iff $Tr \nvDash^{\exists} \neg\varphi$ and sometimes $\vDash^{\forall}$ is used for $\vDash$ to emphasize that we refer to validity.

3.4.2 Equivalence properties

Some properties are defined as the indistinguishability by an attacker of two processes. Indistinguishability can be modeled by trace equivalence. To define trace equivalence we first defined the notion of *static equivalence* which models the indistinguishability of two sequences of terms (represented as substitutions).

Definition 3. *Two substitutions* $\sigma_1, \sigma_2 : \mathcal{AX} \to \mathcal{T}(\mathcal{F}_c, \mathcal{N}, \emptyset)$ *are statically equivalent, written* $\sigma_1 \sim_E \sigma_2$*, iff*

$$\forall M, N \in \mathcal{T}(\mathcal{F}, \mathcal{N}_{pub}, \mathcal{AX}).\ M\sigma_1 =_E N\sigma_1 \Leftrightarrow M\sigma_2 =_E N\sigma_2$$

We can now define equivalence of processes P and Q by requiring that for any trace of P there exists a trace in Q obtained by the same sequence of labels, i.e., attacker actions, and resulting in statically equivalent attacker knowledge (and vice versa).

Definition 4 (Trace equivalence)**.** *Let* P *and* Q *be two processes.* P *is a trace included in* Q*, written* $P \sqsubseteq_E Q$*, iff*

$$\forall tr, \sigma_P.\ \exists \sigma_Q. \quad \begin{array}{c} (tr, \sigma_P) \in traces^{pi}_{\approx}(P) \\ \Longrightarrow \\ (tr, \sigma_Q) \in traces^{pi}_{\approx}(Q) \ \wedge\ \sigma_P \sim_E \sigma_Q \end{array}$$

P and Q are trace equivalent, written $P \approx_E Q$, iff $P \sqsubseteq_E Q$ and $Q \sqsubseteq_E P$.

3.5 Overview of translations and results

We give here a summary of the translations and associated correctness results. Recall that Tamarin's input language is a set of multiset rewrite rules and security properties are expressed in the first-order logic of Section 3.4.1. ProVerif's specification language, on the other hand, is a dialect of the applied π-calculus, as Sapic$^+$, but with subtle differences between both languages. Properties in ProVerif are expressed either using dedicated queries for reachability properties or observational equivalence. We will denote the satisfaction relation of ProVerif's reachability queries by $\models^{\mathrm{PV}}$.

Reachability properties We will show that, for reachability it is possible to directly translate Sapic$^+$ specifications into both Tamarin's and ProVerif's input formats. In the following sections, we define in more detail these translations which we denote by $[\![\cdot]\!]^{\mathrm{Tam}}$ for the translation to Tamarin and $[\![\cdot]\!]^{\mathrm{PV}}$ for the one to ProVerif (by abuse of notation we use the same translation function for both the processes and the security formulas). We suppose that reachability formulas only contain $K(\cdot)$ facts, as well as facts defined through events (to avoid clashes with reserved facts added by the translation). The correctness and the conditions under which the translation can be applied are stated in the following theorems.

Theorem 1. *Let P be a process and φ a formula such that P does not contain conditional branchings with destructors.*

$$P \models \varphi \quad \Leftrightarrow \quad [\![P]\!]^{\mathrm{Tam}} \models [\![\varphi]\!]^{\mathrm{Tam}}$$

One may note that although the property language for Sapic$^+$ and Tamarin is the same, we need to translate the formula φ as it encodes additional information. We can now define a similar correctness theorem for the translation to ProVerif.

Theorem 2. *Let P be a process and φ a formula such that*

- $\mathrm{Pat} = \mathrm{Pat}_{ex}(\mathcal{F}_{\mathsf{data}})$ *and P does not contain conditional branchings with variable quantifiers;*
- *φ is a formula with a single quantifier alternation.*

Then $P \models \varphi \Leftrightarrow [\![P]\!]^{\mathrm{PV}} \models^{\mathrm{PV}} [\![\varphi]\!]^{\mathrm{PV}}$.

Equivalence For equivalence, we can export Sapic$^+$ to either the ProVerif or DeepSec tool.

ProVerif indeed allows to verify a strong process equivalence, that we denote by $P \cong_E Q$. It was shown in [17] that $\cong_E$ implies observational equivalence, which in turn implies trace equivalence. Therefore, for *extended processes*, i.e., non-stateful processes, and the same hypotheses on processes as in in Theorem 2 we have that

$$\text{if } [\![P]\!]^{\mathrm{PV}} \cong_E [\![Q]\!]^{\mathrm{PV}} \textit{ then } P \approx_E Q.$$

Moreover, on elementary processes the syntax of Sapic$^+$ and DeepSec coincide for predicates restricted to equality. Therefore, we can directly, for free, employ DeepSec to verify trace equivalence on this subclass of processes. Hence, we will not detail the (straightforward) translation to DeepSec in the following. Note, however, that DeepSec only supports a constructor-destructor term algebra defined by a subterm convergent rewrite system and no equational theory. As of now, there is no clear link between process equivalence and Tamarin's notion of equivalence between rewrite systems [12]. We thus do not translate equivalence properties to Tamarin.

4 From Sapic$^+$ to Tamarin

We extend the 2014 translation from Sapic to Tamarin [36] with new syntactic features that improve usability (cf. Appendix D) or interoperability with ProVerif (let binders with pattern matching and destructors). Moreover, we heavily optimize the number of rules produced and the encoding of common edge cases for communication and database access.

Multiset rewriting Tamarin describes processes as sets of multiset rewrite rules (MSRs), each of form $l \dashv\![\; a \;]\!\mapsto r$ with l, a and r being multisets of facts. Facts consist of a fact symbol (e.g. F) and a (possibly empty) sequence of terms $t_1, \ldots, t_n$), written $F(t_1, \ldots, t_n)$. Substitution application can be lifted to facts and then MSRs, where the result is called a rule instance. The system state is a multiset of facts S. It can be rewritten by any rule instance $l \dashv\![\; a \;]\!\mapsto r$ that *applies*, meaning that l is contained in S (modulo the equational theory E). The application of this rule substitutes all facts in l by those in r. The set of traces is the sequence of actions $a_0, \ldots, a_n$ in a series of steps starting from the empty multiset, $\emptyset \xrightarrow{a_0} S_0 \cdots \xrightarrow{a_n} S_n$. The fact $\mathsf{Fr}(n)$ is special: it can only be introduced once per name n.

Sapic translation We only give the idea of the translation here and refer to [36] for details. A SAPIC reduction $\mathcal{C} \rightarrow \mathcal{C}'$ is represented by one or more MSRs of form

$$\mathsf{state}_{\mathfrak{p}}(\tilde{x}), \ldots \xrightarrow{a} \mathsf{state}_{\mathfrak{p}'}(\tilde{x}'), \ldots$$

where $\mathsf{state}_{\mathfrak{p}}(\tilde{x})$ is a 'ticket' to apply this rule. That 'ticket' is available (i.e., present in S) iff the set of currently running processes contains the subprocess of the overall process at position $\mathfrak{p}$[1] with all unbound variables and names set to the values $\tilde{x}$. Protocol inputs and outputs or name restrictions can be expressed by adding Tamarin's special facts In, Out and Fr to the left or right, respectively. Parallel execution is captured by having two state-facts on the right, conditionals by having two rules with the same state-fact on the left, etc.

For efficiency reasons, state manipulation is not encoded in the system state S, but instead in the actions: inserts, lookups, locks and unlocks follow the above schema, but the action a logs these events, e.g., $\mathsf{state}_{\mathfrak{p}}(\tilde{x}) \xrightarrow{Insert(s,t)} \mathsf{state}_{\mathfrak{p}'}(\tilde{x})$ represents an insertion, $\mathsf{state}_{\mathfrak{p}}(\tilde{x}) \xrightarrow{Lookup(s,y)} \mathsf{state}_{\mathfrak{p}'}(\tilde{x} \cdot y)$ represents a lookup. In the latter, y is a priori unbound; however, a trace is only considered valid if it adheres to the following restriction: each lookup action follows a corresponding insert action that was not overwritten and any lock is either the first for its term, or a previous unlock released it.

Example 1. *The process* $\mathsf{new}\ a; (\mathsf{event} E(a) \mid \mathsf{out}\ (c, h(a)))$ *translates to* $\mathsf{Fr}(a) \dashv\![]\!\mapsto \mathsf{state}_1(a), \mathsf{state}_2(a)$ *for the top-level* new*, and two rules for either side of the parallel:* $\mathsf{state}_1(a) \dashv\![\; E(a) \;]\!\mapsto \emptyset$ *and* $\mathsf{state}_2(a), \mathsf{In}(c) \dashv\![]\!\mapsto \mathsf{Out}(h(a))$*. In the latter rule,* $\mathsf{In}(c)$ *enforces that the channel* c *is known to the attacker before it can retrieve* $h(c)$*. Using the predefined rule for freshness* $\emptyset \dashv\![]\!\mapsto \mathsf{Fr}(x)$*, we can rewrite* $\emptyset$ *to* $\{\ \mathsf{Fr}(b)\ \}$ *(for any* $b \in \mathcal{E}$*) and proceed as follows:* $\{\ \mathsf{Fr}(b)\ \} \rightarrow \{\ \mathsf{state}_1(b), \mathsf{state}_2(b)\ \} \xrightarrow{E(b)} \{\ \mathsf{state}_2(b)\ \}$*, producing the trace* $E(b)$*.*

4.1 The Sapic$^+$ extensions

First, we show how let bindings with else-branches can be translated to Tamarin, then, how we can add destructors on their right-hand side.

4.1.1 Let bindings with pattern matching

Sapic only supports let bindings for a single variable of the form $\mathsf{let}\ x = t\ \mathsf{in}\ P$. We generalize to pattern matchings with a failure branch of the form $\mathsf{let}\ s = t\ \mathsf{in}\ P\ \mathsf{else}\ Q$ where $s, t \in \mathcal{T}(\mathcal{F}_c, \mathcal{E}, \mathcal{X})$.

[1]We can represent a process as a binary tree and identify positions via sequences of 1 and 2 for the left and right branch, respectively,

Assuming the current position in the process is represented by $\mathsf{state}_{\mathfrak{p}}(\tilde{x})$, we add the following rules, where $v_1, \ldots, v_n = vars(s) \setminus \tilde{x}$ are the unbound variables in the pattern.

$$\mathsf{state}_{\mathfrak{p}}(\tilde{x}), \mathsf{Fr}(n) \to \mathsf{Let}(t, n), \mathsf{state}^{\mathsf{semi}}_{\mathfrak{p}}(\tilde{x}) \tag{1}$$

$$\mathsf{state}^{\mathsf{semi}}_{\mathfrak{p}}(\tilde{x}), \mathsf{Let}(s, n) \to \mathsf{state}_{\mathfrak{p}\cdot 1}(\tilde{x} \cdot v_1 \cdot \ldots \cdot v_n) \tag{2}$$

$$\mathsf{state}^{\mathsf{semi}}_{\mathfrak{p}}(\tilde{x}), \mathsf{Let}(y, n) \:-\!\![\ \mathsf{NoMatch}_s(\tilde{x}, t)\ ]\!\!\to \mathsf{state}_{\mathfrak{p}\cdot 2}(\tilde{x}) \tag{3}$$

The fresh name n identifies the Let-fact with this rule instance. The $\mathsf{state}^{\mathsf{semi}}_{\mathfrak{p}}(\tilde{x})$ represents a 'semi'-state in the process where the reduction of the let-process at position $\mathfrak{p}$ is not yet completed. It permits the application of one of the two following rules.

Rule (2) represents the case where the term t in $\mathsf{Let}(t, n)$ can be matched with the pattern s in $\mathsf{Let}(s, n)$ and the process moves into the positive branch P at position $\mathfrak{p}\cdot 1$, while rule (3) is the else branch where its left-hand side (lhs) fact $\mathsf{Let}(y, n)$ with the fresh variable y can merge with any $\mathsf{Let}(t', n)$ produced by rule (1). To enforce that we only take the else branch when the matching fail, we use a restriction: the third rule can only appear in a trace if s cannot be instantiated to match t.

$$\forall \tilde{x}, t, i.\ \mathsf{NoMatch}_s(\tilde{x}, t)@i \implies \forall v_1, .., v_n.\ t \neq s$$

In Appendix E.1.1, we show the correctness of this translation. Notably, we managed to extend the 40-page proof from [36] in a black box fashion with the following technique. Their SAPIC semantics includes a feature for embedding MSRs within the process, which we omitted here for the ease of presentation. We expressed the above three rules using this feature, and show that the translation of the let construct into a process with the embedded MSRs is correct. The correctness of the translation to MSRs then follows by transitivity.

4.1.2 Let bindings with destructors

We now generalize this development to allow destructor function symbols to appear on the right-hand side (rhs) of a let expression. Recall that Tamarin does not support destructor functions and hence all terms in the resulting MSRs need to be in $\mathcal{T}(\mathcal{F}_c, \mathcal{N}, \mathcal{X})$. On the other hand, the rhs term t in the process $\mathsf{let}\ s = t\ \mathsf{in}\ P\ \mathsf{else}\ Q$ may contain destructor symbols. For simplicity, we focus on the case where the destructor symbol dest must be at the top level [2] and where only one rewrite rule applies to dest and this rewrite rule reduces to a single variable[3]. We translate $\mathsf{let}\ x = \mathsf{dest}(\vec{t})\ \mathsf{in}\ P\ \mathsf{else}\ Q$ where $\mathsf{dest}(\vec{u}) \to x \in E_D$ to $\mathsf{let}\ \vec{u} = \vec{t}\ \mathsf{in}\ [\![P]\!]^{\mathsf{dest}}\ \mathsf{else}\ [\![Q]\!]^{\mathsf{dest}}$ with all variables in $\vec{u}$ fresh except for x and $[\![\cdot]\!]^{\mathsf{dest}}$ the recursive call to this translation. Note first that neither $\vec{u}$ nor $\vec{t}$ contain a destructor symbol, so the translated let-expression is covered by the translation for pattern matches we just discussed. The translated process checks that the parameters to the destructor $\vec{t}$ match the pattern $\vec{u}$ defined by the reduction. If they match, this will assign a value to the variable x that occurs in $\vec{u}$. We once again extend the proof of correctness in a black box fashion (Appendix E.1.2).

4.2 Optimizations

The number of rules produced by the translation and the encodings chosen have decisive impact on the verification speed, as we will show. This is the reason why handwritten Tamarin models are typically faster to verify than translated models. For non-experts, what constitutes the best encoding is opaque and performing these optimizations is often out of reach. Here, we show the potential for improvement by discussing two optimizations, one that applies in general, one that applies to a frequent edge case.

