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

Vincent Cheval
Inria Paris

Charlie Jacomme
CISPA Helmholtz Center for Information Security

Steve Kremer
Université de Lorraine LORIA & Inria Nancy

Robert Künnemann
CISPA Helmholtz Center for Information Security

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-EDHOC` [49] and the `Privacy-Pass` [23] protocols, both under standardization, the `SSH` [50] protocol with the agent-forwarding feature, and the recent `KEMTLS` [48] protocol, a post-quantum version of the main TLS key exchange.

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 [18, 29], 5G authentication protocols [14], WPA 2 [30], Noise [34, 39] and Signal [27, 38], 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 [40] to TAMARIN [47] and PROVERIF's applied π calculus [19] 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 [25], and DEEPSEC [26]. 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 [14]: 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 [44] or human errors [17]) 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., [37, 38, 41]) 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 [40]) 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 [40]. 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-EDHOC` 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 [14] 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 [8] and, its successor, AVANTSSAR projects [7] 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 backends 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., [6, 16, 46]). Similarly, many other independent tools enable automatic protocol verification (see [12] 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 $\mathcal{N}_{\mathtt{pub}}$ and secret (protocol) names $\mathcal{N}_{\mathtt{priv}}$. 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 $enc \in \mathcal{F}_c$, $k \in \mathcal{N}_{\mathtt{priv}}$ 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 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

$$[lhs] \; -[\, actions \,]\!\!\rightarrow \; [rhs].$$

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.

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

In rule $\mathtt{R_0}$ we require a fresh long-term key `lk`, shared between P and Q, and move these two parties into the states $\mathtt{SP(lk)}$ and $\mathtt{SQ(lk)}$, 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 `k` is tagged as an honest key. Q on the other hand, when in state $\mathtt{SQ(lk)}$ and with an available input matching $\mathtt{enc}(\langle \mathtt{k}, \mathtt{'hs'}\rangle, \mathtt{lk})$, can accept key k. The facts $\mathtt{Fr}(\cdot), \mathtt{In}(\cdot)$ and $\mathtt{Out}(\cdot)$ 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 $\mathtt{Out(t)}$ adds `t` to the attacker knowledge; conversely, $\mathtt{In(t)}$ requires the attacker to construct a term that matches `t`. When we generated the facts $\mathtt{SP(lk)}$ and $\mathtt{SQ(lk)}$ 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 `lt`, generating a fresh session key `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 [32, 47]. 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 `k`, `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
```

```
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

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.

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 equa-

tional 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 [33].

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

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 [9]). Direct encodings in SAPIC (e.g., SGX reports [37] or accountability [41]) translate *for free*, and become available in PROVERIF for the first time.

3.2 Protocols

3.2.1 Syntax

We present the syntax of SAPIC$^+$ in Fig. 2 where t, st, t_i range over arbitrary terms, 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 and perform input and output over a channel t_1, based on some control flow.

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

(elementary processes)	*(extended processes)*
0	| event F; P
| new n; P	| $!P$
| $P \mid Q$	*(stateful processes)*
| out(t_1, t_2); P	| insert st, t; P
| in(t_1, x); P	| delete st; P
| if ϕ then P else Q	| lookup st as x in P else Q
| let $p = t$ in P else Q	| lock st; P
	| unlock st; P

Figure 2: Syntax of our process calculus

Extended processes model unbounded protocols which may also raise events. Events provide a convenient way to model reachability properties. *Stateful processes* can manipulate globally shared states st. Conditionals are described by first-order formulae ϕ over equalities on terms, possibly containing variable quantifiers, as in [40].

3.2.2 Semantics

We equip SAPIC$^+$ with an operational semantics suited for expressing both equivalence and reachability properties, in contrast to SAPIC, which only supported reachability properties. A *configuration* defines the current execution of the process and contains, among others, a set of executable processes $\mathcal{P}$ and the attacker knowledge inside a substitution σ. The domain of σ will contain variables from $\mathcal{AX} = \{\mathtt{att}_i\}_{i \in \mathbb{N}}$.

