Subterm-based proof techniques for improving the automation and scope of security protocol analysis

Cas Cremers
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
cremers@cispa.de

Charlie Jacomme
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
charlie.jacomme@cispa.de

Philip Lukert
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
philip.lukert@cispa.de

2023 IEEE 36th Computer Security Foundations Symposium (CSF) | 979-8-3503-2192-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSF57540.2023.00001

***Abstract*—During the last decades, many advances in the field of automated security protocol analysis have seen the field mature and grow from being applicable to toy examples, to modeling intricate protocol standards and finding real-world vulnerabilities that extensive manual analysis had missed.**

However, modern security protocols often contain elements for which such tools were not originally designed, such as protocols that construct, by design, terms of unbounded size, such as counters, trees, and blockchains. Protocol analysis tools such as Tamarin and ProVerif have some very restricted support, but typically lack the ability to effectively reason about dynamically growing unbounded-depth terms.

In this work, we introduce subterm-based proof techniques that are tailored for automated protocol analysis in the Tamarin prover. In several case studies, we show that these techniques improve automation (allow for analyzing more protocols, or remove the need for manually specified invariants), efficiency (reduce proof size for existing analyses), and expressive power (enable new kinds of properties). In particular, we provide the first automated proofs for TreeKEM, S/Key, and Tesla Scheme 2; and we show substantial benefits, most notably in WPA2 and 5G-AKA, two of the largest automated protocol proofs.

1. Introduction

The Tamarin prover is a state-of-the-art security protocol analysis tool that has been used for the analysis of highly detailed models of a wide range of security protocols. Notable examples include TLS 1.3, 5G-AKA, Wifi's WPA2, and EMV (Chip-and-Pin) [1], [2], [3], [4], in each case finding attacks or proving new properties. Tamarin was first released in 2012 [5] and has seen substantial development over the last decade. This includes extending its range of equational theories (e.g., best-in-class Diffie-Hellman modeling [6], exclusive or [7], multisets and bilinear pairing [8], and a generalization of subterm-convergent user-specific theories [9]), induction, improved proof-finding heuristics, improved reasoning methods [10], a wider range of modeling options [11] and support for observational equivalence [12].

Despite this active development and significant progress, there are still several types of protocol analysis problems that pose a challenge for Tamarin. In the default setting, Tamarin's backwards search works by refining so-called dependency graphs until either a solution is found (typically a counterexample to, or attack on, the intended security property) or it can be shown that no solutions exist (corresponding to a proof that the property holds). It can additionally use a form of induction over trace events. However, Tamarin 1.6 (the latest version) cannot reason about arbitrary-depth *subterms*. Notably, while arbitrary-depth subterms did not occur in classical simple protocol models (e.g., [13]), they do occur naturally in detailed case studies of modern protocols. Examples of such protocols include hash-chain based protocols (blockchains, Tesla, S/Key), protocols based on tree structures (TreeKEM), and protocols using natural numbers (YubiKey, PKCS#11, WPA2, 5G-AKA). In each of the mentioned protocols, there is typically a relation between temporal ordering and the construction of a dynamic term. For example, counters may increase monotonically, trees might be extended while keeping existing subtrees intact, and blockchains are strictly increasing terms by design. Such protocols pose a new kind of challenge to the automated provers, as they introduce a new form of unboundedness: not only do we want to consider an unbounded number of sessions, but each session of the protocol itself may, by design, construct terms of unbounded size.

In this work, we set out to extend the Tamarin prover version 1.6 with a subterm relation and more generic subterm-based proof techniques. Our goal is two-fold. First, we extend the language of security properties for more expressive power. Second, we improve the automation of Tamarin for more complex case studies, for example by improving the analysis times, or by enabling automated analysis of protocols that could previously only been analyzed with manual guidance (e.g., by specifying reusable lemmas or invariants). This

DOI 10.1109/CSF57540.2023.00001

goal of improving automation is one of the main aspects of our design choices.

Contributions. Our main contribution is the introduction of subterms and subterm-based proof techniques suitable for automated analysis of security protocols with the Tamarin Prover. This enables, for example, automated analysis of hash-chain based protocols and reasoning about natural numbers, in the presence of equational theories.

In particular, our new proof techniques enable the first automated analysis of the TreeKEM, S/Key, and Tesla Scheme 2 protocols, where a subterm relation both helps simplify the proof and expressing the desired security properties, and additionally significantly improve the automation level in case studies on YubiKey, PKCS#11, CH'07, and two of the largest Tamarin case studies to date: WPA2 and 5G-AKA; in particular they remove the need for certain manually specified invariants (reusable lemmas and oracles) and reduce proof size and proving time (up to 30x).

Related Work. ProVerif [14] is the main other widely used automated protocol verifier in the unbounded setting. It was extended with support for natural numbers with GSVerif [15] and as a builtin later on [16], which enabled the analysis of versions of the Yubikey and PKCS#11 protocols. They handle natural numbers similar to our approach, where they have a dedicated type, and a special proof technique that helps reasoning about monotonous counters. However, our approach is more general as we build over a subterm ordering that has a broader scope of applications. Furthermore, due to the ProVerif-internal limitation of not having support for associative-commutative operators, they restrict themselves to natural numbers with an increment operator instead of the more powerful addition. In practice, this design choice means that in ProVerif one can only model adding a constant to a variable. In contrast, our approach also allows specifying the addition of two variables. We note that the manual of ProVerif version 2.04 [17] refers to a subterm predicate, but this is not a proof technique: the subterm predicate can only be used in the premise of restrictions, and thus only to restrict the possible behaviors of a protocol. We do not know of any paper or case study in ProVerif that refers to this predicate.

Both Scyther [18] and CPSA [19], [20] use a proof technique that links fresh values occurring inside a term to where this value originates from. Such a technique implicitly relies on some specific case of a subterm ordering notion, but in a coarse and very restricted way compared to our approach. Notably, there is no explicit subterm predicate that can be used to specify invariants over the protocol, like monotonicity. Furthermore, Scyther does not support equational theories. We are not aware of any fresh-value subterm technique used inside the Maude-NPA [21] tool.

A first prototype of natural numbers for Tamarin was proposed in the Master Thesis of [22]. It contained a dedicated type system and its proof techniques were quite restricted. It did not contain any proofs of correctness, and was never integrated into Tamarin.

Some of our case studies use and extend prior Tamarin models by several authors. The two main speed-ups that we obtain are over the 5G-AKA model [2] and the WPA2 model [3]. In terms of scale, these are also the largest formal models available for these protocols. We also speed-up YubiKey and PKCS#11, which were studied using the previously mentioned GSVerif [15] as well as Sapic [23], a Tamarin front-end that uses an applied pi-calculus as its input language. We note that our extensions naturally carry over to Sapic.