[2]We can extend this to arbitrary destructor terms in $\mathcal{T}(\mathcal{F}_c, \mathcal{N}, \mathcal{X})$ simply by nesting several let expressions, one for each destructor symbol in the term.

[3]The transformation can be extended to the case where destructors are defined with multiple rewrite rules $\mathsf{dest}(x_i) \to u_i$ by sequentially applying the transformation of $\mathsf{dest}(x_{i+1}) \to u_{i+1}$ on the else-branch of the translation of $\mathsf{dest}(x_i) \to u_i$. Since our rewrite system is convergent, we can arbitrarily fix the order of application of rewrite rules.

	# rules		running time (s)		
Case Study (Section 6.2)	NC	C	NC	C	R
`KEMTLS` [45]	98	42	1.4k	47	28
`KEMTLS-CA` [45]	124	57	20.7k	1.1k	18
`KEMTLS-NO-AEAD` [45]	94	41	11.7k	222	52
`LAKE-DH-KCI` [46]	86	48	1.3k	320	4
`LAKE-DH-FS` [46]	121	56	1.5k	265	5
`SSH` [47]	67	37	53	4	11
`SSH-NEST` [47]	106	58	3.1k	16	10
`Privacy-Pass` [20]	44	20	13	10	1.3
`AC` [10, 33]	48	12	2	1	2
`AKE` [10, 33]	35	15	1.3	0.8	1.5
`OTP` [32, 33]	94	40	17	6	3
`NSL` [40, 37]	59	25	80	18	4.4

NC: without compression C: with compression R: ratio

Figure 4: Benchmarks for path compression

4.2.1 Path compression

The original Sapic translation produces at least one rule per position in the process tree, but often they can be compressed. Consider the translation of the process $\mathsf{new}\ a;\ \mathsf{new}\ b;\ \mathsf{out}(c, \langle a, b\rangle)$. We obtain three rules:

$$\mathsf{state}_{[]}(), \mathsf{Fr}(a) \to \mathsf{state}_1(a) \quad \mathsf{state}_1(a), \mathsf{Fr}(b) \to \mathsf{state}_{11}(a, b)$$
$$\mathsf{state}_{11}(a, b) \to \mathsf{state}_{111}(a, b), \mathsf{Out}(c, \langle a, b\rangle)$$

As only the last step is observable, these three rules can be compressed into one:

$$\mathsf{state}_{[]}(), \mathsf{Fr}(a), \mathsf{Fr}(b) \to \mathsf{state}_{111}(a, b), \mathsf{Out}(c, \langle a, b\rangle)$$

This compression step is not always permissible; hence we need to define carefully under which circumstances correctness is maintained. For instance, two rules cannot be compressed if the second rule may require the attacker to know the output of the first. e.g., $\mathsf{new}\ a; \mathsf{out}(c, a); \ldots$ cannot be compressed with its continuation $\ldots; \mathsf{in}(c, x); \ldots$.

We define this compression as an optimization of the MSRs that is, in principle, applicable to handwritten Tamarin models. We provide in Appendix E.2.1 the complete set of conditions under which rules can be compressed and the corresponding proof (once again made in a black box fashion).

We find that this optimization is very effective (see Fig. 4). On all examples, the reduction in the number of rules is significant, and the running time of the verification can be reduced by a factor of up to 52. We observe that the ratio is more pronounced on examples where verification is slow to begin with, possibly indicating that path compression is most effective during the constraint solving procedure (as opposed to precomputation).

4.2.2 Alternate secret channel encoding

For private communications, it depends on the knowledge of the adversary whether or not the sender can proceed after emitting a message. Naturally, the encoding of channels in Sapic$^+$ reflects that, but typically, private channels remain trivially secret throughout the protocol run. We optimize the translation for this frequently occurring special case.

We syntactically check a sufficient condition of secrecy: a name n is a secret channel if there is a single process of form $\mathsf{new}\ n; P$ and all other occurrences of n have either the form $\mathsf{in}(n, m)$ or

$\mathsf{out}(n, m)$ with n not occurring in m. If the condition is fulfilled, we can remove one out of two rules produced for each out-processes and one out of three rules in the translation of in-processes. The rules removed capture the case where an attacker can deduce the channel name and thus trigger the asynchronous communication behavior. It both reduces the number of rules and removes the need for a case distinction about whether n is deducible.

We find significant performance improvements with three case studies (see Fig. 6). `SSH-NEST` and `OTP` will be detailed in Section 6.2 and have a speed up of two and three, while `U2F-TOY` was already in the SAPIC's repository. Note that on the latter, TAMARIN times out without this optimization.

5 From Sapic$^+$ to ProVerif

The syntax and semantics of PROVERIF are very similar to SAPIC$^+$ as both tools use dialects of the applied π-calculus for their input languages.

Translating conditionals Extended processes only differ in the semantics of conditional branching containing destructors. In SAPIC$^+$ $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ reduces to P when $u =_E v$, which implies that both u and v reduce to constructor terms. Q is executed when $u \neq_E v$, thus when u and v are not equal or u or v is not a message. In PROVERIF, the semantics for the *then* branch is the same, however, Q is only executed when both u and v *are* messages and u and v are not equal. The process blocks when either u or v is not a message. Such $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ conditional branching is translated in PROVERIF using let bindings.

This may seem anecdotal, but verification results can change if a part of the process becomes non-executable. PROVERIF or DEEPSEC users may not even be aware of this discrepancy, illustrating the risks of working with different tools. There are multiple such pitfalls when trying to move from models in PROVERIF, TAMARIN or DEEPSEC, either due to some subtle differences between the semantics or different implementation choices. Notably, bindings inside pattern matchings, destructor inside conditional and pairs are not handled the same way in PROVERIF and TAMARIN. We detail those differences in Appendix C.

5.1 Translating states

PROVERIF has no direct syntax for stateful processes and requires an encoding using internal communication on private channels. Although semantically correct, PROVERIF struggles to prove security properties relying on this encoding which leads to false attack. The recent GSVERIF front end [21] addresses many false attacks related to stateful protocols. However, GSVERIF's transformations easily lead to non-termination issues and are sometimes too restrictive to handle general stateful processes. We therefore updated the GSVERIF tool (available in [1, 2]) to lift these restrictions and better exploit the most recent features of PROVERIF (*e.g.*, lemmas, axioms) leading to more efficient verification and favoring termination.

5.1.1 From states to channels

Although the applied π-calculus does not provide explicit constructs for stateful processes, private *memory cells* are classically encoded by reading and writing on private channels. For a private channel `cell`, reading the cell is achieved by `in(cell,x); P`, and writing by `out(cell,t)|P`. Note that for writing, the output is put in parallel as communication over private channels is synchronous. With this encoding, as long as an output on `cell` is not available, the cell is in fact locked.

States in SAPIC$^+$ behave similarly to this encoding when a lookup is directly preceded by a lock, or an insert directly followed by an unlock. We call such a state *pure*. They can be translated through replacement of

$\mathsf{lock}\ t; \mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ \_\ \mathsf{else}\ 0$	by	$\mathsf{in}(t, x); \_$	and
$\mathsf{insert}\ t, t'; \mathsf{unlock}\ t; \_$	by	$\mathsf{out}(t, t');\ \_$	

where $n \in \mathcal{N}_{\mathtt{priv}}$ is a fresh private name. We give a precise definition of pure states and show the correctness of this encoding in Appendix E.2.3.

This transformation also provides an alternate encoding when translating to Tamarin. Interestingly, it does not always improve performance, although it describes a natural way to write cells in Tamarin. E.g., on the `AC` protocol [10, 33], the verification time shrinks from 127 seconds to 1 but on the `SD` protocol [3, 37], the encoding seems to lead to non-termination. Using Sapic$^+$, we can easily switch between both encodings.

5.1.2 Handling non-static states

The above translation of states is efficient and sufficient for most of our cases studies. However, it does not cover all the Sapic$^+$ processes. Notably, two syntactically different states need to be un-unifiable, since our transformation replaces the state t with a fresh private name. Thus, if two states were to become unified in the Sapic$^+$ process, then their respective translated fresh channel names would still be considered as different, hence yielding an unsound translation. We show how we could build a more general translation that removes these restrictions.

Though ProVerif does not have a native syntax for states, it allows declaring *tables* that are a form of global memory. Processes can insert and look up elements inside a table. However, unlike states in Sapic$^+$, tables are append-only: once inserted, an element cannot be removed nor replaced. The syntax for table is illustrated as follows:

```
table my_tbl(bitstring,bitstring).
let P(u:bitstring,v:bitstring) = insert my_tbl(u,v);
  in(c,x:bitstring); get my_tbl(=x,z) in Q else R
```

`table my_tbl(bitstring,bitstring)` declares a table `my_tbl` containing pairs of `bitstring`. `insert my_tbl (u,v)` inserts the pair `(u,v)` in the table whereas `get my_tbl(=x,z) in Q else R` executes `Q` if `my_tbl` contains an element `(x,t)` for some `t`, and in that case binds `z` to `t`; otherwise `R` is executed. Note that if multiple elements match the pattern `(=x,z)`, then ProVerif chooses non-deterministically which element is bound to z.

We rely on these tables to encode states that may be unified with other states as follows: we declare two new tables `lock_tbl` and `state_tbl`, both containing pairs of `bitstring` and `channel (t,n)` where `t` corresponds to Sapic$^+$'s state and `n` is the fresh private channel used in our state translation. Intuitively, to lock a state `t`, we first lookup in the table whether a private channel has already been generated for a state equal to t (modulo E). If so, we translate the lock as an input on this private channel; otherwise, we generate a fresh private channel, insert it in the table and execute the remaining process. Intuitively, the Sapic$^+$ construct `lock t; P` translates to ProVerif as:

```
get lock_tbl(=t,x_n) in
  in(x_n,y); P
else new n; insert lock_tbl(t,n); P
```

Similarly, `unlock t; P` consists of a look up for a private channel for `t` and an output on this channel.

```
get lock_tbl(=t,x_n) in out(x_n,0) | P else 0
```

Note that for the lock/unlock mechanism, it is irrelevant which term is sent on the private channels; hence we always output the constant 0.

A caveat remains: consider the process $(\mathsf{lock}\ t; P)|(\mathsf{lock}\ t; Q)$. Assuming an empty table, the execution can start by executing the construct `get lock_tbl(=t,x_n)` for both locks, reducing into the two else branches (since the table is empty). Thus two fresh channels would be generated and inserted for t. To prevent this race condition, we consider a private channel `glock` that will act as a global lock to ensure that the else branch is executed before any other lookup in the table `lock_tbl`.

```
in(glock,z); get lock_tbl(=t,x_n) in
  out(glock,z) | in(x_n,y); P
else new n; insert lock_tbl(t,n); (out(glock,z) | P)
```

The translation for the SAPIC$^+$ lookup and insert follows a similar structure, but as these two constructs are not blocking, the placement of input and output on private channels differs. See Appendix F.2 for the translation and its proof of correctness.

5.2 Translating queries

PROVERIF translates security properties of various kinds (*e.g.* authentication, secrecy, non-interference, real-or-random secrecy) into either reachability queries (called correspondence query in PROVERIF's manual) or equivalence queries. For reachability properties, we know that the state transformation preserves the set of traces. However, not all SAPIC$^+$ queries can be translated into PROVERIF reachability queries, as PROVERIF permits only one quantifier alternation. Furthermore, the translation of special facts about communication or adversarial knowledge requires great care due to semantic differences. Note that GSVERIF supports only reachability queries, hence support for equivalence queries is limited to protocols without state.

Reachability properties In the latest PROVERIF release, reachability properties can be described through a sorted first-order logic formula over atomic trace formulas as defined in Definition 1, similarly to SAPIC$^+$ and TAMARIN. There are, however, some differences in the facts and the fragment of the logic that prevent translation of all SAPIC$^+$ queries to PROVERIF queries and vice versa.

The first difference is that PROVERIF only considers queries with at most one quantifier alternation. More specifically, a query must be of the form $F_1@i_1 \wedge \ldots \wedge F_n@i_n \Rightarrow \phi$ where ϕ is a quantifier-free trace formula and its disjunctive normal form (DNF) does not contain negations of facts $F@i$. All variables in $F_1@i_1 \wedge \ldots \wedge F_n@i_n$ are quantified universally and remaining variables in ϕ are quantified existentially. For example, the following query in PROVERIF syntax

```
query x,y:my_type, z:bitstring, i,j,k:time; event(A(x))@i && event(B(y))@j  ⟹  event(C(x,y,z))@k
    && k < j
```

quantifies variables x, y, i, j universally and z, k existentially. In SAPIC$^+$'s syntax, this query is expressed as

```
All x y #i #j. A(x)@i & B(y)@j  ⟹  Ex z #k. C(x,y,z)@k & k < j
```

The second difference is the set of allowed facts and their semantics. Facts in PROVERIF include events, as illustrated by the previous example, *message facts* of the form `mess(c,t)` that hold when a term t has been sent on channel c, and *attacker facts* of the form `attacker(t)` that hold when the attacker can deduce the term t.

Though message and attacker facts may seem very similar to the K fact, their semantics are incomparable. Like for the K fact, an attacker can trigger `mess(c,t)` provided it can deduce c and t. However, `mess(c,t)` is also triggered when the output and input rules on private channels are executed.

Unfortunately, the attacker fact does not correspond to a K fact either: `K(t)@i` holds when the attacker *did deduce* `t` at timestamp `i` (e.g., to execute an input on a public channel); whereas `attacker(t)@i` holds if `t` *is deducible* at the timestamp `i`. This difference in semantics has important ramifications. First, two K facts cannot hold at the same timestamp but two attacker facts can. Second, and more importantly, it makes the translation of queries impossible when a K fact occurs in the conclusion of the query. Consider the process `new a; new b; out(c,(a,b)); event A` and the following queries.

```
query i,j:time; event(A)@i  ⟹  attacker(a)@j.
```

```
All #i. A@i  ⟹  Ex #j. K(a)@j
```

The first (PROVERIF) query holds whereas the second (SAPIC$^+$) does not. The query

```
query i,j:time; event(A)@i  ⟹  mess(c,a)@j
```

does not hold either but as explained above, message facts cannot translate K facts either due to the internal communication.

The semantic issue of the `K(t)@i` fact versus the attacker fact only occurs when it is existentially quantified. When the fact is universally quantified, however, we can safely translate $K(t)@i$ to `attacker(t)@i`. For example, the following

```
All x y #i #j. A(x)@i & K(y)@j ⟹ Ex z #k. C(x,y,z)@k
```

will be correctly translated into the query

```
query x,y,z:bitstring, i,j,k:time; event(A(x))@i && attacker(y)@j ⟹ event(C(x,y,z))@k
```

Therefore, we only allow universally quantified K facts in queries.