The semantics is a labeled transition system between configurations, defined by a relation $\xrightarrow{\ell}$ where ℓ may be empty, or a set of *facts*. We use facts to annotate processes and log events. For instance, ℓ may contain $K(t)$ to log that the attacker knows t; $\mathsf{In}(R, R')$ to log the *recipes* used by the attacker to compute the protocol inputs; $\mathsf{Out}(R)$ to log the recipe used by the attacker to compute the channel of an output. Formally, a recipe is a term containing only variables from $\mathcal{AX}$ and public names. We give an excerpt of the semantics in Fig. 3 (see Fig. 9 in Appendix A for the full version).

We define $P \xrightarrow{l_1 \cdots l_n} \mathcal{C}$ when $\{P\}, \emptyset \rightarrow^* \xrightarrow{l_1} \rightarrow^* \ldots \rightarrow^* \xrightarrow{l_n} \rightarrow^* \mathcal{C}$. Such a sequence of labels, together with the attacker knowledge σ of the resulting configuration $\mathcal{C}$, defines a *trace*. We denote by $traces^{pi}(P)$ the set of traces of a process P, i.e., $\{(tr, \sigma) \mid \exists \mathcal{P}.\ P \xRightarrow{tr} \mathcal{P}, \sigma\}$. For equivalence properties, we only consider the traces (tr, σ) of $traces^{pi}(P)$ where we removed from tr all facts other than In and Out, denoted $traces^{pi}_{\approx}(P)$.

$$\mathcal{P} \cup^{\#} \{\mathsf{out}(t_1,t_2);P\}, \sigma \xrightarrow{\mathsf{Out}(R),K(t_1)} \mathcal{P} \cup^{\#} \{P\}, \sigma \cup \{\mathtt{att}_n \mapsto t_2\}$$
if $t_1 =_E R\sigma, R$ a recipe, $t_2 \not\to_E$ fail and $n = |\sigma| + 1$

$$\mathcal{P} \cup^{\#} \{\mathsf{in}(t,x);P\}, \sigma \xrightarrow{\mathsf{In}(R,R'),K(\langle t,R'\sigma\rangle)} \mathcal{P} \cup^{\#} \{P\{x \mapsto R'\sigma\}\}, \sigma$$
if $t =_E R\sigma, R'\sigma \not\to_E$ fail and R, R' recipes

$$\mathcal{P} \cup^{\#} \{\mathsf{let}\ p = t\ \mathsf{in}\ P\ \mathsf{else}\ Q\}, \sigma \to \mathcal{P} \cup^{\#} \{P\tau\}, \sigma$$
if $p\tau =_E t$ and τ is grounding for p

Figure 3: Operational semantics (excerpt)

3.3 Security properties

3.3.1 Reachability properties

To express reachability properties, we use the same temporal logic as the one of TAMARIN [47] and introduced in Section 2.2. 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. Intuitively, a time point corresponds to an index of a trace sequence $tr = \ell_1 . . . . \ell_n$. We denote by $\mathcal{X}_{temp}$ the set of time point variables.

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 and a time point i. A trace formula is a first-order formula over atomic trace formulas.

The satisfaction on closed atomic trace formulae on a trace sequence $tr = \ell_1 . . . . \ell_n$ is naturally defined as

- $t_1 \approx t_2$ iff $t_1 =_E t_2$;
- $i \lessdot j$ iff $1 \leq i < j \leq n$;
- and $F@i$ iff $1 \leq i \leq n$ and $F \in \ell_i$.

The satisfaction of a trace formula φ on a trace sequence tr, denoted $tr \vDash \varphi$, is extended as expected. Note that quantified variables must be instantiated with terms of the same sort. For example, the formula $\exists i \in \mathcal{X}_{temp}, x \in \mathcal{X}.\varphi$ holds in tr if and only if there exists an integer n and a message t such that $\varphi\{i \mapsto n; x \mapsto t\}$ holds in tr. Given a process P, we write $P \vDash \varphi$ when for all traces (tr, σ) in $trace^{pi}(P)$, $tr \vDash \varphi$.

3.3.2 Equivalence properties

Unlike SAPIC, SAPIC$^+$ also supports indistinguishability properties, as introduced in Section 2.3. Such properties can be modeled by trace equivalence. We first define static equivalence, modelling the indistinguishability of two sequences of terms represented as substitutions.