Our novel case studies, TreeKEM [24], S/Key [25], and Tesla Scheme 2 [26], do not have any automated proofs that we know of. Tesla Scheme 1 was proven secure in [27], in which Scheme 2 was mentioned as an example that showcases the limitations of Tamarin.

Reproducibility. We provide the source code of our extended Tamarin version as well as our case studies at [28]. Alternatively, we provide a docker that contains pre-built binaries of the multiple Tamarin versions as well as our example, allowing to reproduce the results from Tables 1 and 2.

After installing Docker[1], one simply has to pull the image and enter it to reproduce our results:

```
docker pull securityprotocolsresearch/tamarin:st
docker run -it securityprotocolsresearch/tamarin:st bash
```

Overview. We first provide the required background on Tamarin in Section 2 and then formally describe in Section 3 our extensions over subterms and natural numbers. We report on our case studies in Section 4. We give additional details about Tamarin and the full proofs of the soundness and completeness of our extensions in the extended version [29].

2. Background: The Tamarin Prover

We now introduce the theoretical background needed for our extensions.

Messages. Messages sent over the network are abstracted by so-called **terms**, which are built over a set of atoms by function application from a set of function symbols Σ. The atoms are drawn either from a set of constant values N or a set of variables V, and we denote by $T_\Sigma(V, N)$ the corresponding set of terms. Each atomic value can be of the generic message sort *msg*, or of the sub-sort *fresh* to model the sampling of a random value inside a protocol, or of the sub-sort *pub* to model a public and attacker-known constant. *fresh* values are prefixed by $\sim$ and *pub* values by \$. For instance, with $\Sigma = \{enc, dec\}$, the term $enc(m, \sim sk)$ models the encryption of some variable message m with a secret key sk. For a term $f(t_1, \ldots, t_n)$, we say that f occurs at the top of the terms, and that each t_i occurs below f.

1. https://docs.docker.com/get-docker/

To capture the cryptographic properties of the primitives modeled by the function symbols, we define as an **equational theory** the equality relations that hold over terms. For a symmetric encryption, we would declare the following equation:

$$dec(enc(m,k),k) = m$$

In Tamarin, there is a builtin model for pairs, denoted by $\langle x, y\rangle$ and with the associated projections *fst*, *snd* where we have the following equation for the first projection

$$fst(\langle x, y\rangle) = x$$

Formally, an equation is an unoriented pair $t = t'$ of terms in $T_\Sigma(V)$, and an equational theory is a set E of equations, which together introduce a congruence relation $=_E$ over terms. In Tamarin, we require from E that each term can be rewritten to a so-called *normal form* modulo E. In most cases, E will be clearly fixed by the context in which case we simply write $=$ for $=_E$.

Looking forward, equations will also be used to capture constraints over terms in the solving procedure. We will then need to consider the possible set of solutions of an equation. An **E-unifier** for an equation $t = t'$ is a substitution σ that is a mapping from variables to terms with $t\sigma =_E t'\sigma$. The set of unifiers for a given equation is usually infinite, e.g., $x = x$ is true for any substitution of the variable x. Thus, we only consider a so-called *complete* set of unifiers $CSU_E(t = t')$, which is a subset of all unifiers such that they can be instantiated to cover all unifiers. Intuitively, if $CSU_E(t = t')$ is empty for a given equation, it means that it can never be satisfied.

Only the equalities defined by the equational theory hold. A unary function h for which we define no equation then models an idealized hash function. Tamarin has builtin definitions for many primitives such as symmetric and asymmetric encryption, signatures, exclusive-or, Diffie-Hellman, etc. Notably, Tamarin has a builtin for multisets that have been used to model counters and which are built with a union function symbol $+$ defined as an associative and commutative (AC) operator:

$$x + y = y + x \qquad x + (y + z) = (x + y) + z$$

We say that a symbol function f is **cancellative** if there exists an equation such that f occurs at the top on one side and some variable only occurs on the same side, i.e., the variable may be "cancelled". $\oplus$ is notably cancellative due to the equation $x \oplus x = 0$. We say a function symbol f is **reducible** if there exists an equation such that f occurs at the top on the left side of a rewriting rule. For example, both *fst* and *dec* are reducible. Note that we do not consider the builtin multiset symbol $+$ to be reducible, because the AC-operator $+$ is handled separately in Tamarin.

Protocols. The states of threads of agents that perform a protocol are modeled with facts of the form $F(t_1, \ldots, t_n) \in \mathcal{F}$, with F taken from a set of fact names and $t_1, \ldots, t_n \in T_\Sigma(N, V)$. The global state of all agents is captured as a multiset S of such facts. A protocol is then modelled as a set of multi-set rewriting rules, where each rule specifies how the multiset, and thus the protocol state, can evolve through time.

Formally, a rule is a tuple $ri = (id, l, a, r)$ written $id : l -[a]\!\!\rightarrow r$ where id is a unique name and l, a, r are multisets of facts. To extract properties from this tuple, we define $name(ri) = id$ for the name, $prems(ri) = l$ for the premises that are consumed by the rule, $acts(ri) = a$ for the actions used to annotate the execution trace, and $concs(ri) = r$ for the conclusions that are produced by the rule. The actions a are later used for specifying security properties. Tamarin comes with several builtin facts to model protocols: $\texttt{Fr}(\sim n)$ to sample fresh random values such as nonces and keys, $\texttt{Out}(t)$ and $\texttt{In}(t)$ are used for outputs to and inputs from the attacker-controlled network, and $\texttt{K}(t)$ enables reasoning about the attacker's knowledge. Those also come with a set of built-in rules to model the attacker deduction over the K fact that includes the closure of the knowledge by function application modulo E.

Example 1. *The two following rules model the beginning of a hash chain protocol, where $R1$ models that each agent samples a fresh identity* id *and a fresh seed* k*, and $R2$ allows to build some arbitrary long hash chain using a loop over the agent state.*

$$R1 : \texttt{Fr}(\sim\texttt{k}), \texttt{Fr}(\sim\texttt{id}) -[\texttt{Init}(\sim\texttt{id}, \sim\texttt{k})]\!\!\rightarrow \texttt{State}(\sim\texttt{id}, \sim\texttt{k})$$
$$R2 : \texttt{State}(\sim\texttt{id}, x) -[\texttt{Chain}(\sim\texttt{id}, x)]\!\!\rightarrow \texttt{State}(\sim\texttt{id}, h(x))$$

The computation could be concluded and the hash chain sent over the network by adding the rule

$$R3 : \texttt{State}(\sim\texttt{id}, x) -[]\!\!\rightarrow \texttt{Out}(h(x))$$

Facts can either be linear or persistent. While a linear fact will be consumed by a rule, a persistent fact will always stay inside the protocol state once produced. By convention, a persistent fact name is prefixed by the symbol !. In the previous example State is linear. Turning it into a persistent fact !State would mean that each step of $R2$ would still store inside the memory the intermediate values of the hash chain — values that could then be reused inside some other rules.