ProVerif also allows *table facts* checking whether an element occurs in a table and *injective events*. We obviously ignore table facts since tables are not part of our syntax. Injective events allow to express a correspondence query where the occurrence of some event can be injectively associated to another event. For example, `inj-event(A) ⟹ inj-event(B)` ensures that there are at least as many `B` events as there are `A` events. Such properties cannot be expressed in Sapic$^+$'s first-order logic and would require to extend the query syntax. Sapic$^+$, however, allows users to export a ProVerif query in the input file if injective queries are needed (avoiding a separate file).

6 Practical evaluation and case studies

6.1 The implementation

We integrated our translation procedures as a separate package that is distributed with and integrated into Tamarin, in contrast to SAPIC's original compiler [37], which is a separate program. The benefits are threefold: we benefitted from Tamarin's libraries for manipulating and parsing terms, we could integrate our translation into Tamarin's graphical user interface and, finally, Sapic$^+$ will instantly support future extensions to Tamarin's term algebra. The package is open source and compatible with MacOS and Linux. We provide a docker image **robertkuennemann/sapicplusplatform**[1] that allows to run all the tools easily and reproduce our case studies (detailed instructions in Appendix A).

The code uses a common parsing infrastructure and shares code for typing and annotating code (e.g., to identify matching lock and unlocks). The translation code is highly modular and exploits template mechanisms for easily adjustable output to ProVerif,GSVerif and DeepSec. We added about 5500 lines of code for the exports and the optimizations. Once the ProVerif export and the modular interface were done, the DeepSec export required around 200 lines of code.

The user provides an input file in Tamarin's `spthy`-format, which can include a process, and can choose to either translate into the target language of choice, or use Tamarin's internal constraint solver.

6.2 The new workflow in action

We provide below several complex case studies of real-life protocols, some of which constitute their first formal analysis. We also provide some case studies that were adapted either from existing Sapic$^+$ or Tamarin models. As a whole, our set of case studies illustrate:

- how Sapic$^+$ allows leveraging ProVerif's high level of automation when Tamarin's automation is insufficient;
- how Sapic$^+$ allows using Tamarin to prove lemmas that cannot be proved by ProVerif so they can be used as axioms inside ProVerif;
- how DeepSec can be used when ProVerif cannot prove equivalence;

Case Study	ProVerif	Tamarin	Property
KEMTLS [45]	1	47	Auth., PFS
KEMTLS-NO-AEAD	1	222	Auth., PFS
KEMTLS-CA	1	1145	MA, Unlink.
LAKE-DH-FS [46]	17	265	MA, PFS
LAKE-DH-KCI	7	320	KCI
SSH [47]	1	4	MA, Sec.
SSH-NEST	2	316	MA, Sec.
SSH-NEST(X)	$7 \times 3^{X-1}$	∞	MA, Sec.
Privacy-Pass [20]	1	7	Unforg., Unlink.
NSL [40, 37]	1	18	MA, Sec.
	DeepSec		
KEMTLS-CA [45]	240		Unlink. (3 sess.)
NSL [40, 37]	1		Strong Sec. (1 sess.)

PFS: Perfect Forward Secrecy KCI: Key Compromise Impersonation
Auth.: Authentication MA: Mutual Authentication Sec.: Secrecy
Unforg. : Unforgeability Unlink. : Unlinkability

Figure 5: Running time (seconds) of the case studies

- and finally that the Tamarin models produced by Sapic$^+$ can be analyzed as efficiently as dedicated and fine-tuned Tamarin models.

We note that there are cases where a direct encoding is preferable to a translation from Sapic$^+$. One instance are protocols with complex state machines, which are more easily expressed in Tamarin. Fig. 5 summarizes the running times of some of our case studies on a 2019 13”-MacBook Pro with 16GB RAM.

6.2.1 Novel case studies

`KEMTLS` Many post-quantum secure alternatives to currently deployed key-exchange protocols are starting to appear. In the coming years, formal verification tools are likely to play an essential role in their standardization process. We present here an analysis of `KEMTLS` [45], a recently proposed alternative for the core key exchange of TLS.

The protocol relies on two main ingredients: a signature scheme for the server certificate and a Key Encapsulation Mechanism (KEM). Intuitively, a KEM is an asymmetric encryption scheme that uses the parties' long-term public keys to encrypt an ephemeral secret, which can be used to derive a shared key.[4] `KEMTLS` derives a secret shared key with authentication of the server after ten messages and six intermediate keys. We used Sapic$^+$ to verify the protocol over three models:

- A first, basic `KEMTLS` model, for which we verify server authentication and Perfect Forward Secrecy (PFS) of the final key. ProVerif proves these properties in a few seconds, Tamarin in around a minute.
- The `KEMTLS-NO-AEAD` model sanity checks the security claims. Indeed, in `KEMTLS`, most messages are encrypted using authenticated encryption, but in its security proof, nothing is assumed about this primitive. We thus modeled a version of `KEMTLS` without any encryption, and proved that the same security properties hold.
- The `KEMTLS-CA` model, where we model the client authentication option. This option was proposed in Appendix C of [45] to match the existing option of TLS, but was provided

[4]In the symbolic model, we assume that keys generated by a KEM are uniformly sampled.

without a security proof. It allows a user to authenticate to a server with a long-term certificate. We proved using ProVerif (1 second) and Tamarin (20 minutes) mutual authentication for this extension. We were also able to prove unlinkability of clients, modelled as strong unlinkability in terms of trace equivalence as defined e.g. in [31, 8], for three sessions using DeepSec, and an unbounded number of sessions with ProVerif.

`LAKE` Lightweight Authenticated Key Exchange (`LAKE`) is a recently standardized key exchange protocol designed for resource-constrained devices. It is a DH based key exchange, where authentication can either rely on a signature scheme and long-term public keys, or on a long-term DH share and its public group element. With Sapic$^+$, we could develop correct models of this protocol quickly. We debugged and proved them with ProVerif's approximate version of the equational theory, before strengthening the guarantees using the same model but with Tamarin's more complete theory.

Our models include both authentication modes, the client being able to chose between them dynamically. In the `LAKE-DH-FS` model, we proved mutual authentication, KCI and PFS of the exchanged key.

In concurrent work [42], a different version of the `LAKE` IETF draft was also analyzed with Tamarin. While they consider more properties than us, we consider a model supporting two different authentication modes.

`SSH` It is a widely deployed protocol that notably allows logging in on a remote server, relying on a user's long-term signature key. In its basic version, it has been analyzed previously using computer-aided verification tools, both symbolic [39] and computational [19]. The protocol supports an additional agent-forwarding feature, allowing SSH connections to be nested. Analyzing this feature is complex: a secure channel established using a long-term signature key is used to forward a signature request for the same long-term key. The protocol with a single depth nesting was analyzed with an interactive prover in the computational model [7], but it relies on an external composition result.

The `SSH` model verifies authentication and secrecy. Interestingly, a previous Tamarin model [39] required 26 seconds while our Sapic$^+$ model is verified in three seconds.[5] Sapic$^+$ can thus compile models that are verified in the same order of magnitude as a handmade Tamarin model.

The `SSH-NEST` model contains one nested connection. This model is verified very efficiently using ProVerif, and takes around 5 minutes in Tamarin with the more precise DH modeling. The `SSH-NEST(`X`)` model supports an arbitrary depth nesting, but with checks to manually bound the depth to X. Without a fixed bound, we were unable to make ProVerif terminate. We ran ProVerif with a bound up to 5: verification takes 7 seconds for depth $X = 1$, and each increment in depth approximately triples the running time, up to a running time of 6 minutes for depth 5. We were unable to make Tamarin terminate.[6]

`Privacy-Pass` The `Privacy-Pass` protocol, proposed in [27] and under standardization [20], is a token-based authorization of clients to servers that aims to preserve client anonymity. A server issues tokens upon client requests and tokens can be spent only once. It is based on a complex cryptographic primitive, a Verifiable Oblivious Pseudo-Random Function (VOPRF), whose equational theory is not supported by DeepSec. With the `Privacy-Pass` model, we verified token unforgeability, using ProVerif and Tamarin, and unlinkability (as defined in [31, 8]) of the clients using ProVerif.

6.2.2 Existing Models

We used the export feature on some existing Sapic protocol models to illustrate its ease of use, where with only minor modifications, all models were executable in ProVerif. In addition, we

[5]Timing obtained by running the SSH model of [39] on the same laptop with the same version of Tamarin.

[6]The model leads to over a thousand partial deconstructions.

also ported an existing Tamarin model.

5G AKA A previous case study of the 5G AKA authentication standard was performed recently in Tamarin [11], leading to one of the most complex Tamarin analysis to date. On this model and with on a complicated handwritten oracle to guide the tool, they were able to verify 4 sanity check lemmas, 3 security properties and find 7 attack traces.

To illustrate the usability and interest of Sapic$^+$, we have reimplemented their main model using the platform. We were able to design a model such that after adapting their handwritten oracle, Tamarin was able to prove the 3 security properties in around a minute on the model produced by Sapic$^+$. In comparison, the original Tamarin model took 3 times longer to verify those 3 properties. This example thus give evidence that Sapic$^+$ can even on complex case studies produce models that are as fast, or even faster to verify than Tamarin fine-tuned ones.

Moreover, by manually adding a simple parameter to guide the resolution, ProVerif finds 6 of the 7 attack traces automatically, in contrast to the original model, which required a complex oracle to find these attack traces. The last attack can also be found by ProVerif, but requires editing the process to help the trace reconstruction terminate.

SGX report models With Sapic$^+$, the modeling of the SGX report capability from [33] carries over to ProVerif for free. On the models of `AC`, `AKE` and `OTP`, ProVerif always answered in a second and Tamarin in a few seconds. Some interesting results were that on the `SOC` protocol, ProVerif does not terminate, but Tamarin answered in seconds. On a flawed model of `AC`, the `AC-F-sid` model, ProVerif reports 'cannot be proved', while Tamarin successfully reconstructed the attack trace. Nevertheless, with minor modifications to the generated ProVerif model[7], we could ensure termination and attack reconstruction.

Others Finally, we imported the `Scytl` e-voting protocol [24] from the GSVerif benchmark [21], and the secure device (`SD`)[3, 37] and `NSL` models[37] from Sapic's repository. We verify the strong secrecy of the key in `NSL` for one session with DeepSec, where ProVerif reports 'cannot be proved'. On the `Scytl` protocol, ProVerif answers in a second and Tamarin in a minute, but ProVerif reports 'cannot be proved' on the single vote property (without further expert input) while Tamarin proves it.

6.2.3 Tamarin proofs of ProVerif axioms

Since its recent update [18] ProVerif allows specifying axioms that can be used to prove a protocol by adding an extra assumption that may hold but that ProVerif cannot prove. An example of such a case is given in Example 6 of [18], where an axiom is used to specify that an action can only occur once. Consider the following subprocess

```
new stamp; in(c,xr); event S(stamp,xr); new r;
out(c,sign(aenc(vote,(r,xr),pk (sk e),sk ))
```

On such a process, ProVerif cannot prove that the event S can only occur once with a given timestamp, due to its abstraction. As this fact is required to prove the expected security properties, the following axiom is needed:

```
All stamp xr xr2 #i #i2. S(stamp,xr)@i & S(stamp,xr2)@i2
        ⟹ i=i2 & x=xr2
```

While ProVerif cannot prove this, this is the sort of formula that Tamarin typically handles well. We thus ported this example to Sapic$^+$, successfully using Tamarin to prove the axiom, and ProVerif to prove with the axiom the expected security property, both being formally combined.

Axioms are also typically how GSVerif [21] functions: its adds axioms about states, axioms that were proven correct by hand. Once again using Sapic$^+$, we proved for the previous 5G example that the axiom generated by GSVerif did hold thanks to Tamarin.

[7]Additonal lemmas and *nounif* instructions.

Conclusion

We introduced Sapic$^+$, a protocol verification platform that allows to transparently use three major verification tools, ProVerif, Tamarin and DeepSec, to efficiently verify protocols from a single protocol model. The translations are carefully optimized and proven correct, and we developed novel case studies of real life protocols. For ease-of-use, we made the entire tool chain available on Docker Hub [1].

We plan to generalize some of our encodings (notably permitting destructors to occur anywhere in the process) and to devise new optimizations, in particular when exporting stateful protocols to ProVerif. We see potential in helping ProVerif terminate by deriving lemmas from a static analysis of the process.

References

[1] Sapic+ tool chain. `https://hub.docker.com/r/robertkuennemann/sapicplusplatform`.

[2] Tamarin (develop). `https://github.com/tamarin-prover/tamarin-prover`.

[3] Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark Dermot Ryan. Statverif: Verification of stateful processes. *J. Comput. Secur.*, 22(5):743–821, 2014.

[4] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, et al. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*, volume 7214 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2012.

[5] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[6] Michael Backes, Jannik Dreier, Steve Kremer, and Robert Künnemann. A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 76–91. IEEE, 2017.

[7] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *42nd IEEE Symposium on Security and Privacy (SP 2021)*, pages 537–554. IEEE, 2021.

[8] David Baelde, Stéphanie Delaune, and Solène Moreau. A method for proving unlinkability of stateful protocols. In *33rd IEEE Computer Security Foundations Symposium (CSF 2020)*, pages 169–183. IEEE, 2020.

[9] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy (SP 2021)*, pages 777–795. IEEE, May 2021.

[10] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *IEEE European Symposium on Security and Privacy (EuroS&P 2016)*, pages 245–260. IEEE, 2016.

[11] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*, pages 1383–1396. ACM, 2018.

[12] David A. Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2015)*, pages 1144–1155. ACM, 2015.

[13] David A. Basin, Michel Keller, Sasa Radomirovic, and Ralf Sasse. Alice and bob meet equational theories. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 160–180. Springer, 2015.

[14] David A. Basin, Sasa Radomirovic, and Lara Schmid. Modeling Human Errors in Security Protocols. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 325–340. IEEE Computer Society.

[15] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502. IEEE Computer Society, 2017.

[16] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Computer Security Foundations Workshop*, pages 82–96. IEEE Comp. Soc.

[17] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebraic Methods Program.*, 75(1):3–51, 2008.

[18] Bruno Blanchet, Vincent Cheval, and Cortier Véronique. Proverif with lemmas, induction, fast subsumption, and much more. In *Proceedings of the 43th IEEE Symposium on Security and Privacy (S&P'22)*. IEEE Computer Society Press, May 2022.

[19] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations and application to SSH. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 4(1):4–31, 2013.

[20] Sofia Celi, Alex Davidson, and Armando Faz-Hernández. Privacy Pass Protocol Specification. Internet-Draft draft-ietf-privacypass-protocol-01, Internet Engineering Task Force, February 2021. Work in Progress.

[21] Vincent Cheval, Véronique Cortier, and Mathieu Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in proverif. In *31st IEEE Computer Security Foundations Symposium (CSF 2018)*, pages 344–358. IEEE Computer Society, 2018.