Definition 1. *Two closed substitutions σ_1, σ_2 on messages are statically equivalent, written $\sigma_1 \sim_E \sigma_2$, iff*

$$\forall M, N \text{ recipes},\ M\sigma_1 =_E N\sigma_1 \Leftrightarrow M\sigma_2 =_E N\sigma_2$$

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

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

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

Other equivalence properties Besides trace equivalence, other process equivalences are sometimes considered. In particular, *observational equivalence* (which can be characterized as a labelled bisimulation [4]) is a stronger equivalence that implies trace equivalence, which itself implies testing equivalence [24]. On a subclass of processes, called *determinate*, all these notions coincide [24].

In general, properties can be expressed with any of these equivalences, but with subtle differences in the adversary's distinguishing power. For instance, (the notion of strong) unlinkability was initially defined using observational equivalence [5], and later using trace equivalence [11, 35].

Unfortunately, tools such as ProVerif and Tamarin cannot directly show these equivalences. They support *diff-equivalence*, a stronger equivalence that requires the two processes to have the same structure and to only differ in the terms [20]. While diff-equivalence is easier to prove and provides a convenient proof technique (as it implies observational and trace equivalence) it may sometimes be too strong and lead to false attacks.

As we will see below, for SAPIC$^+$ we rely on PROVERIF's ability to prove diff-equivalence, which then implies trace equivalence. Obviously, we suffer from the same limitation as PROVERIF when false attacks arise. In that case we can use DEEPSEC which proves trace equivalence precisely, but at the cost of bounding the number of sessions.

3.4 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 3.3.1. 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 $\vDash^{\mathrm{PV}}$.

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 describe in more detail these translations which we denote by $[\![\cdot]\!]^{\text{TAM}}$ for the translation to TAMARIN and $[\![\cdot]\!]^{\text{PV}}$ 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.*

$$P \vDash \varphi \quad \Leftrightarrow \quad [\![P]\!]^{\text{TAM}} \vDash [\![\varphi]\!]^{\text{TAM}}$$

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*

- $\text{Pat} = \text{Pat}_{ex}$ *and P does not contain conditional branchings with variable quantifiers;*
- φ *is a formula with a single quantifier alternation.*

Then $P \vDash \varphi \Leftrightarrow [\![P]\!]^{\text{PV}} \vDash^{\text{PV}} [\![\varphi]\!]^{\text{PV}}$.

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 [20] 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 Theorem 2, we have that

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

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 [15]. We thus do not translate equivalence properties to Tamarin.

4 From SAPIC$^+$ to TAMARIN

We extend the 2014 translation from SAPIC to TAMARIN [40] with new syntactic features that improve usability (cf. Section 4.1.1) 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.

4.1 SAPIC$^+$ extensions and updates

4.1.1 New syntactic features

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 ...)`

4.1.2 New semantical features

Moreover, we introduce two new semantical features in let-bindings. SAPIC only supports let bindings for a single variable of the form let $x = t$ in P. We generalize to pattern matchings with a failure branch of the form let $s = t$ in P else Q where $s,t \in \mathcal{T}(\mathcal{F}_c, \mathcal{E}, \mathcal{X})$. In a second step, we allow the right-hand side t to contain destructor functions, i.e., $t \in \mathcal{T}(\mathcal{F}_c, \mathcal{N}, \mathcal{X})$.

The pattern matching permits to concisely extract relevant segments from incoming messages, e.g.,

```
let <key,'hs'>=dec(cipher,sk) in [..]
```

in the running example. The failure branch (`else out(c, 'abort')`) allows to react if the incoming message does not decrypt, a feature that, e.g., AEAD schemes or CCA2-secure public key encryption schemes posses. As both TAMARIN and SAPIC do not support destructor functions, we provided translation for let bindings with destructors where a restriction encodes the failure conditions.

In [1], we show the correctness of this translation. Notably, we extend the 40-page proof from [40] in a black box fashion with the following techniques. For the first step, we exploit that the SAPIC semantics includes a feature for embedding MSRs within the process, which we omitted here for the ease of presentation. We expressed the first step 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. The second step is expressed as a source-to-source transformation that is valid due to the correctness of the first step. We thus obtain a fully modular proof of correctness.

4.1.3 Updates to avoid pitfalls

We made several changes to the translation of SAPIC when moving to SAPIC$^+$ to avoid some pitfalls either when translating to PROVERIF or when expressing equivalence-based properties .