Formally, a set of rules RU defines a labeled transition relation over protocol states S, where $S\ (l -[a]\!\!\rightarrow r)\ S'$ is possible when $l -[a]\!\!\rightarrow r$ is a valid instantiation of a rule $l_x -[a_x]\!\!\rightarrow r_x$ from RU, i.e., there exists a substitution σ from variables to ground terms such that $l -[a]\!\!\rightarrow r = l_x\sigma -[a_x\sigma]\!\!\rightarrow r_x\sigma$, $l \subset S$ and $S' = S \setminus^\sharp l \cup^\sharp r$.

An execution in this transition system is a sequence of states S_i ($i \geq 0$, $S_i = \emptyset$) which are connected by rules:

$$S_0\ (l_1 -[a_1]\!\!\rightarrow r_1)\ S_1\ \ldots\ S_{n-1}\ (l_n -[a_n]\!\!\rightarrow r_n)\ S_n$$

The trace of this execution is $a_1, \ldots, a_n$. We denote the set of all traces of a protocol P as $traces(P)$. Implicitly, the user-specified rules of P are extended with Tamarin's built-in rules to generate the transition system.

Security properties. TAMARIN allows for specification of properties in a temporal first-order logic that may hold over the (infinite) set of traces of a protocol. Given a trace $a_1, \ldots, a_n$, each a_i corresponds to the multiset of action facts that occur at timepoint i. The atoms of the logic are then defined over message and timepoint variables as:

- $F(t_1, \ldots, t_k)@i$, where $F(t_1, \ldots, t_k) \in \mathcal{F}$ and i is a timepoint, which is true if there exists such an occurrence of F (modulo substitution of variables) in a_i;
- $t = t'$, which holds if the equality over the terms hold;
- $i < j$, which holds if the timepoint ordering holds.

TAMARIN's logic is then built over those atoms with conjunction, disjunction, implication, and universal/existential quantification over message or timepoint variables. A formula ϕ holds for a protocol P if it holds over all traces of P. In TAMARIN's framework, all formulas (including main theorems) are specified as *lemmas*.

Example 2. *We can express over the protocol from Example 1 that the Chain action for a given id is always raised at most once for a given value x of the hash chain with the following* no_replay *lemma:*

$$\forall\ id, x, i, j.\ \texttt{Chain}(id, x)@i\ \&\ \texttt{Chain}(id, x)@j \Rightarrow i = j$$

Looking ahead, this property holds trivially by the fact that the hash chain for a specific id is in some sense strictly growing. There is, however, no way to express such a property given the existing predicates of TAMARIN*'s logic. Using the dedicated* K *fact that models the attacker knowledge, we could also express that the attacker can only know the last element of the chain, and not any hash chain value computed before that:*

$$\forall\ id, x, i.\ \texttt{Chain}(id, x)@i \Rightarrow \neg(\exists\ j.\ \texttt{K}(x)@j)$$

Constraint solving. To prove or disprove a formula for a protocol, TAMARIN essentially solves a constraint solving problem: the rules generate constraints on the possible executions, and the formula is negated and converted into a set of logical constraints. TAMARIN's algorithm applies sound and complete constraint solving rules to refine, simplify, or case-split such constraint systems. If TAMARIN can find a solution for the constraint system, this constitutes a counterexample to the formula; if it can establish that no solution exists, then this constitutes a proof that the property holds.

Example 3. *Consider the formula*

$$\forall\ id, x, y, i, j.\ \ \texttt{Chain}(id, x)@i\ \&\ \texttt{Chain}(id, y)@j \\ \&\ y = h(x)\ \Rightarrow \neg(i = j)$$

Any potential counterexample to this formula would be captured with the constraint system $\Gamma = [\texttt{Chain}(id, x)@i \wedge \texttt{Chain}(id, y)@j \wedge y = h(x) \wedge i = j]$. *Using its set of constraint solving rules,* TAMARIN *would then prove the formula by deriving a contradiction from this constraint system, or by finding a counterexample.*

$$\frac{\Gamma \quad s = t}{\Gamma\sigma_1 \mid \ \ldots\ \mid \Gamma\sigma_n}\ \text{IF}\ CSU(s = t) = \{\sigma_1, \ldots, \sigma_n\}$$

Figure 1: The $S_\approx$ rule

We provide in Fig. 1 the rule $S_\approx$[5] as an example of a constraint solving rule. It specifies that given Γ and an equation, we can try to derive a contradiction by exploring all the new constraints obtained when considering the possible ways to solve this equation. This corresponds to applying each substitution from the *CSU* to Γ and then consider the disjunction of the new constraints.

Example 4. *We have the most general set of unifiers* $CSU(h(x) = y) = \{\sigma : y \mapsto h(x)\}$, *where in this particular case there is only a single possibility. From* Γ *of Example 3, a* $S_\approx$ *application over* $y = h(x)$ *then yields* $\Gamma_1 = [\texttt{Chain}(id, x)@i \wedge \texttt{Chain}(id, h(x))@j \wedge i = j]$. *Using other rules that we do not detail here, we could then deduce, from the fact that* $i = j$ *and that the given system never allows raising* Chain *twice at the same timepoint, that we need to have* $x = h(x)$, *which instantly leads to a contradiction, as* $CSU(x = h(x)) = \emptyset$.

This constraint solving problem is in general undecidable. In practice, TAMARIN relies on a set of heuristics to decide which of the applicable constraint solving rules should be applied to a given constraint system. If TAMARIN terminates, it explored all the possibilities and either yields an attack or a proof. When the analysis fails to terminate using the default heuristics, users can either use TAMARIN's interactive mode to try to perform the proof themselves, declare intermediate lemmas that can be reused for later proofs, or define so-called oracles that can programmatically override the built-in heuristics where needed.

3. Subterm-based proof techniques

In this section, we formally describe our two main extensions to TAMARIN:

- the addition of a subterm ordering and related proof techniques in Section 3.1, and
- a precise model for natural numbers, for which we reuse and build upon the subterm ordering and provide specialized proof strategies in Section 3.2.

This allows us to provide two new proof techniques:

- the Fresh Ordering rule in Section 3.3, based on the assumption that a random value cannot be used inside a term before it was created;
- the Monotonicity rule in Section 3.4, which relies on detecting that some facts are manipulating terms that are always increasing w.r.t. to the subterm

ordering, which allows introducing a correlation between the timepoint ordering and subterm ordering exists.