[22] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. DEEPSEC: deciding equivalence properties in security protocols theory and practice. In *2018 IEEE Symposium on Security and Privacy (SP 2018)*, pages 529–546. IEEE Computer Society, 2018.

[23] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[24] Véronique Cortier, David Galindo, and Mathieu Turuani. A formal analysis of the neuchatel e-voting protocol. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P 2018)*, pages 430–442. IEEE, 2018.

[25] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1773–1788. ACM.

[26] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. A formal analysis of IEEE 802.11's WPA2: Countering the kracks caused by cracking the counters. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1–17. USENIX Association.

[27] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.

[28] Jannik Dreier, Charles Duménil, Steve Kremer, and Ralf Sasse. Beyond subterm-convergent equational theories in automated verification of stateful protocols. In *Principles of Security and Trust - 6th International Conference (POST 2017)*, volume 10204 of *Lecture Notes in Computer Science*, pages 117–140. Springer, 2017.

[29] PKI Task Force. PKI for machine readable travel documents offering ICC read-only access. Technical report, International Civil Aviation Organization, 2004.

[30] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David A. Basin. A spectral analysis of noise: A comprehensive, automated, formal analysis of diffie-hellman protocols. In *29th USENIX Security Symposium (USENIX Security 2020)*, pages 1857–1874. USENIX Association, 2020.

[31] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for unbounded verification of privacy-type properties. *J. Comput. Secur.*, 27(3):277–342, 2019.

[32] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In Ruby B. Lee and Weidong Shi, editors, *The Second Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, page 11. ACM, 2013.

[33] Charlie Jacomme, Steve Kremer, and Guillaume Scerri. Symbolic models for isolated execution environments. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 530–545. IEEE, 2017.

[34] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 435–450. IEEE, 2017.

[35] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise explorer: Fully automated modeling and verification for arbitrary noise protocols. In *IEEE European Symposium on Security and Privacy (EuroS&P 2019)*, pages 356–370. IEEE, 2019.

[36] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, pages 163–178, San Jose, CA, USA, May 2014. IEEE Computer Society Press.

[37] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *J. Comput. Secur.*, 24(5):583–616, 2016.

[38] Robert Künnemann, Ilkan Esiyok, and Michael Backes. Automated verification of accountability in security protocols. In *32nd IEEE Computer Security Foundations Symposium (CSF 2019)*, pages 397–413. IEEE, 2019.

[39] Pascal Lafourcade and Maxime Puys. Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In *Foundations and Practice of Security - 8th International Symposium, (FPS 2015)*, volume 9482 of *Lecture Notes in Computer Science*, pages 137–155. Springer, 2015.

[40] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Softw. Concepts Tools*, 17(3):93–102, 1996.

[41] Sjouke Mauw, Zach Smith, Jorge Toro-Pozo, and Rolando Trujillo-Rasua. Distance-Bounding Protocols: Verification without Time and Location. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 549–566. IEEE Computer Society.

[42] Karl Norrman, Vaishnavi Sundararajan, and Alessandro Bruni. Formal analysis of EDHOC key establishment for constrained iot devices. In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 210–221. SCITEPRESS, 2021.

[43] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. AVATAR: A sysml environment for the formal verification of safety and security properties. In *11th Annual International Conference on New Technologies of Distributed Systems (NOTERE 2011)*, pages 1–10. IEEE, 2011.

[44] Benedikt Schmidt, Simon Meier, Cas Cremers, and David A. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *25th IEEE Computer Security Foundations Symposium (CSF 2012)*, pages 78–94. IEEE Computer Society, 2012.

[45] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS 20)*, pages 1461–1480. ACM, 2020.

[46] Göran Selander, John Preuß Mattsson, and Francesca Palombini. Ephemeral Diffie-Hellman Over COSE (EDHOC). Internet-Draft draft-ietf-lake-edhoc-07, Internet Engineering Task Force, May 2021. Work in Progress.

[47] Tatu Ylonen and Chris Lonvick. The Secure Shell (SSH) Transport Layer Protocol. `https://tools.ietf.org/html/rfc4253`.

A Docker instructions

These instructions define how to run the docker image.

Docker installations

- Linux: follow instructions at `https://docs.docker.com/get-docker/`
- MacOS, there is a binary docker for Mac, which can be installed as a package (`https://docs.docker.com/docker-for-mac/`), or, if you have homebrew, via 'brew cask install docker'.

Sapic$^+$ image The docker image that automatically installs all the required tool and provide an easy access to Sapic$^+$ is available on dockerhub[1].

After the installation of docker, the ready to use image can be pulled in a few seconds with the following command:

```
user@PC  $ docker pull protocolplatform/protocolplatform:latest
latest: Pulling from protocolplatform/protocolplatform
69692152171a: Pull complete
205f84d8d925: Pull complete
4f4fb700ef54: Pull complete
d40febe920a4: Pull complete
951aab113191: Pull complete
1ce93dea7c43: Pull complete
756e36a6a0ee: Pull complete
578a009fd5f1: Pull complete
0d7b23b7972f: Pull complete
2e58ea6824e9: Pull complete
5edd7e3a65e5: Pull complete
c08c877fcea6: Pull complete
59ceebc22d18: Pull complete
9b00f0afd6bb: Pull complete
f3379feff7e7: Pull complete
7ed5ec83d93c: Pull complete
gest: sha256:540d7197a065bc6979289d462873e0c76a28f2bfac05a5e1e94853ed1e9b38d1
Status: Downloaded newer image for protocolplatform/protocolplatform:latest
docker.io/protocolplatform/protocolplatform:latest
```

To launch the docker, execute the command `$ docker run protocolplatform/protocolplatform:latest`, which should display a README with instructions on how to use the tools.

Re-building the image This is only needed for regenerating the image when we update the software. The docker building files are directly included inside the tamarin-prover repository[2].

1. From the root of the repository, the docker image is built with `$./etc/docker/build.sh`.
2. Once the main image is built, one can also run the case studies of the paper with `$./etc/docker/build-benchs.sh`.

B Benchmarks

See Figures 6 and 7.

C Pitfalls when carrying over models

We already discussed the following differences:

- ProVerif and Tamarin do not allow to make statements about the attacker knowledge in a completely equivalent manner.

	case study	w/o compr.	w/ compr.	ratio
[32, 33]	`OTP`	25	8	3
[47]	`SSH-NEST`	594	316	2
	`U2F-TOY`	∞	30	∞

Figure 6: Benchmarks for the secret channel optimization compression: running time (in seconds)

	case study	state restr.	state facts	ratio
[10, 33]	`AC`	127	1	127
[10, 33]	`AC-F-sid`	64	250	$\frac{1}{4}$
[10, 33]	`AC-F-counter`	266	4	66
[3, 37]	`SD`	47	∞	$\frac{1}{\infty}$

Figure 7: Benchmarks for the state facts option: running time (in seconds)

- Some equational theories in one tool only have approximate equivalents in the other.

The differences due to implementations choices are the following ones.

Pattern matching and bindings TAMARIN supports general pattern matchings, where the pattern $\langle x, x, y\rangle$ will for instance expect twice the same value for x, and if y is previously bound, y would need to match this bound value.

PROVERIF supports pattern matchings, with an explicit $=$ to indicate that a variable should not be rebound but match the current value of the variable. For example, the pattern $\langle x, x, y\rangle$ will accept three distinct values, and rebind x to the second value. $\langle = x, x, y\rangle$ will expect the first value to be the previous binding of x, and then bind x to the second value.

We believe that such subtle behavior may be confusing, and therefore forbid rebinding of variables, and thus multiple occurrences of the same variable in a pattern matching. This syntactic restriction avoids confusion about the possible semantics, and enforces the behavior of PROVERIF and TAMARIN to coincide.

Destructors in conditionals In PROVERIF a conditional $\mathsf{if}\ \ t_1 = t_2\ \mathsf{then}\ \ P\ \mathsf{else}\ \ Q$ will

- transition to P if $\mathsf{Msg}(t_1)$, $\mathsf{Msg}(t_2)$ and $t_1 =_E t_2$,
- transition to Q if $\mathsf{Msg}(t_1)$, $\mathsf{Msg}(t_2)$ and $t_1 \neq_E t_2$, and
- block if either $\neg\mathsf{Msg}(t_1)$ or $\neg\mathsf{Msg}(t_2)$, that is if a destructor fails.

In SAPIC$^+$ (as in DEEPSEC) we prefer the condition for entering the else branch to be the negation of the condition for entering the if branch, i.e., $\neg\mathsf{Msg}(t_1)$, or $\neg\mathsf{Msg}(t_2)$ or $t_1 \neq_E t_2$. Note that this does not decrease expressivity as the behaviour of PROVERIF can be encoded as $\mathsf{let}\ \ x_1 = t_1\ \mathsf{in}\ \mathsf{let}\ x_2 = t_2\ \mathsf{in}\ \mathsf{if}\ \ x_1 = x_2\ \ \mathsf{then}\ \ P\ \mathsf{else}\ \ Q$. To ensure consistent semantics we actually translate the conditional $\mathsf{if}\ \ t_1 = t_2\ \ \mathsf{then}\ \ P\ \mathsf{else}\ \ Q$ in SAPIC$^+$ to $\mathsf{let}\ \ (= t_1) = t_2\ \mathsf{in}\ P\ \mathsf{else}\ \ Q$ in PROVERIF. If general conditional formulae occur in the process, we need to redefine in PROVERIF an equality function `eq'` consistent with our semantics that we use only in the conditional branchings formulas:

```
fun eq'(bitstring,bitstring):bool
reduc
  forall x:bitstring; eq'(x,x) = true
  otherwise forall x: bitstring or fail, y: bitstring or fail; eq'(x,y) = false.
```

Pairs In TAMARIN, tuples are syntactic sugar for right nested tuples, i.e., $\langle x, y, z\rangle = \langle z, \langle y, z\rangle\rangle >$. In PROVERIF, there is a dedicated function symbol for each possible length of tuples, with corresponding projections. Then, $\langle x, y, z\rangle$ corresponds to $(x, y, z)_3$.

This small difference actually implies distinct behaviours for let bindings or pattern matching in inputs.. For instance, consider let $\langle x, y, z\rangle = m$ in $\cdots$ In the SAPIC$^+$ style, this pattern would accept $\langle m, n, s, t\rangle$, as it would correspond to $\langle m, \langle n, \langle s, t\rangle\rangle\rangle$, and we would have $z = \langle s, t\rangle$. In PROVERIF, this would only accept tuples of exactly length 3, and a four tuple would be rejected. This can lead to very different behaviour for a protocol, and both remove attacks or create false attacks.

To ensure a sound export, we translate tuples in SAPIC$^+$ as nested pairs in PROVERIF.

Matching in inputs The pattern matching inside an input behaves differently in PROVERIF and SAPIC. For PROVERIF, this is only a syntactic sugar where given a pattern p, in(t, p); P corresponds to in(t, x); let $p = x$ in P else 0. Notably, the input is always executable, and the continuation may fail. The semantics of SAPIC is different, as the matching is directly made in the input, and the input is not always executable.

We defined SAPIC$^+$ following the behaviour of SAPIC$^+$, as it was important to obtain the same behaviours for the equivalence semantics of PROVERIF and DEEPSEC.

D Syntactic extensions

With SAPIC$^+$, we make several small syntactic extensions to SAPIC, importing some useful features from PROVERIF.

Types SAPIC$^+$ provides some small typing capabilities, that can be used to sanity check the model. Function symbols can be declared with a type, and variables can also be bound with a type:

```
functions: enc(bitstring,skey):bitstring

new sk:skey; in(m:bitstring); out(enc(m,sk)).
```

A type inference algorithms allows both to sanity check the models, looking for incompatible typing annotations, and to ease the export to PROVERIF. Notably, events in PROVERIF need to be declared with the type of their arguments, that we can infer from the annotation in the process and the event usages.

Processes with explicit parameters SAPIC allows to declare sub-processes that can be called inside the main process with the following syntax.

```
let P = ...

let Q = ...

process (new skP; new skQ; !P | !Q)
```

Such declarations can lead to mistakes in the model. Indeed, assuming that `P` should only have access to `skP` and `pk(skQ)`, we may use inside it by mistake `skQ`. SAPIC$^+$ now supports the declaration of sub-processes with some explicit parameters, that are binding the sub-variables, improving the readability of the models.

```
let P(skP,pkQ) = ...

let Q(skQ,pkP) = ...

process (new skP; new skQ; !P(skP,pk(skQ)) | !Q(skQ,pk(skP))
```

Explicit pattern matchings SAPIC supports the same pattern matching capabilities as TAMARIN. Thus, one can write processes of the form `new sk; new token; P(sk) | in(enc(< token, m>,sk))`. In big processes, implicit pattern matchings can make the model difficult to verify and understand. We thus add the capability to specify which variables correspond to a pattern match, and which variable are to be bound. We use the same notation as PROVERIF, adding as a prefix the $=$ symbol, for instance yielding `new sk; P(sk) | in(enc(< =token, m>,=sk))`.

Remark that to avoid modelling mistakes, we recommend that modellers do not use pattern matchings over function symbols, but rather rely on let bindings: `new sk; P(sk) | in(cypher); let <=token,m> = dec(cypher,sk) in ...)`

E Sapic$^+$ to Tamarin

We first recall the syntax and semantics of labelled multiset rewriting rules, which constitute the input language of the TAMARIN tool [44].

Definition 5 (Multiset rewrite rule)**.** *A labelled multiset rewrite rule ri is a triple (l, a, r), $l, a, r \in \mathcal{F}^*$, written $l \dashv[\ a\ ]\!\rightarrow r$. We call $l = prems(ri)$ the premises, $a = actions(ri)$ the actions, and $r = conclusions(ri)$ the conclusions of the rule.*

Definition 6 (Labelled multiset rewriting system)**.** *A labelled multiset rewriting system is a set of labelled multiset rewrite rules R, such that each rule $l \dashv[\ a\ ]\!\rightarrow r \in R$ 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 $l' \dashv[\ a'\ ]\!\rightarrow r' \in_E ginsts(l \dashv[\ a\ ]\!\rightarrow r)$ 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

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

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

Definition 7 (Labelled transition relation)**.** *Given a multiset rewriting system R we define the* labeled transition relation *$\rightarrow_R \subseteq \mathcal{G}^\# \times \mathcal{P}(\mathcal{G}) \times \mathcal{G}^\#$ as*

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

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

Definition 8 (Executions)**.** *Given a multiset rewriting system R we define its set of executions as*

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

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

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

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.

Definition 10 (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, Event, InEvent, $Pred_{pr}$, $Pred\_not_{pr}$ $\mid pr \in \Sigma_{pred}$ $\}$.

Next we define the *hiding* operation which removes all reserved facts from a trace.