	# rules		running time (s)		
Case Study (Section 6.2)	NC	C	NC	C	R
KEMTLS [48]	98	42	1.4k	47	28
KEMTLS-CA [48]	124	57	20.7k	1.1k	18
KEMTLS-NO-AEAD [48]	94	41	11.7k	222	52
LAKE-EDHOC [49]	121	56	1.5k	265	5
SSH [50]	67	37	53	4	11
SSH-NEST [50]	106	58	3.1k	16	10
Privacy-Pass [23]	44	20	13	10	1.3
AC [13,37]	48	12	2	1	2
AKE [13,37]	35	15	1.3	0.8	1.5
OTP [36,37]	94	40	17	6	3
NSL [40,43]	59	25	80	18	4.4

NC: without compression C: with compression R: ratio

Figure 4: Benchmarks for path compression

Notably, the pattern matching inside an input behaves differently in PROVERIF and SAPIC. For PROVERIF, $\mathsf{in}(t,p)$; P with pattern p is merely syntactic sugar for $\mathsf{in}(t,x)$; 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 changed SAPIC$^+$ to follow PROVERIF's semantics, as it was important to obtain the same behaviours for the equivalence semantics of PROVERIF and DEEPSEC.

Another pitfall is that while TAMARIN supports general pattern matchings, where the pattern $\langle x,x,y\rangle$ will for instance expect twice the same value for x and either bind y if it is a fresh variable or expect it to match the existing value of y, 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 behaviors may be confusing, and therefore SAPIC$^+$ forbids 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.

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.

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 new a; new b; $\mathsf{out}(c, \langle a, b \rangle)$. We obtain three msrs from the translation, but as only the last step is observable, these three rules can be compressed into one.

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., new a; $\mathsf{out}(c,a)$; ... cannot be compressed with its continuation ...; $\mathsf{in}(c,x)$; ....

We define this compression as an optimization of the MSRs which is, in principle, applicable to handwritten TAMARIN models. We provide in [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 Figure 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 communication, 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 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. 5, Appendix A). `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 SAPIC's repository. Note that on the latter, TAMARIN times out without this optimization.

	case study	w/o compr.	w/ compr.	ratio
[36, 37]	`OTP`	25	8	3
[50]	`SSH-NEST`	594	316	2
	`U2F-TOY`	∞	30	∞

Figure 5: Benchmarks for the secret channel optimization compression: running time (in seconds)

	case study	state restr.	state facts	ratio
[13, 37]	`AC`	127	1	127
[13, 37]	`AC-F-sid`	64	250	$\frac{1}{4}$
[13, 37]	`AC-F-counter`	266	4	66
[6, 40]	`SD`	47	∞	$\frac{1}{\infty}$

Figure 6: Benchmarks for the state facts option: running time (in seconds)

5 From SAPIC$^+$ to PROVERIF

The syntax and semantics of PROVERIF are very similar to SAPIC$^+$ as both use dialects of the applied π calculus for their input languages. We describe the main differences and how they are handled bellow. We do not detail the translation for DEEPSEC, as it is similar to the one of PROVERIF.

5.1 Translating conditionals

Extended processes only differ in the semantics of conditional branching containing destructors. In SAPIC$^+$, if $u = v$ then P 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 are 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 if $u = v$ then P else Q conditional branching is translated in PROVERIF using let bindings.

While this may seem anecdotal, 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, which illustrates the risks of working with different tools. There are many other pitfalls in manually translating models between PROVERIF, TAMARIN or DEEPSEC, concerning, among others, bindings inside pattern matchings, destructor inside conditionals and how pairs are encoded. We detail those differences in [1].

5.2 Translating states

ProVerif has no direct syntax for stateful processes and requires an encoding that relies on internal communication on private channels. Although semantically correct, ProVerif struggles to prove security properties relying on this encoding. In practice, it leads to false attacks. The recent GSVerif front end [25] addresses many false attacks related to stateful protocols. However, GSVerif's transformations easily lead to nontermination issues and are sometimes too restrictive to handle general stateful processes. We therefore updated the GSVerif tool (available in [2, 3]) 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.

Although the applied π calculus lacks 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

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

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 [1].

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 [13, 37], the verification time shrinks from 127 seconds to 1 but on the `SD` protocol [6, 40], the encoding seems to lead to nontermination (see Figure 6, Appendix A). Using Sapic$^+$, we can easily switch between both encodings.