3.1. Subterms

Our goal is to introduce a subterm predicate that captures a dependency relation on terms. Intuitively, if x is a subterm of t, then x is needed to compute t. To be amenable to automated reasoning, such a relation must be a strict partial order and notably satisfy transitivity. A first intuitive definition for this subterm relation is the syntactic subterm relation:

Definition 1 (Syntactic subterm)**.** *$\sqsubset_{synt}$ is the minimal transitive closure of $\{t_i \sqsubset_{synt} f(\dots, t_i, \dots) \mid f \in functions, t_i \in terms\}$*

It is, however, more difficult to define a meaningful subterm relation when dealing with an *equational theory* E. Morally, it makes sense that if two terms are equivalent modulo E, they can be exchanged in a subterm predicate. This intuition is formally captured by the consistency notion.

Definition 2 (Consistent relation)**.** *We say that $\sqsubset_x$ is consistent modulo E if for all terms s, s', t, t', we have:*

$$(s =_E s' \wedge t =_E t') \Rightarrow ((s \sqsubset_x t) \Leftrightarrow (s' \sqsubset_x t'))$$

This property is not satisfied by $\sqsubset_{synt}$ as we have $x + y \sqsubset_{synt} x + y + z$ but not $x + y \sqsubset_{synt} y + z + x$. Cancellative function symbols also add a layer of complexity. Exclusive-or is a good example of a cancellative function: $x \sqsubset_{synt} x \oplus x$ holds while $x \sqsubset_{synt} 0$ does not hold, even though we have that $x \oplus x =_E 0$.

Once a consistent relation has been found, there is the need for a constraint solving strategy over the corresponding predicate. Tamarin often uses variables as placeholders for arbitrary terms in order to reason symbolically. To deal with variables, we would ideally want to use a similar strategy as for equations. There, recall that we find the most general set of unifiers and substitute all variables with them. However, the set of most general unifiers for subterms $s \sqsubset t$ can be infinite, $h(x) \sqsubset y$ has for example the unifiers $y \mapsto g(h(x))$, $y \mapsto g(g(h(x)))$, $\dots$, $y \mapsto g^n(h(x))$, and we will have to come up with a dedicated proof technique.

We now provide in the following the definition of a consistent subterm relation for the equational theories supported by Tamarin, after which we detail a *constraint solving* algorithm for subterms.

Equational theory. In Tamarin, E is internally divided into two parts: AC, the Associative and Commutative part and R, a user-defined convergent rewriting system. For AC, we define $\sqsubset_{AC}$ as follows:

Definition 3 (AC-subterm)**.**
$s \sqsubset_{AC} t \coloneqq \exists\, s', t'.\ (s' =_{AC} s) \wedge (t' =_{AC} t) \wedge (s' \sqsubset_{synt} t')$

This works well for AC as it is not cancellative. However, if we define $\sqsubset_{R,AC}$ similarly, we get $x \sqsubset_{R,AC} 0$ because 0 can be expanded to the equivalent term $x \oplus x$. Luckily, the convergence of the rewriting system R provides a unique (up to AC) normal form for each term, e.g., $x \oplus x \downarrow_{R,AC} = 0$. With this normal form, we can define $\sqsubset_{R,AC}$ in a one-way fashion:

Definition 4 (R,AC-subterm)**.**
$s \sqsubset_{R,AC} t \ \coloneqq\ (s \downarrow_{R,AC}) \sqsubset_{AC} (t \downarrow_{R,AC})$

With this definition, we trivially obtain that $\sqsubset_{R,AC}$ is a consistent relation modulo the equational theory R, AC:

Lemma 1. *$\sqsubset_{R,AC}$ is R, AC consistent.*

As the equational theories supported by Tamarin are of the form R, AC, the $\sqsubset_{R,AC}$ definition is thus a suitable subterm relation for our purpose in Tamarin, and we choose it as our interpretation of $\sqsubset$. In particular, in the remainder of this paper, we will often write $\sqsubset$ as a shorthand for the chosen the subterm relation $\sqsubset_{R,AC}$.

Constraint Solving and $\sqsubset_{R,AC}$. In the proof search, Tamarin will now produce constraints of the form $t \sqsubset t'$, for two terms t, t' that may contain variables. To ensure the validity of a subterm predicate in Tamarin's constraint system, we follow a proof strategy with three main points (simplified):

1) Deconstruct the right side of the subterms until we only have variables. I.e., at the end, all subterms are of the form $s \sqsubset x$ where x is a variable. For example, the solving algorithm replaces $x \sqsubset h(y)$ with $x \sqsubset y \vee x = y$.
2) Check that we do not have loops of the form $x \sqsubset y \wedge y \sqsubset x$ or $h(x) \sqsubset x$, i.e., the transitive closure of $\sqsubset$ forms a directed acyclic graph.
3) At the end, for each subterm $s \sqsubset x$, we apply the substitution $x \mapsto fun(s)$ where *fun* is a fresh function symbol. This is done implicitly.

This algorithm will either derive a contradiction or provide a valid way to instantiate the constraint. The first solving step is formally captured in the rule RECURSE of Fig. 2, where we specify that given a constraint containing $t \sqsubset f(t_1, \dots, t_n)$, we may introduce a disjunction of constraints that either say that $t = f(t_1, \dots, t_n)$ or that t is a subterm of one of the t_i. This rule does not hold if f is an Associative Commutative function symbol (as $x \sqsubset a + (b + c)$ can for instance be satisfied by $x \sqsubset a + c$), nor when f is reducible. The second solving step where we check for loops is formally described in the rule CHAIN of Fig. 2, where we write $a \bmod n$ to denote the modulo operation for relating x_n and t_0.

If there is no equational theory, these steps ensure that all subterm constraints are met, as $s \sqsubset fun(s)$ holds trivially under all substitutions. For AC, we can adapt the RECURSE rule as seen in Fig. 3. However, for arbitrary rewriting rules, we cannot do this kind of recursion. For example, $x \sqsubset x \oplus y$ is not equivalent to $(x = x) \vee (x \sqsubset x) \vee (x = y) \vee (x \sqsubset y)$ which would be

RECURSE:

$$\frac{t \sqsubset f(t_1, \ldots, t_n)}{t = t_1 \mid t \sqsubset t_1 \mid \cdots \mid t = t_n \mid t \sqsubset t_n}\ \mathcal{I}$$

if f is not AC and not a reducible operator

CHAIN:

$$\frac{t_0 \sqsubset x_0 \qquad \cdots \qquad t_n \sqsubset x_n}{\bot}\ \mathcal{I}$$

- *if x_i are variables of sort msg, and*
- *x_i is syntactically in $t_{(i+1) \bmod (n+1)}$ and not below a reducible operator*

Figure 2: The recurse rule deconstructs the right side of a subterm predicate into a disjunction of equalities and smaller subterms. The chain rule detects loops in the subterm relation and enables deriving a contradiction. The $\mathcal{I}$ denotes insertion into the constraint system and is added here for consistency with the extended version [29].