Definition 11 (*hide*)**.** *Given a trace tr and a set of facts F we inductively define hide([]) = [] and*

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

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

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

E.1 Sapic$^+$ extensions

E.1.1 Let bindings with pattern matching

The original Sapic translation actually allows writing MSR rules directly inside processes. We use this feature to encode let bindings in a black box fashion w.r.t. to the proof of the Sapic translation to Tamarin.

Definition 12. *Let P be a Sapic$^+$ process. We define the translation $[\![P]\!]^{let}$ as follows:*

$$\begin{aligned}[\![\mathsf{let}\ s = t\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]^{let} := \quad & \mathsf{new}\ b; [] \dashv[]\mapsto [\mathsf{Let}(t,b)]; \\ & ([\mathsf{Let}(s,b)] \dashv[]\mapsto []; [\![P]\!]^{let}) \\ & |([\mathsf{Let}(x,b)] \dashv[\ \mathit{restr}(\forall fv(p).t \neq s)\ ]\mapsto []; [\![Q]\!]^{let})\end{aligned}$$

where x is a fresh variable, and where $[\![P]\!]^{let}$ is lifted homomorphically to the other constructs.

We show that the validity of this encoding by showing that the set of traces is the same on both side.

Lemma 1. *Let $E_D = \emptyset$ and P be a Sapic$^+$ process. We then have:*

$$traces^{pi}_{\vdash}(P) = traces^{pi}_{\vdash}([\![P]\!]^{let})$$

Proof. We first recall the difference between our semantics and the original Sapic semantics, where MSR rules can be inserted inside processes. Sapic$^+$ configurations $(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^\# \{\mathsf{let}\ s = t\ \mathsf{in}\ P\ \mathsf{else}\ Q\}, \sigma, \mathcal{L})$ are extended with a set of ground facts $\mathcal{S}^{\mathrm{MS}} \subset \mathcal{G}$, and the semantics are equipped with the additional rule:

$$\begin{aligned}(\mathcal{E}, \mathcal{S}, \mathcal{S}^{\mathrm{MS}}, \mathcal{P} \cup^\# \{[l] \dashv[\ a\ ]\mapsto [r]; P\}, \sigma, \mathcal{L}) \xrightarrow{a'} \quad & (\mathcal{E}, \mathcal{S}, \mathcal{S}^{\mathrm{MS}} \cup^\# r' \setminus lfacts, \mathcal{P} \cup^\# \{P\tau\}, \sigma, \mathcal{L}) \\ & \text{if } \exists \tau, l', a', r'.\ \tau \text{ is grounding for } [l] \dashv[\ a\ ]\mapsto [r], \\ & [l'] \dashv[\ a'\ ]\mapsto [r'] =_E ([l] \dashv[\ a\ ]\mapsto [r])\tau, \\ & lfacts(l') \subset^\# \mathcal{S}^{\mathrm{MS}}, pfacts(l') \subset^\# \mathcal{S}^{\mathrm{MS}} \\ & \forall \mathsf{restr}(\phi) \in a', \phi \text{ holds}\end{aligned}$$

Remark that we allow rules to be labelled with restriction of the form $\mathsf{restr}(\phi)$, where the rule can be executed if and only if ϕ is true.

In this semantics, we have that:

$$\begin{array}{rcl}(\mathcal{E},\mathcal{S},\mathcal{S}^{\mathrm{MS}},\{[\![\mathsf{let}\ s=t\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]^{\mathtt{let}}\},\sigma,\mathcal{L}) & \to & (\mathcal{E}\cup b',\mathcal{S},\mathcal{S}^{\mathrm{MS}},R_1,\sigma,\mathcal{L})\\ & \to & (\mathcal{E}\cup b',\mathcal{S},\mathcal{S}^{\mathrm{MS}}\cup \mathsf{Let}(t,b'),R_1,\sigma,\mathcal{L})\end{array}$$

where b' is a fresh name and

$$\begin{array}{rl} & [] \,-[]\!\to [\mathsf{Let}(t,b')];\\ R_1 := & ([\mathsf{Let}(s,b')] \,-[]\!\to [];[\![P]\!]^{\mathtt{let}})\\ & |([\mathsf{Let}(x,b')] \,-[\ \mathsf{restr}(\forall fv(s).t\neq s)\ ]\!\to [];[\![Q]\!]^{\mathtt{let}})\end{array}$$

Now, we remark that in this setting $\exists \tau, l', a', r'$ such that τ is grounding for $[\mathsf{Let}(s,b')] -[]\!\to []$ and $[l'] -[\ a'\ ]\!\to [r'] =_E ([\mathsf{Let}(s,b')] -[]\!\to [])\tau$ and $\mathit{lfacts}(l') \subset^{\#} \mathcal{S}^{\mathrm{MS}}$ is equivalent to $\exists \tau. s\tau =_E t$. This is notably due to the fact that b' is fresh, and thus only $\mathsf{Let}(t,b') \in \mathcal{S}^{\mathrm{MS}}$ can be matched on the left hand side. And further, we have that $\exists \tau. t =_E s\tau \Leftrightarrow \neg(\forall fv(s).t \neq s)$.

So, there are only two possibilities:

$$\begin{array}{rcl}(\mathcal{E}\cup b',\mathcal{S},\mathcal{S}^{\mathrm{MS}}\cup \mathsf{Let}(t,b'),R_1,\sigma,\mathcal{L}) & \to & (\mathcal{E}\cup b',\mathcal{S},\mathcal{S}^{\mathrm{MS}},[\![P]\!]^{\mathtt{let}}\tau,\sigma,\mathcal{L})\\ & & \text{if } \exists\tau.t =_E s\tau\end{array}$$

and

$$\begin{array}{rcl}(\mathcal{E}\cup b',\mathcal{S},\mathcal{S}^{\mathrm{MS}}\cup \mathsf{Let}(t,b'),R_1,\sigma,\mathcal{L}) & \to & (\mathcal{E}\cup b',\mathcal{S},\mathcal{S}^{\mathrm{MS}},[\![Q]\!]^{\mathtt{let}},\sigma,\mathcal{L})\\ & & \text{if for all}\tau.t \neq_E s\tau\end{array}$$

We thus see that the, without other processes in parallel, possible reductions of $[\![\mathsf{let}\ s=t\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]^{\mathtt{let}}$ are exactly the same as the reductions of $\mathsf{let}\ s=t\ \mathsf{in}\ P\ \mathsf{else}\ Q$. We can immediately conclude here that $\mathit{traces}^{pi}_{\vdash}(P) \subset \mathit{traces}^{pi}_{\vdash}([\![P]\!]^{\mathtt{let}})$. For the other direction, we remark that even though we split a single transition into two transitions, both are silents and do not influence and cannot be influenced by the execution in parallel of other processes. This is due to the fresh name inserted inside the facts. We thus also have that $\mathit{traces}^{pi}_{\vdash}(P) \supset \mathit{traces}^{pi}_{\vdash}([\![P]\!]^{\mathtt{let}})$.

□

Reusing the translation from SAPIC to TAMARIN, described in [36, Fig. 4] and that we denote by $[\![]\!]^{\mathrm{TAM}}$, we can build a translation from SAPIC$^+$ to TAMARIN that preserves the set of traces and thus the validity of the formulas. This translation also adds the restrictions required to make the translation correct.

Theorem 3. *Let P be a process without destructor symbols, and φ a formula without* In *and* Out *facts. We have that:*

$$P \vDash \varphi \Leftrightarrow [\![[\![P]\!]^{let}]\!]^{\mathrm{TAM}} \vDash [\![\varphi]\!]^{\mathrm{TAM}}$$

Proof. After using the trace preserving encoding of Lemma 1, we conclude by using Theorem 1 of [36], that we can leverage as we now have a process that only contains constructs of the SAPIC language. □

E.1.2 Let bindings with destructor

Definition 13. *Let* dest *be a destructor symbol with an associated rule of the form $\mathit{dest}(\vec{u}) \to x \in E_D$.*

We define $[\![P]\!]^{dest}$ as:

$$[\![\mathsf{let}\ x = \mathit{dest}(\vec{t})\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]^{dest} := \mathsf{let}\ \vec{u} = \vec{t}\ \mathsf{in}\ [\![P]\!]^{dest}\ \mathsf{else}\ [\![Q]\!]^{dest}$$

if all variables in $\vec{u}$ are fresh except for x, and where $[\![P]\!]^{dest}$ is inductively propagated.

To show the correction of the encoding, we simply show the validity of the syntactic sugar.

Lemma 2.

$$traces^{pi}(P) = traces^{pi}([\![P]\!]^{dest})$$

Proof. By definition of the application of a rewrite rule to a term, which can only be applied if we can unify the argument of the destructor with the arguments of the rewrite rule, we have that:

$$\exists \tau. x\tau =_E \mathsf{dest}(\vec{t}) \\ \Leftrightarrow \\ \exists \tau. \vec{u}\tau =_E \vec{t}$$

This directly implies the equivalence of the executability condition of the two let bindings in $[\![P]\!]^{\mathsf{dest}}$. □

Corollary 1. *Let P be a process with destructor symbols only as head function symbols of right hand side term of* let *constructs, and φ a formula without* In *and* Out *facts. We have that:*

$$P \vDash \varphi \Leftrightarrow [\![[\![[\![P]\!]^{dest}]\!]^{let}]\!]^{\textsc{Tam}} \vDash [\![\varphi]\!]^{\textsc{Tam}}$$

Proof. Direct application of Theorem 1 of [36], Lemma 1 and Lemma 2. □

E.2 Sapic optimizations

E.2.1 Path Compression

Some rules produced by the original Sapic translation can be compressed, for instance:

$$[s_1, \mathsf{Fr}(x)] -[]\rightarrow [s_2] \cup [s_2, \mathsf{Fr}(y)] -[]\rightarrow [s_3] \\ \Rightarrow [s_1, \mathsf{Fr}(x), Fr(y)] -[]\rightarrow [s_3]$$

We try to provide a general algorithm to compress as much as possible the set of rules produced by Sapic$^+$, and thus improve the performances.

We must consider in which cases compressing two rules, one of which produces some fact and one consumes it, does not alter the possible set of traces. We provide bellow a general algorithm to compress a set of rules of the form

$$[\{s_1\} \cup^{\#} Fr_1 \cup^{\#} In_1] -[\ E_1\ ]\rightarrow [R_1]$$

where intuitively s_1 is the control state of the protocol, and multisets of fresh facts Fr_1 and input facts In_1 may be required. We first express, in an algorithmic form, a condition under which two rules can be merged. In this Section, we actually refer to E_1 for $hide(E_1)$.

```
canMerge $r_1$ $r_2$ :=
    let $[P_1] -[\ E_1\ ]\rightarrow [R_1] = r_1$ in
    let $[P_2] -[\ E_2\ ]\rightarrow [R_2] = r_2$ in
    if $r1$ or $r2$ contain any persistent state then
        False
    if $E_1 \neq \emptyset$ and $E_2 \neq \emptyset$ then  (* we cannot merge rules if it makes events be simulataneous *)
        False
    else if $Out \in R_1$ and $In \in P_2$ then (*  we cannot merge rules if the first one give knowledge that
      may be needed for the other one *)
        False
    else if $\#R_1 > 1$ and  $E_2 \neq \emptyset$ then (*  we cannot merge rules if we are breaking asynchronous
     behavior (i.e u->v,w, and v-E->t cannot be compressed, else an event that could have happened
     with w before E cannot do so anymore. *)
        False
    else if $E_1 \neq \emptyset$ and  $In \in P_2$ then (* we cannot merge rules if the second rule needs some input,
     that may be computable by the attacker only after the first rule raised an event. *)
        False

     else
         True
```

Lemma 3. *For any two rules*

$$l := [\{s_1\} \cup^{\#} Fr_1 \cup^{\#} In_1] \dashv\!\![\ E_1\ ]\!\!\rightarrow [R_1] \quad \text{and} \quad r := [\{s_2\} \cup^{\#} Fr_2 \cup^{\#} In_2] \dashv\!\![\ E_2\ ]\!\!\rightarrow [R_2]$$

and any set of facts F such that $s_2 \in R_1$, $s_2 \neq s_1$, s_1 and s_2 not persistent and $s_2 \notin F$,

$$\text{if} \quad \texttt{canMerge}\ l\ r \quad \text{then} \quad traces^{msr}(F \cup l \cup r)) = traces^{msr}(F \cup m)$$

where $m := [\{s_1\} \cup^{\#} Fr_1 \cup^{\#} Fr_2 \cup^{\#} In_1 \cup^{\#} In_2] \dashv\!\![\ E_1 \cup E_2\ ]\!\!\rightarrow [R_1 \cup R_2 \setminus \{s_2\}]$

Proof. Note that $s_2 \notin F$, so l is the only possible source of s_2 and l each application of r must be preceded by a different application of l. We assume that `canMerge` $l\ r$ holds, and thus have that:

$$\begin{array}{rll} & E_1 = \emptyset \vee E_2 = \emptyset & (i) \\ \wedge & Out \notin R_1 \vee In \notin P_2 & (ii) \\ \wedge & R_1 = \{s_2\} \vee E_2 = \emptyset & (iii) \\ \wedge & E_1 = \emptyset \vee In \notin P_2 & (iv) \end{array}$$

First, we prove that:

$$traces^{msr}(F \cup m) \subseteq traces^{msr}(F \cup l \cup r)$$

Indeed, any application of m can be replaced by an application of l directly followed by an application of r, as for any facts set S such that $lfacts(m) \subseteq^{\#} S$:

$$S \xrightarrow{E_1 \cup E_2}_m S \cup^{\#} R_1 \cup^{\#} R_2 \setminus (\{s_1\} \cup^{\#} \{s_2\} \cup^{\#} Fr_1 \cup^{\#} Fr_2 \cup^{\#} In_1 \cup^{\#} In_2)$$

is replaced by

$$\begin{array}{ll} S & \xrightarrow{E_1}_l S \cup^{\#} R_1 \setminus (\{s_1\} \cup^{\#} Fr_1 \cup^{\#} In_1) \\ & \xrightarrow{E_2}_r S \cup^{\#} R_1 \cup^{\#} R_2 \setminus (\{s_1\} \cup^{\#} Fr_1 \cup^{\#} In_1 \cup^{\#} \{s_2\} \cup^{\#} Fr_2 \cup^{\#} In_2) \end{array}$$

By (i), we have that either $E_1 = \emptyset$ or $E_2 = \emptyset$, which allows us to conclude.

Second, we prove that:

$$hide(traces^{msr}(F \cup l \cup r)) \subseteq hide(traces^{msr}(F \cup m)$$

First, remark that as $s_2 \notin F$, for any execution $e \in exec^{msr}(F \cup l \cup r)$, there are more applications of l than r.