This translation of states is efficient and sufficient for most of our cases studies. However, it does not cover all the Sapic$^+$ processes. For the remaining cases that do not match our syntactic conditions (i.e., not all states are *pure*), we exploit ProVerif's `table` construct. Tables define a form of global memory that processes can insert element to and look up elements from. However, unlike Sapic$^+$'s store and locking table, ProVerif's tables are append-only: once inserted, an element cannot be removed nor replaced. We use these tables to associate states with a secret channel name as in the pure case, but now we have to take care that the secret channel is created on first access only. This is difficult: as the store is global, the first access is not bound to a particular position in the process. Thus we must make sure those operations are atomic through a dedicated locking channel. The translations for lock/unlock and insert/lookup follow this pattern, but differ in when the placement of the private channel communication, as the former lock construct is blocking, while insert is not. See [1] for the translation and its proof of correctness.

5.3 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 queries 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 some semantic pitfalls we will explain as we go along. 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 Section 3.3.1, similarly to Sapic$^+$ and Tamarin. There are, however, some differences in the facts and the fragment of the logic that prevent the 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 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`, and we therefore only allow universally quantified K facts in queries. For example, the following lemma will correctly be translated in the given query.

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

ProVerif also allows for *injective events* that 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, but Sapic$^+$ 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 [40], 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` [2] that allows to run all the tools easily and reproduce our case studies (detailed instructions in [1]).

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;
- 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. 7 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` [48], 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.[1] `KEMTLS` derives a secret shared key with

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

Case Study	ProVerif	Tamarin	Property
5G AKA [14]	78	95	Auth
KEMTLS [48]	1	24	Auth., PFS
KEMTLS-NO-AEAD	1	108	Auth., PFS
KEMTLS-CA	1	247	MA, Unlink.
LAKE-EDHOC [49]	3	96	MA, PFS, KCI
SSH [50]	1	3	MA, Sec.
SSH-NEST	2	147	MA, Sec.
SSH-NEST(X)	$7 \times 3^{X-1}$	∞	MA, Sec.
Privacy-Pass [23]	1	2	Unforg., Unlink.
NSL [40,43]	1	3	MA, Sec.
	DeepSec		
KEMTLS-CA [48]	955		Unlink. (3 sess.)
NSL [40,43]	362		Strong Sec. (3 sess.)

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

Figure 7: Running time (seconds) of the case studies

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 [48] to match the existing option of TLS, but was provided 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 [11,35], for three sessions using DeepSec, and an unbounded number of sessions with ProVerif.

LAKE-EDHOC Lightweight Authenticated Key Exchange (LAKE-EDHOC) 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-EDHOC model, we proved mutual authentication, KCI and PFS of the exchanged key.

In concurrent work [45], a different version of the LAKE-EDHOC 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 [42] and computational [22]. 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 [10], but it relies on an external composition result.

The SSH model verifies authentication and secrecy. Interestingly, a previous Tamarin model [42] required 26 seconds while our SAPIC$^+$ model is verified in three seconds.[2] 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.[3]

Privacy-Pass The Privacy-Pass protocol, proposed in [31] and under standardization [23], is a token-based authorization of clients to servers that aims to preserve client anonymity. A server issues tokens upon client requests and

[2] Timing obtained by running the SSH model of [42] on the same laptop with the same version of Tamarin.

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

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 [11, 35]) 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 also ported an existing TAMARIN model.

5G AKA A previous case study of the 5G AKA authentication standard was performed recently in TAMARIN [14], 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 [37] 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[4], we could ensure termination and attack reconstruction.

Others Finally, we imported the `Scytl` e-voting protocol [28] from the GSVERIF benchmark [25], and the secure device (`SD`) [6, 40] and `NSL` models [40] 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 [21] 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 [21], 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 [25] 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.

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 [2].

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.

[4] Additonal lemmas and *nounif* instructions.

Acknowledgments. We thank Ioana Boureanu and the anonymous reviewers for their helpful comments and suggestions. This work has been partly supported by the ANR Research and teaching chair in AI ASAP, ANR TECAP (decision number ANR-17-CE39-0004-03), and the the ERC Synergy Grant "imPACT" (No. 610150).

References

[1] Sapic+: protocol verifiers of the world, unite! Technical report.

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

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

[4] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[5] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 107–121. IEEE Computer Society, 2010.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] David A. Basin, Sasa Radomirovic, and Lara Schmid. Modeling human errors in security protocols. In *IEEE 29th Computer Security Foundations Symposium (CSF 2016)*, pages 325–340. IEEE Computer Society, 2016.