AC-RECURSE:

$$\frac{t \sqsubset t_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} t_n}{\exists x.\ t \mathbin{+\!\!+} x = t_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} t_n \mid t \sqsubset t_1 \mid \cdots \mid t \sqsubset t_n}\ \mathcal{I}$$

- *where x is a new variable,*
- *$+\!\!+$ is an ac-operator and neither reducible nor the addition from natural numbers, and*
- *there is no t_i with $+\!\!+$as top operator (flatness)*

NEG:

$$\frac{\neg s \sqsubset r \qquad t \sqsubset r}{\neg s \sqsubset t \qquad s \neq t}\ \mathcal{I}$$

Figure 3: AC-RECURSE and NEG. AC-RECURSE works similarly to RECURSE while the existential quantification ensures that cases like $t = t_4 + t_1$ are captured. Additionally note, that we require flatness (t_i don't have $+$ as uppermost operator) as a performance optimization to avoid adding more existential quantifications than necessary. The NEG rule deals with negative subterms.

trivially true, independent of y. To avoid this problem, we explicitly exclude reducible operators from the RECURSE rule, which are the function symbols that are at the top of left sides of rewriting rules. E.g., for the rule $fst(\langle x, y\rangle) \to_{R,AC} x$ we have that *fst* is reducible, but the pair function $\langle, \rangle$ is not.

In conclusion, the strategy is: recursing on irreducible operators and hoping that we do not end up with a reducible operator at the end. If that happens, the result is that we can neither prove nor disprove this claim, but we observe that reducible operators are quite rare in protocols where subterms make sense, e.g., we could not find a sensible meaning of subterms for XOR. The most frequent usage is for hashes h, key derivation functions *kdf*, pairs $\langle, \rangle$, and multisets $+\!\!+$, which are all irreducible.

Finally, subterms can also occur in negated form in a logical constraint. We recurse similarly on them, such that we end up with a variable on the right-hand side. To automatically derive a contradiction from those negated subterms, we add the rule NEG in Fig. 3. It inserts two new constraints that rule out the contradictory case of $\neg(s \sqsubset r) \wedge (s \sqsubset r)$. If we now (implicitly) apply the substitution $x \mapsto fun(s)$ for each subterm $s \sqsubset x$ at the end, we know that the negative subterms constraints are not violated. The soundness and completeness of the rules are proved in the extended version [29].

3.2. Refinements for Natural Numbers

We now turn to our extension for natural numbers. Two kinds of numbers are used in protocols: some are used as nonces or encryption keys, while the others are smaller values typically used for counters. From a security analysis point of view, they have two very different sets of properties: nonces and keys cannot be guessed by the attacker, and we often do not need to consider the underlying algebraic properties of those integers. In contrast, for counters, every number can be guessed by the attacker with non-negligible probability and we do need to consider the algebraic properties of the addition.

The first kind is traditionally modeled as fresh random values as we have illustrated before. In the following, we provide an efficient model for small integer values. We prove the correctness of our encoding in the extended version [29].

Modeling decisions. There are two main styles to define numbers: as in Peano arithmetic with a 1 and a successor function, or as in Presburger arithmetic with a 1 and an addition of two numbers. In contrast to ProVerif where numbers are implemented in Peano style [15], we use Tamarin's ability to cope with associative-commutative operators to implement the + of Presburger arithmetic. This has the substantial advantage that we can sum arbitrary numbers $n + m$ without having to resort to implementing this with loops that might cause non-termination, e.g., by applying the successor function m times in a loop.

Recall that the multiset operator $+\!\!+$ is commutative $n+\!\!+m = m+\!\!+n$ and associative $(n+\!\!+m)+\!\!+o = n+\!\!+(m+\!\!+o)$; this is why many existing Tamarin models use the multiset operator $+\!\!+$ in combination with a public symbol for '1' to model counters in Presburger arithmetic [3], [2], [30], [31]. For example, a 3 would be represented as '1'$+\!\!+$'1'$+\!\!+$'1', i.e., the number of '1's in the multiset indicates the number represented. A zero cannot be represented as we otherwise would need to switch to the theory ACU, where the U stands for unit ($n +\!\!+ 0 = n$), which is not supported by Tamarin. In practice, there is usually no need for an explicit zero as counters can also

start at one without impacting the security analysis.

The problem of the existing modelings is that the multiset operator can be used with terms of any sort, i.e., there can be other elements than '1' inside the multiset, potentially including secret values. This substantially complicates the proof search which typically significantly slows down the verification of these models. One of the sources of complexity is that the attacker is required to construct each number individually and that it also tries to extract information from multisets that represent numbers. In practice, these two behaviors make no sense as the attacker can directly guess these small numbers. A solution to this is to explicitly define numbers as public values. To this end, we introduce a new sort nat that precisely captures the natural numbers. The two only ways to construct a term of sort nat are $1 : \mathsf{nat}$ and the custom AC-operator $+ : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$. That implies that the attacker can never extract useful information from a nat and never needs to prove that they can construct a nat. We will see later that this leads to substantial speedups and aids termination of a protocol's analysis.

However, these speedups come at a cost as with a strengthened type system, we may hide some type flaw attacks. For example, consider a protocol with an oracle that is supposed to sign small counters but accidentally also signs nonces as they are both implemented as 64bit integers. If we model this oracle with nat in Tamarin, we do not capture the bug. If the second part of the protocol is a challenge which asks the attacker to sign nonces, we have an attack in the real world but not in the symbolic model. To capture this attack, the model would need to go back to using the construction with the multiset operator while avoiding the sort nat. This boils down to the general trade-off between the level of automation and the accuracy of the model. In general, we are guaranteed to model type flaw attacks if we do not assume messages received from the network to have any specific type. Apart from that, we can use the sort nat arbitrarily (e.g., in local state or when sending to the network) which still yields automation improvements.

Less-than relation. When using the subterm relation as a less-than ordering over natural numbers, we observe that the following holds if m and n are nat:

1) It is a total ordering: $(n \sqsubset m) \vee (n = m) \vee (m \sqsubset n)$. This is used for negating a less-than equation.
2) $n \sqsubset m$ can be rewritten to the equation $\exists x.\ n + x = m$. This is used at the end of the constraint solving algorithm for subterms instead of using the fresh function symbol *fun* for the substitution $x \mapsto \mathit{fun}(s)$ (the symbol *fun* of the generic constraint solving algorithm cannot be used for numbers as it would violate the type constraints).
3) It is discrete, which means that we can sometimes extract equations, e.g., from $(m \sqsubset n) \wedge (n \sqsubset m{+}1{+}1)$ follows that $n = m{+}1$. We determine these equations with an efficient UTVPI-algorithm (short for *Unit Two Variable Per Inequality*). This algorithm builds a graph out of the constraints where variables are nodes in the graph and (directed) edges are the orderings between them. After construction, the graph is checked for zero-weight cycles with an adaption of the Bellman-Ford algorithm. For more details, see [32].