- Let us consider the case where there is no r.
  - If $E_1 \neq \emptyset$, then, we can replace all occurrences of l by m (when l is executable, so is m), and it yields the same trace (as $E_2 = \emptyset$ by (i)).
  - If $E_1 = \emptyset \wedge E_2 = \emptyset$, then, we can also replace all occurrences of l by m.
  - If $E_1 = \emptyset \wedge E_2 \neq \emptyset$, then by (iii), $R_1 = s_2$. And thus, l does not produce anything which is used in the trace, and we can simply remove all occurrences of l and preserve the executability of the trace.
- we now consider the last occurrence of r, along with its closest occurrence l.
  - If $E_2 = \emptyset \wedge E_1 \neq \emptyset$, we remove both l and r and insert m at the position of l in the execution.
    * If $\#P_2 \leq 1$, then $P_2 = \{s_2\}$, and m can be executed at the position of l.
    * If $\#P_2 > 1 \wedge Out \in R_1$, the difficulty would be if $In_2 \in P_2$, but by $E_1 \neq \emptyset$ and (iv) this is impossible, and m can be executed at the position of l.

– If $E_2 = \emptyset \wedge E_1 = \emptyset \wedge \mathit{Out} \in R_1$, we remove both l and r and insert m at the position of l in the execution. $\mathit{Out} \in R_1$ implies that $\mathit{In} \notin P_2$ by (ii), thus $P_2 = \{s_2\} \cup^{\#} \mathit{Fr}_2$. Hence, m can be executed at the position of l.

– If $E_2 = \emptyset \wedge E_1 = \emptyset \wedge \mathit{Out} \notin R_1 \wedge \mathit{In} \in P_2$, we remove both l and r and insert m at the position of r in the execution. As $\mathit{Out} \notin R_1$, the application of l is useless for all rules between l and r, and we can indeed apply m at r and maintain the executability of the actions before.

– If $E_2 = \emptyset \wedge E_1 = \emptyset \wedge \mathit{Out} \notin R_1 \wedge \mathit{In} \notin P_2$, we remove both l and r and insert m at the position of l. m is indeed executable at this place, as P_2 only contains s_2 and fresh facts.

– If $E_2 \neq \emptyset$, then we remove both l and r and insert m at the position of r in the execution. By (i), we have that $E_1 = \emptyset$, and by (iii) we have that $R_1 = \{s_1\}$. Thus, l is not required by any rules in between l and r, and we can indeed remove its execution, and only execute m instead of r.

□

We can thus apply the optimization on the set of MSR rules produced by the SAPIC translation, and we preserve the set of traces. Remark that for translating the inputs, SAPIC may behave in two different ways, depending on the lemmas that we have in the file:

- in all generality, and input is translated with an MSR rule labelled with an event $InEvent$, which is used in the α_{inev} restriction([37]).
- under some conditions, see [6, Appendix E], the α_{inev} restriction is removed, and thus, the translation of input is with a rule that does not contain any event.

In the first case, two consecutive inputs will not be merged into a single rule, as we have events inside the labels, while in the second case, it is possible.

E.2.2 Secret channel optimization

When a channel is completely secret, we can simplify the translation, by essentially removing the rule that allows the attacker to intercept the messages, and making the communication through a dedicated fact. We can define the set of secret channels, by looking syntactically if the identifier of channel is a name that does not occur anywhere else.

Definition 14. *Let P be a process. We denote by $\mathtt{SecretChannels}(P)$ the set of names in $\mathcal{N}_{priv}$ such that for any $n \in \mathtt{SecretChannels}(P)$,*

- *there is a single occurrence of the form* new n*,*
- *all other occurences of n in P are either of the form* $\mathsf{in}(n, M)$ *or* $\mathsf{out}(n, M)$ *where n does not occur in M.*

We then perform a simplification of the translation of SAPIC from processes to MSR. We optimize the translation by removing the rules of the translation that corresponding to attacker communications, as the channel is secret.

Definition 15. *We denote by $[\![]\!]^{\mathrm{TAM}}$ the translation of* SAPIC *defined in Figure 7 of [36]. Let P be a protocol, we define $[\![]\!]^{T,secret}_P$ as $[\![]\!]^{\mathrm{TAM}}$ for which we change the definition on the* in *and* out *construct as follows when $n \in \mathtt{SecretChannels}(P)$:*

$$\begin{array}{rcl}
[\![\mathsf{out}(n,N);Q,p,\tilde{x}]\!]^{T,secret}_P & = & [\mathsf{state}_p(\tilde{x})] \to [\mathsf{Msg}(n,N), \mathsf{state}^{\mathsf{semi}}_p(\tilde{x})], \\
 & & [\mathsf{state}^{\mathsf{semi}}_p(\tilde{x}), \mathsf{Ack}(n,N)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x})] \ \cup [\![P, p\cdot 1, \tilde{x}]\!]^{\mathrm{TAM}} \\
[\![\mathsf{in}(n,N);P,p,\tilde{x}]\!]^{T,secret}_P & = & [\mathsf{state}_p(\tilde{x}), \mathsf{In}(\langle n,N\rangle)] \ -\!\![\ \mathrm{InEvent}(\langle n,N\rangle)\ ]\!\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup vars(N))], \\
 & & [\mathsf{state}_p(\tilde{x}), \mathsf{Msg}(n,N)] \to [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup vars(N)), \mathsf{Ack}(n,N)] \\
 & & \cup [\![P, p\cdot 1, \tilde{x} \cup vars(N)]\!]^{\mathrm{TAM}}
\end{array}$$

For this modification we show that the soundness of the translation still holds, i.e. that Lemma 1 of [36] still holds.

Lemma 4. *Let P be a protocol that does not contain any let bindings and destructors.*

$$traces^{pi}_{\vdash}(P) = hide(traces^{msr}([\![P, \emptyset]\!]_P^{\texttt{T,secret}}))$$

Proof. For $n \in \texttt{SecretChannels}(P)$, the difference between the set $[\![P, \emptyset]\!]^{\text{TAM}}$ and the set $[\![P, \emptyset]\!]_P^{\texttt{T,secret}}$, is that we removed rules of the form:

$$[\mathsf{state}_p(\tilde{x}), \mathsf{In}(n)] \dashv[\ \text{InEvent}(n)\ ]\!\rightarrow [\mathsf{Out}(N), \mathsf{state}_{p\cdot 1}(\tilde{x})]$$
$$[\mathsf{state}_p(\tilde{x}), \mathsf{In}(\langle n, N\rangle)] \dashv[\ \text{InEvent}(\langle n, N\rangle)\ ]\!\rightarrow [\mathsf{state}_{p\cdot 1}(\tilde{x} \cup vars(N))]$$

We remark that rules of the previous form can never be executed. Indeed, as n only occurs in channel name position, it never occurs inside a K fact. And thus, facts of the form $\mathsf{In}(n)$ or $\mathsf{In}(\langle n, N\rangle)$ can never be produced. We then easily conclude that $traces^{msr}([\![P, \emptyset]\!]^{\text{TAM}}) = traces^{msr}([\![P, \emptyset]\!]_P^{\texttt{T,secret}})$, and with Lemma 1 of [36]. $\square$

E.2.3 State encoding optimization

We show the soundness of encoding states with secret channels when the state is only accessed in a particular fashion. This is used both for the TAMARIN and the PROVERIF export.

Definition 16. *Let P be a process and cell be a ground term. We say that t is a pure state if*

- *There is a single instance of* $\mathsf{insert}\ t, t'$ *for some t' not under a lock nor replication*
- *All other instances of* $\mathsf{insert}\ t, t'$ *should be followed by an unlock, i.e.* $\mathsf{insert}\ t, t'; \mathsf{unlock}\ t$ *for some t', and should be necessarily preceded by the construct* $\mathsf{lock}\ t; \mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ \_\ \mathsf{else}\ 0)$ *for some x after possibly arbitrary many other actions that do not involve t.*
- *no other state identifier t' occurring in P can be unified with t, i.e., for all substitutions σ, $t \neq t'$ implies $t\sigma \neq_E t'\sigma$.*

Definition 17. *Let P be a process and t a pure state of P. We fix a fresh name $n \in \mathcal{N}_{priv}$ in P. We first define an auxiliary translation function over a process and a set of bound names as:*

$$\begin{array}{lll}
[\![\mathsf{new}\ a; P, \mathcal{E}]\!]_t^{st'} := & \mathsf{new}\ a; \mathsf{new}\ n; [\![P, \mathcal{E} \cup \{a, n\}]\!]_t^{st'} & \textit{if } fn(t) \subset \mathcal{E} \cup \{a\} \\
 & \mathsf{new}\ a; [\![P, \mathcal{E} \cup \{a\}]\!]_t^{st'} & \textit{else} \\
[\![\mathsf{lock}\ t; \mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P, \mathcal{E}]\!]_t^{st'} := & \mathsf{in}(n, x); [\![P, \mathcal{E}]\!]_t^{st'} & \\
[\![\mathsf{insert}\ t, M; \mathsf{unlock}\ t; P, \mathcal{E}]\!]_t^{st'} := & \mathsf{out}(n, M) | [\![P, \mathcal{E}]\!]_t^{st'} & \\
[\![\mathsf{insert}\ t, M; P, \mathcal{E}]\!]_t^{st'} := & \mathsf{out}(n, M) | [\![P, \mathcal{E}]\!]_t^{st'} &
\end{array}$$

and where $[\![P]\!]_t^{st'}$ is propagated inductively.

We then define $[\![P]\!]_t^{st} := [\![P, \emptyset]\!]_t^{st'}$.

Lemma 5. *Let P be a process without destructors symbols with a pure state t.*

$$traces^{pi}_{\vdash}(P) = traces^{pi}_{\vdash}([\![P]\!]_t^{st})$$

Proof. We call n the fixed fresh name used in $[\![P]\!]_t^{\mathsf{st}}$. For the proof, we actually extend $[\![P, \mathcal{E}]\!]_t^{\mathsf{st'}}$ to some cases that do not occur in the normal translation:

$$\begin{array}{ll}
[\![\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P, \mathcal{E}]\!]_t^{\mathsf{st'}} := & \mathsf{in}(n, x); [\![P, \mathcal{E}]\!]_t^{\mathsf{st'}} \\
[\![\mathsf{unlock}\ t; P, \mathcal{E}]\!]_t^{\mathsf{st'}} := & [\![P, \mathcal{E}]\!]_t^{\mathsf{st'}}
\end{array}$$

And we extend its application to multiset of processes. Let us define a bijective mapping f from configurations reachable in P to configurations reachable with the same observable trace in $[\![P]\!]_t^{\mathsf{st}}$.

We show by induction on the last step of the execution that there exists a bijective mapping f (a bi-simulation actually) over configurations such that for all tr, if $(\emptyset, \emptyset, \{P\}, \emptyset, \emptyset) \xRightarrow{tr} \mathcal{C} = (\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$ then $(\emptyset, \emptyset, [\![P]\!]_t^{\mathsf{st}}, \emptyset, \emptyset) \xRightarrow{tr} f(\mathcal{C}) = (\mathcal{E}', \mathcal{S}', \mathcal{P}', \sigma, \mathcal{L}')$ such that,

- $\mathcal{S}' = \mathcal{S} \setminus [t \mapsto M]$;
- $\mathcal{L}' = \mathcal{L} \setminus \{t\}$;
- $\mathcal{E}' = \mathcal{E} \cup \{n\}$ if $fn(t) \subset \mathcal{E}$ and $\mathcal{E}' = \mathcal{E}$ otherwise;
- $\mathcal{P}' \supset [\![\mathcal{P}, \mathcal{E}']\!]_t^{\mathsf{st'}}$
- $\mathsf{out}(n, M) \in \mathcal{P}'$ if and only if $[t \mapsto M] \in StoreA$, and $t \notin \mathcal{L}$ or $\mathsf{unlock}\ t; P \in \mathcal{P}$ or $\mathsf{lookup}\ t\ \mathsf{as}\ x\mathsf{in} P \in \mathcal{P}$.

And conversely if $(\emptyset, \emptyset, [\![P]\!]_t^{\mathsf{st}}, \emptyset, \emptyset) \overset{tr}{\Longrightarrow} \mathcal{C}' = (\mathcal{E}', \mathcal{S}', \mathcal{P}', \sigma, \mathcal{L}')$ then $(\emptyset, \emptyset, \{P\}, \emptyset, \emptyset) \overset{tr}{\Longrightarrow} f^{-1}(\mathcal{C}') = (\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$ with the same conditions.

We first remark that in general, as t cannot be unified with any other state identifier, t uniquely identifies the state across any execution. And by defining n when all the names appearing in t are defined, substituting t by n in all state identifiers preserves the possible equalities between replications of t.

$\Rightarrow$ we first show the left to right implication. If the last step of the execution is a rule that does not involve states, the property trivially holds as the last step of the reduction can trivially be performed on the other side, and we inherit the required hypothesis by induction.

Now, we branch depending on the last rule, that goes from configuration $\mathcal{C}$ to $\mathcal{D}$, and we assume that $P \overset{tr}{\Longrightarrow} \mathcal{D}$ and $[\![P]\!]_t^{\mathsf{st}} \overset{tr}{\Longrightarrow} f(\mathcal{C})$ with the above hypothesis over $f(\mathcal{C})$. We need to construct $f(\mathcal{D}$ such that it meets the expected requirements.

- $(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{insert}\ t, M; P\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}[n \mapsto M], \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L})$
  In this case, by induction hypothesis, there is $[\![\{\mathsf{insert}\ t, N; P\}, \mathcal{E}']\!]_t^{\mathsf{st'}} \in \mathcal{P}'$, and we can perform the reduction:

$$\begin{aligned} &f(\mathcal{C}) = (\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{\mathsf{out}(n, M) | [\![P, \mathcal{E}']\!]_t^{\mathsf{st'}}\}, \sigma', \mathcal{L}') \\ &\quad \longrightarrow (\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{[\![P]\!]_t^{\mathsf{st}}\} \cup^{\#} \{[\![\mathsf{out}(n, M), \mathcal{E}']\!]_t^{\mathsf{st'}}\}, \sigma', \mathcal{L}') = f(\mathcal{D}) \end{aligned}$$

  To show that this new configuration satisfy the induction hypothesis, there are two options. Either it is the lone insert, or it is an insert under the lock.

  - In the first lone insert case, we know that $t \notin \mathcal{L}$ and that $[t \mapsto M] \notin \mathcal{S}$. This new configuration $f(\mathcal{D})$ thus satisfies the induction hypothesis, as we now have $\mathsf{out}(n, M) \in \mathcal{P}'$, and $[n \mapsto M] \in StoreA[n \mapsto M]$ and $t \notin \mathcal{L}$.
  - In the other case, it is an insert under a lock, and we know as the state is pure that we in fact have $P = \mathsf{unlock}\ t; P'$. Thus, the new configuration $f(\mathcal{D})$ satisfies the induction hypothesis, as we now have $\mathsf{out}(n, M) \in \mathcal{P}'$, and $[n \mapsto M] \in StoreA[n \mapsto M]$ and $\mathsf{unlock}\ t; P' \in \mathcal{P}$.