[18] 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.

[19] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 82–96. IEEE Computer Society, 2001.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Vincent Cheval, Véronique Cortier, and Stéphanie Delaune. Deciding equivalence-based properties using constraint solving. *Theor. Comput. Sci.*, 492:1–39, 2013.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 1773–1788. ACM, 2017.

[30] 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 *29th USENIX Security Symposium (USENIX Security 2020)*, pages 1–17. USENIX Association, 2020.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

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

[44] 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, 2018.

[45] 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.

[46] 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.

[47] 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.

	PROVERIF	TAMARIN
Protocol language	applied π	MSR
Automation ranking	strong	medium
Interactive proofs	✗	✓
Complex State Machine	∼	✓
Equational theory support:		
finite variant	✓	✓
linear equations	✓	✗
AC operator,XOR	✗	✓
Integer support	✓	≈
DH theory (without addition in exponents)	≈	✓

✓: natively supported ✗: not supported
∼: requires some expertise and/or may greatly reduce automation
≈: supported through approximations

Figure 8: Some high-level differences between PROVERIF and TAMARIN

[48] 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.

[49] 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.

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

A Appendix

In Figure 8, we provide an overview over the respective strengths and weaknesses of PROVERIF and TAMARIN and in Figure 9, the operational semantics of SAPIC$^+$.

Standard operations:

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

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

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

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\nu n;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E}\cup\{n'\},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{n\mapsto n'\}\},\sigma,\mathcal{L})$$
if $n'\in\mathcal{N}_{\mathtt{priv}}$ is fresh

$$(\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) \xrightarrow{K(t)} (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L})$$
if $t=_E R\sigma$ for some $R\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX})$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{out}(t_1,t_2);P\},\sigma,\mathcal{L}) \xrightarrow{\mathsf{Out}(R),K(t_1)} (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma\cup\{\mathtt{att}_n\mapsto t_2\},\mathcal{L})$$
if $t_1=_E R\sigma$ for some $R\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX})$
$\mathsf{Msg}(t_2)$ and $n=|\sigma|+1$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{in}(t,x);P\},\sigma,\mathcal{L}) \xrightarrow{\mathsf{In}(R,R'),K(\langle t,R'\sigma\rangle)} (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{x\mapsto R'\sigma\}\},\sigma,\mathcal{L})$$
if $t=_E R\sigma,\mathsf{Msg}(R'\sigma)$ for some $R,R'\in\mathcal{T}(\mathcal{F},\mathcal{N}_{\mathtt{pub}},\mathcal{AX})$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{out}(t_1,t_2);P,\mathsf{in}(t,x);Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P,Q\{x\mapsto t_2\}\},\sigma,\mathcal{L})$$
if $t_1=_E t$ and $\mathsf{Msg}(t_2)$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{if }\phi\text{ then }P\text{ else }Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{P\},\sigma,\mathcal{L})$$
if ϕ holds

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup\{\text{if }\phi\text{ then }P\text{ else }Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup\{Q\},\sigma,\mathcal{L})$$
if ϕ does not hold

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

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{let}\ p=t\ \mathsf{in}\ P\ \mathsf{else}\ Q\},\sigma,\mathcal{L}) \to (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\tau\},\sigma,\mathcal{L})$$
if $p\tau=_E t$ and τ is grounding for p

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathsf{let}\ p=t\ \mathsf{in}\ P\ \mathsf{else}\ Q\},\sigma,\mathcal{L}) \to (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L})$$
if for all $\tau, p\tau\neq_E t$

Operations on global state:

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{insert }t_1,t_2;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S}[t_1\mapsto t_2],\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L})$$

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

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lookup }t\text{ as }x\text{ in }P\text{ else }Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{u/x\}\},\sigma,\mathcal{L})$$
if $\mathcal{S}(t')=_E u$ is defined and $t=_E t'$

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{lookup }t\text{ as }x\text{ in }P\text{ else }Q\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L})$$
if $\mathcal{S}(t')$ is undefined for all $t'=_E t$

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

$$(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\text{unlock }t;P\},\sigma,\mathcal{L}) \longrightarrow (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\setminus\{t'\mid t'=_E t\})$$

Figure 9: Operational semantics