We stress that the two variable restriction for the UTVPI algorithm does not imply a restriction of our constraint solving algorithm: the UTVPI is only used as an optimization, and if it cannot be applied, we fall back to using the more general approach of using the formulation $\exists x.\ n + x = m$.

FRESH-ORDER:

$$\frac{i : f \qquad j : g \qquad i \neq j}{i < j}\ \mathcal{I}$$

where $j : g$ denotes that rule g occurs at timepoint j, and

- *f has a premise $\mathtt{Fr}(s)$,*
- *g has a premise fact with the term t,*
- *s is syntactically in t but not below a reducible operator*

Figure 4: The basic fresh order rule

3.3. Fresh Ordering

The idea of the Fresh ordering rule is to derive a temporal ordering constraint between creation and usage of fresh variables: it is intuitively clear that a random variable cannot be used before it is created. This proof strategy was already used in Scyther [18] but was not adapted in Tamarin because of its support for equational theories: with equations, it is not easy to determine whether a fresh value is inside a term, e.g., $x \sqsubset\ x \oplus y$ does not hold if $y \mapsto\ x$. However, if there are no reducible operators between the two terms, we can show that we can safely add the rule in Fig. 4. The rule formally specifies that if we have a rule g occurring at timepoint j, denoted $j : g$, and if there is a fresh value s that appears syntactically in g (and not below a reducible operator), then we know that j must be after the timepoint i of the rule f that produced s in the premise $\mathtt{Fr}(s)$. We describe below informally two variants of this first rule, and provide the full rule as well as soundness and completeness proofs in the extended version [29].

"Subterm" improvement. This rule takes its full meaning when combined with the subterm predicate, which precisely captures the fact that a value is needed to build some term. Previously, we required that s is syntactically in the term t and not below a reducible operator. However, the situation can be that we have the predicate $s \sqsubset t$ while s is not (yet) syntactically in t. Then, we can conclude from the subterm predicate that s will be eventually in t (after some constraint solving steps) and already assume that t "uses" s. Because of transitivity of $\sqsubset$, we can also use chains $s \sqsubset t_1 \ldots t_n \sqsubset t$

to ensure that t "uses" s and insert the corresponding timepoint ordering.

"Secret path" improvement. We can also refine the previous rule in the cases where a fresh value s is secretly given to a second rule at time i after the Fresh rule. In this case, we know that no other rule before i can use s because it is not known to them. I.e., all other occurrences of s have to be after i. This can be extended from a single further rule to a path of rules where s is passed secretly.

3.4. Monotonicity

We now provide ways to automatically detect the monotonous behaviors appearing inside a protocol and how to use those behaviors in the constraint solving. To detect facts that may for instance model a counter, we rely on the existing notion of injective facts in Tamarin, which are instances of facts that are guaranteed to not co-exist in a trace. Knowing about this kind of injectivity currently enables timepoint ordering simplifications.

Example 5. *In Example 1 the* `State` *fact is injective, and we know that two instances of the* `State` *fact with the same* $\sim id$ *cannot both exist at the same time.* Tamarin *uses this information to derive contradictions, as it can notably conclude that if a* $\texttt{State}(\sim id, x)$ *fact is produced at timepoint* i *and consumed at timepoint* j*, there cannot exist another* $\texttt{State}(\sim id, y)$ *in between* i *and* j*.*

We can improve the reasoning over those facts by detecting if they imply monotonous behaviors. To do so, we must inspect the contents of these injective facts that may be seen in this case as storage cell used to store a set of values. For instance, we can syntactically see that in Example 1 the variable x models a storage cell containing a strictly increasing sequence of keys. This means that we can correlate bigger keys with later time points of fact instances representing the same storage cell. We thus extend the injective fact reasoning with techniques associated to monotonicity. All in all, we determine five special cases for contents of storage cells: Strictly Increasing, Strictly Decreasing, Increasing, Decreasing and Constant.

In the following, we first show how to better detect injective facts, which makes our technique more broadly applicable, and how the monotonous behaviors can be inferred over such facts. We then show how this extra information can be used inside the constraint solving. The soundness and completeness proofs of our approach can be found in the extended version [29].

Injective monotonous facts. Injective facts were previously detected by ensuring that there exists a fresh value id that is only used as the first argument of a fact S, there is a single rule that produce the S fact from a $\texttt{Fr}(\sim id)$ fact, and otherwise there are only rules that consume a single S fact and can then produce it. We provide a more general detection, that notably allows producing multiple injective facts at the same time. We improve the detection of injective facts in general and now detect this behavior with the following rule-set:

Definition 5. *A fact* `Fact` *is detected as injective if*

1) *it is linear and not persistent*
2) *for each conclusion of* $\texttt{Fact}(id, \dots)$ *of each rule, there is no other conclusion* $\texttt{Fact}(id, \dots)$ *with the same first term and*
   a) *either there is a premise* $\texttt{Fr}(id)$
   b) *or there is exactly one premise* $\texttt{Fact}(id, \dots)$

The set of injective facts gives us a set of potential storage cells over which we can detect monotonicity properties. In general, we note that an injective fact can be used to store multiple values, and be, e.g., of the form $\texttt{Store}(id, v_1, v_2, v_3)$, where we can see each value v_i as an independent storage cell. We may also encounter cases where the previous is written using a tuple as follows $\texttt{Store}(id, v_1, \langle v_2, v_3 \rangle)$. We also detect such usages, and do see $v2$ and $v3$ as independent storage cells. For each such storage cell, we detect by a syntactic analysis over the rules if the cell is:

- *constant*, when every rule produces the same value it consumed for this cell;
- *strictly increasing*, when for any rule the value in the premise of the cell is a syntactic subterm (and not below a reducible operator) of the one in the conclusion;
- *decreasing* and (*non-strict*) *increasing*/*decreasing* cells by combinations or inversions of the above.