- $(\mathcal{E}, \mathcal{S}[n \mapsto M], \mathcal{P} \cup^{\#} \{\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\{M/x\}\}, \sigma, \mathcal{L})$ By induction hypothesis, we know that $\mathsf{out}(n, M) \in \mathcal{P}'$, and that $[\![\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P, \mathcal{E}']\!]_t^{\mathsf{st'}} = \mathsf{in}(n, x); [\![P, \mathcal{E}']\!]_t^{\mathsf{st'}} \in \mathcal{P}$. we can thus from $f(\mathcal{C})$ perform the following reduction:

$$\begin{aligned} &f(\mathcal{C}) = (\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{\mathsf{out}(n, M)\} \cup^{\#} \{\mathsf{in}(n, x) | [\![P, \mathcal{E}']\!]_t^{\mathsf{st'}}\}, \sigma', \mathcal{L}') \\ &\longrightarrow (\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{[\![P, \mathcal{E}']\!]_t^{\mathsf{st'}}\{M/x\}\}, \sigma', \mathcal{L}') = f(\mathcal{D}) \end{aligned}$$

  As t is a pure state, so $\mathsf{unlock}\ t; P \notin \mathcal{P} \cup^{\#} \{P\{M/x\}\}$ and $\mathsf{lookup}\ t\ \mathsf{as}\ x\mathsf{in}P \notin \mathcal{P} \cup^{\#} \{P\{M/x\}\}$. Further, $t \in \mathcal{L}$, as this lookup must have been preceded by a lock. As we removed the output over n, $\mathsf{out}(n, M) \notin \mathcal{P} \cup^{\#} \{P\{M/x\}\}$, so this new configuration $f(\mathcal{D})$ verifies the induction hypothesis.

- $(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\text{lock } t;\ P\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \cup \{\, t\,\})$

  We have that $t \notin \mathcal{L}$. So, $\mathsf{out}(n, M) \in \mathcal{P}'$ if and only if $[t \mapsto M] \in \mathit{StoreA}$. We simply set $f(\mathcal{D}) = f(\mathcal{C})$. As t is a pure state, we know that $P = \mathsf{lookup}\ t\ \mathsf{as}\ x \mathsf{in} P'$. So, for $f(\mathcal{D})$ we also have $\mathsf{out}(n, M) \in \mathcal{P}'$ if and only if $[t \mapsto M] \in \mathcal{S}$, and thus the induction hypothesis is satisfied.

- $(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{unlock}\ t; P\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \setminus \{\, t\,\})$

  Because t is a pure state, we know that this unlock was preceded by an insert, and thus that $[t \mapsto M] \in \mathcal{S}$. Further, we obviously have that $\mathsf{unlock}\ t; P \in \mathcal{P} \cup^{\#} \{\mathsf{unlock}\ t; P\}$. By induction hypothesis, we know that in $f(\mathcal{C})$, we have that $\mathsf{out}(n, M) \in \mathcal{P}'$. We set $f(\mathcal{D}) = f(\mathcal{C})$. This new configuration satisfy the induction hypothesis, as we know have for $\mathcal{D}$ that $t \notin \mathcal{L} \setminus \{\, t\,\}$ and still have $[t \mapsto M] \in \mathcal{S}$, while $\mathsf{out}(n, M) \in \mathcal{P}'$.

$\Leftarrow$ we now show the right to left implication. If the last step of the execution is a rule that does not involve an internal communication over channel n, or the addition of a $\mathsf{out}(n, M)$ to set multiset, the property trivially holds. There are thus two interesting cases. Now, we branch depending on the last rule, that goes from configuration $\mathcal{C}'$ to $\mathcal{D}'$, and we assume that $[\![P]\!]_t^{\mathsf{st}} \overset{tr}{\Longrightarrow} \mathcal{D}'$ and that $P \overset{tr}{\Longrightarrow} f^{-1}(\mathcal{C}')$ and with the above hypothesis over $\mathcal{C}'$ and $f^{-1}(\mathcal{C}')$.

- $(\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{\mathsf{out}(n, M) | P'\}, \sigma', \mathcal{L}') \quad \longrightarrow \quad (\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{P'\} \cup^{\#} \{\mathsf{out}(n, M)\}, \sigma', \mathcal{L}')$

  We have two options regarding the processes set $\mathcal{P}$ of $f^{-1}(\mathcal{C}')$, which contains the pre-image of the state translation of $\{\mathsf{out}(n, M) | P'\}$. Either $\mathsf{insert}\ t, M; P \in \mathcal{P}$ or $\mathsf{insert}\ t, M; \mathsf{unlock}\ t; P \in \mathcal{P}$. In both cases, we can make the reduction:

  $$f^{-1}(\mathcal{C}') \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}[t \mapsto M], \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \setminus \{t\}) = f^{-1}(\mathcal{D}')$$

  (remark that in one of the two cases, $\mathcal{L} \setminus \{t\} = \mathcal{L}$) This configuration does satisfy the induction hypothesis, as we do have that $\mathsf{out}(n, M) \in \mathcal{P}' \cup^{\#} \{P'\} \cup^{\#} \{\mathsf{out}(n, M)\}$, and $t \notin \mathcal{L} \setminus \{t\}$ and $[t \mapsto M] \in \mathcal{S}[t \mapsto M]$.

- $(\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{\mathsf{in}(n, x); P'\} \cup^{\#} \{\mathsf{out}(n, M)\}, \sigma', \mathcal{L}') \quad \longrightarrow \quad (\mathcal{E}', \mathcal{S}', \mathcal{P}' \cup^{\#} \{P'\{M/x\}\}, \sigma', \mathcal{L}')$

  By definition of the translation the processes set $\mathcal{P}$ of $f^{-1}(\mathcal{C}')$ contains the process $\{\mathsf{lock}\ t; \mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P\}$. Further, as $\{\mathsf{out}(n, M)\}$ occurs in $\mathcal{C}'$, we know by induction hypothesis that for $f^{-1}(\mathcal{C}')$ we have that $[n \mapsto M] \in \mathcal{S}$ and $t \notin \mathcal{L}$. We can thus reduce it with:

  $$f^{-1}(\mathcal{C}') \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}[t \mapsto M], \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L} \cup \{t\}) = f^{-1}(\mathcal{D}')$$

  This new configuration satisfies the induction hypothesis, as $\{\mathsf{out}(n, M)\}$ does occur in $\mathcal{D}'$, and even if $[t \mapsto M] \in \mathcal{S}[t \mapsto M]$, we have $t \in \mathcal{L} \cup \{t\}$, and as t is a pure state, $\mathsf{unlock}\ t; P \notin \mathcal{P}$ (there must be an insert before) and $\mathsf{lookup}\ t\ \mathsf{as}\ x \mathsf{in} P \notin \mathcal{P}$.

We have demonstrated by induction the existence of a mapping satisfying our condition, that allows to show that the set of observable traces tr is indeed the same on both sides. This thus concludes the proof. □

Remark that in practice, we perform the translation for each pure state t iteratively.

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{0\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E},\mathcal{P},\mathcal{T},\mathcal{A} \qquad (\textsc{Nil})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{P\mid Q\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E},\mathcal{P}\cup\{\!\!\{P,Q\}\!\!\},\mathcal{T},\mathcal{A} \qquad (\textsc{Par})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{!P\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E},\mathcal{P}\cup\{\!\!\{P,!P\}\!\!\},\mathcal{T},\mathcal{A} \qquad (\textsc{Repl})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{new}\ a;P\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E}\cup\{\,a'\,\},\mathcal{P}\cup\{\!\!\{P\{^{a'}/_{a}\}\}\!\!\},\mathcal{T},\mathcal{A} \qquad \text{if } a'\notin\mathcal{E} \qquad (\textsc{Restr})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{out}(N,M);P,\mathsf{in}(N,x);Q\}\!\!\},\mathcal{T},\mathcal{A}\xrightarrow{\mathrm{msg}(N,M)}_o\mathcal{E},\mathcal{P}\cup\{\!\!\{P,Q\{^{M}/_{x}\}\}\!\!\},\mathcal{T},\mathcal{A} \qquad (\mathrm{I/O})$$

$$\mathcal{E},\mathcal{P},\mathcal{T},\mathcal{A}\xrightarrow{\mathrm{msg}(N,M)}_o\mathcal{E},\mathcal{P},\mathcal{T},\mathcal{A} \qquad \text{if } N,M\in\mathcal{A} \qquad (\textsc{Msg})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{let}\ x=D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E},\mathcal{P}\cup\{\!\!\{P\{^{M}/_{x}\}\}\!\!\},\mathcal{T},\mathcal{A} \qquad \text{if } D\Downarrow M \qquad (\textsc{Let1})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{let}\ x=D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E},\mathcal{P}\cup\{\!\!\{Q\}\!\!\},\mathcal{T},\mathcal{A} \qquad \text{if } D\Downarrow \mathsf{fail} \qquad (\textsc{Let2})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{out}(N,M);P\}\!\!\},\mathcal{T},\mathcal{A}\xrightarrow{\mathrm{msg}(N,M)}_o\mathcal{E},\mathcal{P}\cup\{\!\!\{P\}\!\!\},\mathcal{T},\mathcal{A}\cup\{\,M\,\} \qquad \text{if } N\in\mathcal{A} \qquad (\textsc{Out})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{in}(N,x);Q\}\!\!\},\mathcal{T},\mathcal{A}\xrightarrow{\mathrm{msg}(N,M)}_o\mathcal{E},\mathcal{P}\cup\{\!\!\{Q\{^{M}/_{x}\}\}\!\!\},\mathcal{T},\mathcal{A} \qquad \text{if } N,M\in\mathcal{A} \qquad (\textsc{In})$$

$$\mathcal{E},\mathcal{P},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E},\mathcal{P},\mathcal{T},\mathcal{A}\cup\{\,M\,\} \qquad (\textsc{App})$$
$$\text{if } M_1,\ldots,M_n\in\mathcal{A},\ f/n\in\mathcal{F}_c\cup\mathcal{F}_d \text{ and } f(M_1,\ldots,M_n)\Downarrow M$$

$$\mathcal{E},\mathcal{P},\mathcal{T},\mathcal{A}\rightarrow_o\mathcal{E}\cup\{\,a'\,\},\mathcal{P},\mathcal{T},\mathcal{A}\cup\{\,a'\,\} \qquad \text{if } a'\notin\mathcal{E} \qquad (\textsc{New})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{event}(ev);P\}\!\!\},\mathcal{T},\mathcal{A}\xrightarrow{\mathrm{event}(ev)}_o\mathcal{E},\mathcal{P}\cup\{\!\!\{P\}\!\!\},\mathcal{T},\mathcal{A} \qquad (\textsc{Event})$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{insert}\ tbl(M_1,\ldots,M_n);P\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o \qquad (\textsc{Insert})$$
$$\mathcal{E},\mathcal{P}\cup\{\!\!\{P\}\!\!\},\mathcal{T}\cup\{\,tbl(M_1,\ldots,M_n)\,\},\mathcal{A}$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{get}\ tbl(x_1,\ldots,x_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o \qquad (\textsc{Get1})$$
$$\mathcal{E},\mathcal{P}\cup\{\!\!\{P\sigma\}\!\!\},\mathcal{T},\mathcal{A}$$
$$\text{if } tbl(x_1,\ldots,x_n)\sigma\in\mathcal{T} \text{ and } D\sigma\Downarrow true$$

$$\mathcal{E},\mathcal{P}\cup\{\!\!\{\mathsf{get}\ tbl(x_1,\ldots,x_n)\ \mathsf{suchthat}\ D\ \mathsf{in}\ P\ \mathsf{else}\ Q\}\!\!\},\mathcal{T},\mathcal{A}\rightarrow_o \qquad (\textsc{Get2})$$
$$\mathcal{E},\mathcal{P}\cup\{\!\!\{Q\}\!\!\},\mathcal{T},\mathcal{A}$$
$$\text{if for all } \sigma,\ tbl(x_1,\ldots,x_n)\sigma\notin\mathcal{T} \text{ or } \neg(D\sigma\Downarrow true)$$

Figure 8: Transitions between configurations.

F Proverif export

F.1 Without states

We introduce the semantics of ProVerif of in Fig. 8, dropping the part of the configuration dedicated to phases. A configuration $\mathcal{E},\mathcal{P},\mathcal{A}$ is given by a multiset $\mathcal{P}$ of processes, representing the current state of the process, a set of names $\mathcal{E}$ representing the free names of $\mathcal{P}$ and the names created by the adversary, and a set $\mathcal{A}$ of terms known by the adversary. The semantics of processes is defined through a reduction relation $\xrightarrow{l}_o$ where l is either the empty label or a label of the form msg(M,N) or event(ev) with M, M being terms and ev being an event.

We remark that the classical semantics of ProVerif does not have conditionals if $M = N$ then P else Q, because it can be encoded using a let binding with a equal function symbol destructor, let equal$(M,N) =$ `true` in$P$ else Q. We however consider that it is now included in the semantics and the syntax, as it is simpler for our translation. We also add the prefix -P to all rule names, to not coincides with the rules names for our Sapic$^+$ semantics.

We denote by $traces^{proverif}(P)$ the set of traces (observable events) of a process with respect to the ProVerif semantics.

F.2 Translation of non-static states

We rely on the cell semantics from the GSVerif extension [21]. It allows to declare a channel over which all actions can be fired only once. Static states are translated using the encoding presented in Appendix E.2.3.

For non static states, each state identifier is associated with a corresponding secret name, where the association is stored inside a table $\mathsf{state\_tbl}$. The secret name for an identifier is generated during the first access to the state, i.e., when trying to get the secret name inside the table fails and we go inside the $\mathsf{state\_tbl}$. To ensure that two secret names are not created for the same channel, we must make those operations atomic through the use of a dedicated locking channel s.

Locks are modelled in a similar fashion.

Definition 18. *Let P be a process such that on every path on the syntactic tree, each unlock is mapped injectively to a lock, without any other unlock in between nor parallel and replication operators (this corresponds to the general syntactic restriction over processes of [37]). We assume two table identifiers $\mathsf{state\_tbl}$ and $\mathsf{lock\_tbl}$, as well as a secret channel s. We assume that for each state identifier t occurring in P, we have fresh variables $c_t, x_t, y_y \in \mathcal{X}$ and a fresh name $n_t \in \mathcal{N}_{priv}$.*

$$\begin{array}{rcl}
[\![\mathsf{lock}\ t; P]\!]^{\mathsf{P\text{-}st}} & := & \mathsf{in}(s, y_t); \mathsf{get}\ \mathsf{lock\_tbl}(=t, n_t)\ \mathsf{in} \\
 & & \quad \mathsf{out}(s, y_t) | \mathsf{in}(n_t, x_t); [\![P]\!]^{\mathsf{P\text{-}st}} \\
 & & \ \mathsf{else} \\
 & & \quad \mathsf{new}\ n_t; \mathsf{insert}\ \mathsf{lock\_tbl}(t, n_t); \mathsf{out}(s, y_t) | \mathsf{let}\ x_t = 0\ \mathsf{in} [\![P]\!]^{\mathsf{P\text{-}st}} \\
[\![\mathsf{unlock}\ t; P]\!]^{\mathsf{P\text{-}st}} & := & \mathsf{get}\ \mathsf{lock\_tbl}(=t, n_t)\ \mathsf{inout}(n_t, 0) | [\![P]\!]^{\mathsf{P\text{-}st}} \\
[\![\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P\ \mathsf{else}\ Q]\!]^{\mathsf{P\text{-}st}} & := & \mathsf{get}\ \mathsf{state\_tbl}(=t, n_t)\ \mathsf{in} \\
 & & \quad \mathsf{in}(n_t, x); (\mathsf{out}(n_t, x) | [\![P]\!]^{\mathsf{P\text{-}st}}) \\
 & & \ \mathsf{else} \\
 & & \quad [\![Q]\!]^{\mathsf{P\text{-}st}} \\
[\![\mathsf{insert}\ t, M; P]\!]^{\mathsf{P\text{-}st}} & := & \mathsf{in}(s, y_t); \mathsf{get}\ \mathsf{state\_tbl}(=t, n_t)\ \mathsf{in} \\
 & & \quad \mathsf{out}(s, y_t) | \mathsf{in}(n_t, x); (\mathsf{out}(n_t, M) | [\![P]\!]^{\mathsf{P\text{-}st}}) \\
 & & \ \mathsf{else} \\
 & & \quad \mathsf{new}\ n_t; \mathsf{insert}\ \mathsf{state\_tbl}(t, n_t); \mathsf{out}(s, y_t) | (\mathsf{out}(n_t, M) | [\![P]\!]^{\mathsf{P\text{-}st}})
\end{array}$$

and where $[\![P]\!]^{\mathsf{P\text{-}st}}$ is inductively propagated.

To prove the validity of the translation, we require that the general syntactic conditions on $\mathsf{lock}\ t$ and $\mathsf{unlock}\ t$ given in [37] holds (we actually also need it for the export to Tamarin, it is a syntactic condition without which the language does not make sense). It ensures that on every possible path through the process tree, each unlock is injectively mapped to a single lock without replication or parallel between them.

Lemma 6. *Let P be a process with only the equality predicate. We denote by $[\![P]\!]^{\mathrm{PV}}$ the process $\mathsf{out}(s, 0); [\![P]\!]^{\mathsf{P\text{-}st}}$. We have that:*

$$\mathit{trans}(\mathit{traces}^{\mathit{proverif}}([\![P]\!]^{\mathrm{PV}}) = \mathit{traces}^{\mathit{pi}}_{K}(P)$$

Proof. We denote by $\overset{tr}{\Rightarrow}_{\mathrm{ProVerif}}$ the reduction relation corresponding to $\mathit{traces}^{\mathit{proverif}}$. We show that there exists a bijective mapping f between configurations of P and $[\![P]\!]^{\mathsf{P\text{-}st}}$ such that, for all tr, $(\emptyset, \emptyset, \{P\}, \emptyset, \emptyset) \overset{tr}{\Longrightarrow} \mathcal{C} = (\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) \Leftrightarrow (\emptyset, \{[\![P]\!]^{\mathsf{P\text{-}st}}\}, \emptyset, \emptyset\}) \overset{trans(tr)}{\Rightarrow}_{\mathrm{ProVerif}} f(\mathcal{C}) = \mathcal{E}_{\mathrm{ProVerif}}, \mathcal{P}_{\mathrm{ProVerif}}, \mathcal{T}, \mathcal{A}$ such that:

- $t \in \mathcal{L}$ if and only if there exists a unique $n_t \in \mathcal{E}_{\mathrm{ProVerif}}$ such that $\mathsf{lock\_tbl}(t, n_t) \in \mathcal{T}$ and $\{\mathsf{out}(n_t, 0)\} \notin \mathcal{P}_{\mathrm{ProVerif}}$;
- $[t \mapsto M] \in \mathit{StoreA}$ if and only if there exists a unique $n_t \in \mathcal{E}_{\mathrm{ProVerif}}$ such that $\mathsf{state\_tbl}(t, n_t) \in \mathcal{T}$ and $\{\mathsf{out}(n_t, M)\} \in \mathcal{P}_{\mathrm{ProVerif}}$.
- $[\![\mathcal{P}]\!]^{\mathsf{P\text{-}st}} \subset \mathcal{P}_{\mathrm{ProVerif}}$

We reason by induction on the last step of the reduction.
$\Rightarrow$ We branch depending on the last rule, that goes from configuration $\mathcal{C}$ to $\mathcal{D}$, and we assume that $(\emptyset, \emptyset, \{P\}, \emptyset, \emptyset) \overset{tr}{\Longrightarrow} \mathcal{D}$ and $[\![P]\!]_t^{\mathsf{st}} \overset{tr}{\Longrightarrow} f(\mathcal{C})$ with the above hypothesis over $f(\mathcal{C})$. We only consider reduction steps that concern a state operation, as the mapping is trivial for all others.

- $(\mathcal{E}, \mathcal{S}[t \mapsto M'], \mathcal{P} \cup^{\#} \{\mathsf{insert}\ t, M; P\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}[t \mapsto M], \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L})$

  By induction hypothesis, the translation of the insert occurs in the processes of $f(\mathcal{C})$ and as $[t \mapsto M'] \in \mathcal{S}[t \mapsto M']$, there exists a unique $n_t \in \mathcal{E}_{\text{ProVerif}}$ such that $\mathsf{state\_tbl}(t, n_t) \in \mathcal{T}$ and $\{\mathsf{out}(n_t, M')\} \in \mathcal{P}_{\text{ProVerif}}$.

$$\begin{aligned} f(\mathcal{C}) \quad & \rightarrow_o *\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{in}(n_t, x); \mathsf{out}(n_t, M) | [\![P]\!]^{\mathsf{P\text{-}st}}\}\!\} \cup \{\!\{\mathsf{out}(n_t, M)\}\!\}, \mathcal{T}, \mathcal{A} = f(\mathcal{D}) \\ & \rightarrow_o *\mathcal{E}, \mathcal{P} \cup \{\!\{[\![P]\!]^{\mathsf{P\text{-}st}}\}\!\} \cup \{\!\{\mathsf{out}(n_t, M)\}\!\}, \mathcal{T}, \mathcal{A} = f(\mathcal{D}) \end{aligned}$$

  And we do have for $f(\mathcal{D})$ that $\mathsf{out}(n_t, M) \in \mathcal{P}'_{\text{ProVerif}}$, needed as we have in $\mathcal{D}$ the store $\mathcal{S}[t \mapsto M]$. The other hypothesis are inherited form $\mathcal{D} = f(\mathcal{C})$ by the induction hypothesis.

- $(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{insert}\ t, M; P\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}[t \mapsto M], \mathcal{P} \cup^{\#} \{P\}, \sigma, \mathcal{L})$
  By induction hypothesis, the translation of the insert occurs in the processes of $f(\mathcal{C})$ and as t does not appear in $\mathcal{S}$, there does not exist $\mathsf{state\_tbl}(t, n_t) \in \mathcal{T}$ and $\{\mathsf{out}(n_t, M')\} \notin \mathcal{P}_{\text{ProVerif}}$. We can thus take the else branch, and reduce to:

$$f(\mathcal{C}) \quad \rightarrow_o *\mathcal{E}, \mathcal{P} \cup \{\!\{[\![P]\!]^{\mathsf{P\text{-}st}}\}\!\} \cup \{\!\{\mathsf{out}(n_t, M)\}\!\}, \mathcal{T} \cup \{\mathsf{state\_tbl}(t, n_t)\}, \mathcal{A} = f(\mathcal{D})$$

  And as in the previous case, this new configuration satisfies the induction hypothesis.

- $(\mathcal{E}, \mathcal{S}[t \mapsto M], \mathcal{P} \cup^{\#} \{\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P\ \ \mathsf{else}\ \ Q\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{P\{M/x\}\}, \sigma, \mathcal{L})$
  By induction hypothesis, the translation of the lookup occurs in the processes of $f(\mathcal{C})$ and as $[t \mapsto M] \in \mathcal{S}[t \mapsto M]$, there exists a unique $n_t \in \mathcal{E}_{\text{ProVerif}}$ such that $\mathsf{state\_tbl}(t, n_t) \in \mathcal{T}$ and $\{\mathsf{out}(n_t, M)\} \in \mathcal{P}_{\text{ProVerif}}$. We can then perform the reduction:

$$\begin{aligned} f(\mathcal{C}) \quad & \rightarrow_o *\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{in}(n_t, x); \mathsf{out}(n_t, x) | [\![P]\!]^{\mathsf{P\text{-}st}}\}\!\} \cup \{\!\{\mathsf{out}(n_t, M)\}\!\}, \mathcal{T}, \mathcal{A} \\ & \rightarrow_o *\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{out}(n_t, M) | [\![P\{M/x\}]\!]^{\mathsf{P\text{-}st}}\}\!\} \cup \{\!\{\mathsf{out}(n_t, M)\}\!\}, \mathcal{T}, \mathcal{A} \\ & \rightarrow_o *\mathcal{E}, \mathcal{P} \cup \{\!\{\mathsf{out}(n_t, M)\}\!\} \cup \{\!\{[\![P\{M/x\}]\!]^{\mathsf{P\text{-}st}}\}\!\}, \mathcal{T}, \mathcal{A} = f(\mathcal{D}) \end{aligned}$$

  This new configuration then meets the hypothesis, as we once again have P on one side and $[\![P\{M/x\}]\!]^{\mathsf{P\text{-}st}}$ on the other.

- $(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{\mathsf{lookup}\ t\ \mathsf{as}\ x\ \mathsf{in}\ P\ \ \mathsf{else}\ \ Q\}, \sigma, \mathcal{L}) \quad \longrightarrow \quad (\mathcal{E}, \mathcal{S}, \mathcal{P} \cup^{\#} \{Q\}, \sigma, \mathcal{L})$
  The translation of the lookup occurs in the processes of $f(\mathcal{C})$. Further, by induction hypothesis, as t does not occur in $\mathcal{S}$ we have that there does not exist any $\mathsf{state\_tbl}(t, n_t) \in \mathcal{T}$. We can then perform the reduction:

$$f(\mathcal{C}) \quad \rightarrow_o *\mathcal{E}, \mathcal{P} \cup \{\!\{[\![Q]\!]^{\mathsf{P\text{-}st}}\}\!\}, \mathcal{T} \cup, \mathcal{A}$$

  And this new reduction does satisfy the induction hypothesis.

- We do not detail the lock and unlock cases, that are simpler occurrences of the previous cases, and also similar to the proof of Lemma 5.

$\Leftarrow$ We branch depending on the last rule, that goes from configuration $\mathcal{C}'$ to $\mathcal{D}'$, and we assume that $[\![P]\!]_t^{\mathsf{st}} \overset{tr}{\Longrightarrow} \mathcal{D}'$ and that $P \overset{tr}{\Longrightarrow} f^{-1}(\mathcal{C}')$ and with the above hypothesis over $\mathcal{C}'$ and $f^{-1}(\mathcal{C}')$. We defined previously the bijection for all reduction steps that do not concern a translation of a process, or concerns the last step of the reduction of a translated process.

To complete the bijection, we thus only have to define the antecedent for such steps. For reductions that correspond to a partial execution of a translation, we simply map them to the same configuration as the previous one ($f^{-1}(\mathcal{D}') = f^{-1}(\mathcal{C}')$), and it trivially preserves the induction hypothesis, as such steps never introduce a new $\mathsf{out}(n_t, M)$ in the multiset.

$\square$

G Full semantics

Standard operations:

$$
\begin{array}{rcl}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{0\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P|Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P,Q\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{!P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{!P,P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\nu n;P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E}\cup\{n'\},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{n\mapsto n'\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } n'\in\mathcal{N}_{\mathtt{priv}} \text{ is fresh} \\
(\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) & \xrightarrow{K(t)} & (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) \\
 & & \quad \text{if } t=_E R\sigma \text{ for some } R\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{out}(t_1,t_2);P\},\sigma,\mathcal{L}) & \xrightarrow{\mathsf{Out}(R),K(t_1)} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma\cup\{\mathtt{att}_n\mapsto t_2\},\mathcal{L}) \\
 & & \quad \text{if } t_1=_E R\sigma \text{ for some } R\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX}) \\
 & & \quad \mathsf{Msg}(t_2) \text{ and } n=|\sigma|+1 \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{in}(t,x);P\},\sigma,\mathcal{L}) & \xrightarrow{\mathsf{In}(R,R'),K(\langle t,R'\sigma\rangle)} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{x\mapsto R'\sigma\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } t=_E R\sigma, \mathsf{Msg}(R'\sigma) \text{ for some } R,R'\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{out}(t_1,t_2);P,\mathsf{in}(t,x);Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P,Q\{x\mapsto t_2\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } t_1=_E t \text{ and } \mathsf{Msg}(t_2) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{if } \phi \text{ then } P \text{ else } Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \phi \text{ holds} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{if } \phi \text{ then } P \text{ else } Q\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{Q\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \phi \text{ does not hold} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{event}(F);\ P\},\sigma,\mathcal{L}) & \xrightarrow{F} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{let}\ p=t\ \mathsf{in}\ P\ \mathsf{else}\ Q\},\sigma,\mathcal{L}) & \rightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\tau\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } p\tau=_E t \text{ and } \tau \text{ is grounding for } p \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{let}\ p=t\ \mathsf{in}\ P\ \mathsf{else}\ Q\},\sigma,\mathcal{L}) & \rightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) \\
 & & \quad \text{if for all } \tau, p\tau\neq_E t
\end{array}
$$

(Ms

Operations on global state:

$$
\begin{array}{rcl}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{insert } t_1,t_2;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S}[t_1\mapsto t_2],\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{delete } t;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S}[t\mapsto\bot],\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lookup } t \text{ as } x \text{ in } P \text{ else } Q\ \},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{u/x\}\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \mathcal{S}(t')=_E u \text{ is defined and } t=_E t' \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lookup } t \text{ as } x \text{ in } P \text{ else } Q\ \},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) \\
 & & \quad \text{if } \mathcal{S}(t') \text{ is undefined for all } t'=_E t \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lock } t;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\cup\{\,t\,\}) \quad \text{if } t\not\in_E\mathcal{L} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{unlock } t;\ P\},\sigma,\mathcal{L}) & \longrightarrow & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\setminus\{\,t'\mid t'=_E t\,\})
\end{array}
$$

Figure 9: Operational semantics