Monotonicity properties. We now consider the case where a monotonous storage cell (corresponding to an injective fact) is used to store the term s at timepoint i and the term t at timepoint j. Then, the following simplifications can be performed for constant and increasing storage cells:

- If the cell is constant:
  (1) insert $s = t$
- If the cell is strictly increasing:
  (2) if $s = t$, then insert $i = j$
  (3) if $s \sqsubset t$, then insert $i < j$
  (4) if $i < j$ or $j < i$, then insert $s \neq t$
  (5) if $\neg s \sqsubset t$ and $s \neq t$, then insert $j < i$

For (3) and (5) we do not require $s \sqsubset t$ to be an explicit predicate in the constraint system but also apply the rule if $s \sqsubset t$ is trivially true, e.g., for syntactic inclusion. Note that (5) holds as $t \sqsubset s$ holds because of totality within the increasing injective fact. Interestingly, this totality $(s \sqsubset t) \vee (s = t) \vee (t \sqsubset s)$ does not hold in general, but holds here because s and t are used in the same monotonic storage cell which yield a total ordering on the possible contents of the storage cell. If the cell is non-strictly increasing, we can only use rules (3) and (5).

Protocol	Properties	LoC	H. Lemmas		Runtime (s)		Oracle	
New models								
TreeKEM[24]	Forward Secrecy	389	4		8		yes	
S/Key [25]	Authentication	101	1		1		no	
Tesla Scheme 2[26]	Authentication, Secrecy	286	5		8		no	
Previous models								
			Old	New	Old	New	Old	New
WPA2 [3]	Secrecy, Authentication	2446	74	73	5189	559	yes	yes
5G-AKA[2]	Secrecy, Authentication	978	7	6	467	131	yes	yes
YubiKey [30]	Authentication, Replay-Resistance	134	4	3	19	1	no	no
PKCS#11 [31]	Key Generation Properties	301	4	0	74	10	yes	no
CH'07 RFID [7]	Unlinkability	92	0	0	3197	97	no	no

LoC: lines of code (approximate complexity measure)
H. lemmas: helper lemmas automatically proved by Tamarin, but manually added to help prove the target property
Oracle: whether an oracle was needed to help guide the proof search, "no" means more automation

TABLE 1: Benchmark overview: new models and improvements for previous models

Protocol	Runtime in seconds					
	Original Models		*Models using dedicated Subterms and Natural Numbers models*			
	Original	Fresh Order	Basic	Fresh Order	Monotonicity	Monotonicity + Fresh Order
TreeKEM[24]	-	-	∞	**8**	∞	8
S/Key [25]	-	-	**1**	1	1	1
Tesla Scheme 2[26]	-	-	**8**	11	6	8
WPA2 [3]	5189	5750	5142	5142	**386**	559
5G-AKA[2]	480	467	**90**	124	108	131
YubiKey [30]	19	21	**1**	1	1	1
PKCS#11 [31]	74	79	**36**	24	**5**	10
CH'07 RFID [7]	3197	**96**	3197	**96**	2721	97

We compare the running time between the original models (when they exist), and the models modified to use subterms. For the original models, we can compare the plain model only with the Fresh Ordering as it's our only extension that can speed-up models which are not using the subterm operator. On the right side, we modify the models and use the Fresh Order and Monotonicity techniques in all combinations. We highlight some of the most significant changes implied by the individual features.

TABLE 2: Benchmark: impact of individual extensions on new and old models

4. Case Studies

In this section, we demonstrate the usefulness of our Tamarin extensions. In particular:

1) We improve existing analyses by reducing their verification time and removing the need for helper lemmas or oracles. Notably, we reduce the proving time from hours to minutes on a model of WPA2. Furthermore, we have improvements on models of 5G-AKA, YubiKey, PKCS#11, and CH'07.
2) We provide novel case studies of the TreeKEM, S/Key and Tesla Scheme 2 protocols. Here, we highlight the model of TreeKEM which is especially complex due to its tree-based data structure.

We give the high-level results in Section 4.1. We then discuss some details of improved existing models in Section 4.2, before turning to the details of the three novel case studies in Section 4.3, 4.4, and 4.5.

4.1. Overview

We describe the considered models in Table 1, including both older models that we improve on and new models that we developed from scratch. For each model, we summarize the security properties verified, the total running time of the model, how many helper lemmas needed to be specified by hand, and whether the model needs an additional handwritten oracle to help guide the proof. We used 3 threads for each run on a server with a 64 cores Intel(R) Xeon(R) CPU E5-4650L 0 @ 2.60GHz with 750GB of RAM. This scaling was mainly useful for parallelizing multiple case studies - one can also run all case studies sequentially on a normal 4-core machine and 16GB of RAM. We round to the nearest full second, except for times below 1 second, which we always round to 1. For previously existing models, we compare our patched Tamarin version to the most recent Tamarin 1.6, with the notable highlights of:

- a 30x speed-up on the CH'07 RFID protocol, a 24x speed-up on the YubiKey model and, 9x, 8x and 4x speed-up on PKCS, WPA2, and 5G-AKA,
- removing the need for an oracle in PKCS, and
- reducing the number of helper lemmas in all cases.

To evaluate the relative impact of each of our individual extensions, we also perform a more in-depth benchmark. In particular, we consider whether the fresh order rule is useful even outside the subterm context, whether dedicated natural number and subterm models without additional proof technique are already useful, and what happens when we add to the dedicated models either the Fresh Ordering rule, the Monotonicity reasonings, or both. We summarize this benchmark in Table 2. Overall, we find that

- just having a dedicated natural number model and

a subterm predicate is already useful and speeds up the 5G-AKA, YubiKey and PKCS#11 case studies (recall that it reduces the number of possible unifiers), and also enables our three new case studies;
- the Fresh Ordering rule very strongly impacts TreeKEM (without it, the verification does not terminate) and a 30x speedup for CH'07 RFID;
- the monotonicity reasonings cause a huge speed-up for WPA2 and PKCS#11.

We observe that adding only the fresh order or the monotonicity features may actually cause a slowdown compared to Tamarin 1.6, as the time they spend trying to derive contradictions may be wasted if no contradiction is found. However, we always see a significant speed-up compared to the original model when we combine all our extensions. Note that our new proof techniques never increase the proof size (not shown in the tables).

Implementation. The implementation of the subterm predicate, the natural number modeling and of all their associated proof techniques adds around 1400 lines of Haskell code to Tamarin.

The Unicode symbol $\sqsubset$ or alternatively $<<$ can be used to state a subterm predicate between two terms in a formula. Even if no subterm predicate is used inside the model, Tamarin will still use the monotonicity and the fresh ordering techniques.

As discussed previously, there are some cases where the subterm reasoning will fail in the presence of cancellation operators. This introduces a new behavior for Tamarin: previously, it would always give a positive or a negative answer when it terminated; it may now fail to conclude in some branches of the proof search. In our setting, this does not have strong consequences: for attack finding, one can continue to explore the other branches and for proving, one can try to do the proof without any of the subterm optimizations.

If users now want to use numbers in Tamarin, they have to include the builtin natural-numbers. Variables n can be typed with sort nat in two ways: n:nat or %n. The 1 must be typed to avoid clashes with the one from Diffie-Hellman, i.e., we use 1:nat or %1 for natural numbers. Finally, the addition operator is denoted by %+ to avoid clashes with the multiset operator $+$ which is denoted $+$ in Tamarin.

4.2. Speed-ups of Existing Models

WPA2. WiFi Protected Access 2 is a protocol used for securing wireless data transmission. Since 2018, there is a newer protocol WPA3 that provides additional security features, although it is to be expected that WPA2 will still used in many devices in the coming years. Because of its wide usage, WPA2 has been extensively studied, revealing multiple attacks. Among the most severe is the krack attack [33], which enabled decryption of the internet traffic of other devices if the attacker is in range of the WiFi.

[3] provides a full formal model of WPA2 in Tamarin. They formally reproduce the Krack attack and provide proofs of secrecy and authentication for a fixed version of WPA2. They additionally developed an external tool ut-tamarin which they use instead of an oracle to automate most of the proofs. However, four proofs could not be automated this way and were provided as manual proofs. Overall, the model has over 2400 lines and takes 1.5 hours to prove all automated properties.

Analyzing WPA2 is challenging because it requires modeling counters, which protect against replay attacks. The authors use the multiset encoding of numbers described in Section 3.2. To express that the counter n is smaller than m, they use an existential quantification $\exists x.\ n + x = m$. This is, among other things, used in the lemma gtk_encryption_nonces_increase_strictly_over_time which is exactly monotonicity of a counter. Unfortunately, our automatic detection of strictly increasing injective facts does not apply for this lemma because its injectivity relies on more values than the first identifier. Still, proving the lemma gets a performance improvement of 60% due to the UTVPI computation when replacing the existential quantification by the subterm operator.

Another highlight of our improvements is the longest lemma authenticator_ptk_nonce_pair_is_unique which ensures that there are no two states with the same counter - taking 1.4 hours to prove. With the monotonicity proof technique, we can not only prove this lemma instantly, but can also remove its implied lemma ptk_nonce_pair_is_unique completely without any impact on other lemmas and theorems. All in all, we reduce the proving time of the model from 1.5 hours to 9 minutes. We also remove the dependency for ut-tamarin and automate the four manual proofs with an oracle. This improves the overall usability and maintainability of the model as it only uses tools included and maintained within Tamarin.

5G-AKA. Billions of users connect to the internet via a mobile device using the cellular network provided by multiple carriers. To authenticate with a SIM-Card to the home carrier, an Authentication and Key Agreement (AKA) protocol is used. The latest such protocol is called 5G-AKA which is part of the 5G protocol standardized by the 3GPP, the successor of 4G/LTE. It provides authentication and secrecy for the following messages encrypted with the key agreed upon.

In [2], the authors provide a full analysis of 5G-AKA within 1500 lines of Tamarin code. Proving all 124 lemmas takes around 5 hours and needs oracles of an additional 1000 lines of Python code. Properties proven include especially authentication, confidentiality and privacy in a multitude of different versions including binding vs. non-binding channels. During the analysis, the authors find several weaknesses in the protocol and recommend fixes.

The main challenges in modeling 5G-AKA are the incrementing sequence numbers that persist over multiple

sessions. The authors introduce a special monotonicity lemma which is crucial for their proof, and is the largest lemma. Unfortunately, we cannot automatically detect this monotonicity with our new proof technique, as it is quite specialized. However, our extensions still reducing the proof steps needed for this lemma from over 2100 steps to under 400. Additionally, we drastically reduced the proof steps needed for lemmas connected to injective agreement and two invariant lemmas became obsolete because of the different modeling with natural numbers. Here, we focus on the main version of the protocol, comprising 15 of the total 124 lemmas and which takes 8 minutes to prove. Using our extensions, we can cut down this time to less than two minutes.

YubiKey. The YubiKey is a USB token that enables second factor authentication. After the registration of the public key of the YubiKey to a web server, every login of the user onto the web server will require the YubiKey to sign a challenge sent by the server along with a counter stored inside the YubiKey. The YubiKey mechanism was first studied with Tamarin in [30], where the counter was modelled using a multiset, and one of the key lemmas required the monotonicity of the counters. We migrated the model to our natural number modeling, which reduces the proof time from 20 seconds to 1 second.

PKCS#11. We updated the PKCS#11 key wrapping API model from the recent Tamarin analysis from [31]. The API heavily relies on a notion of integer to attach levels to secret keys, where a key of one level can only be used to encrypt keys with higher levels. The original model required the authors of [31] to write dedicated oracles to help the verification, which then ran in 74 seconds. For this model, each of our optimizations leads to a speed-up, until we reach a 9 second verification time, and we no longer need an oracle. We thus removed the need for writing an oracle, which can be a long and tedious step in a Tamarin analysis.

CH'07. The CH'07 [34] scheme is an RFID based tag authentication protocol, previously analyzed with Tamarin in [7]. It relies on a challenge-response mechanism and notably uses the xor operation, and one of its main goals is to guarantee tag unlinkability. This example offer an interesting variant from the previous examples: first, it does not use any counter, hash chain or similar constructs; second, it involves proving observational equivalence. On this example, the fresh order rule has a strong impact as it speeds up the observational equivalence proof from almost an hour to under 2 minutes.

4.3. TreeKEM

In 2018, an internet engineering task force on messaging layer security (IETF-MLS) was founded to standardize the key exchange for messaging apps. End-to-end encrypted messaging between two parties is already standard for most messengers. However, in a group setting, the currently employed protocols either lack security or performance. This is what the MLS working group aims to provide with a continuous group key agreement (CGKA) protocol based on the TreeKEM protocol [24]. A CGKA is a protocol to derive a group key in an updatable way, i.e., if group members join or leave the group or just want to renew the secrecy of the group key. This update mechanism aims to achieve two main properties:

1) Forward secrecy states that if the attacker compromises a key at a certain point in time, the previous keys are still secret.
2) Post compromise secrecy (PCS) states that if the attacker compromises a key, participants can heal the key when the attacker is temporarily passive. That is, if the participants update the key, then the new key is again secret.

The main complexity of TreeKEM arises from the use of a distributed tree structure. A private/public key pair is saved inside each node of a binary tree. Each leaf of the tree corresponds to a participant in the group, and each participant knows exactly the secret keys on the path from their leaf to the root. This implies that the secret key on a leaf is only known by the corresponding participant while the key at the root node is known by all participants. This root key is then the one used to derive a shared group key.

If a participant wishes to update their leaf key, they will update all the secrets on the path from their leaf to the root key, notably updating the root key, and sending the information needed for the updates to the other members by using the public key of each node. After an update, the new shared group key is computed through the application of a key derivation over both the previous group key and the new root key.

Modeling and analyzing the TreeKEM protocol raises two challenges:

- a participant needs to store a list of key pairs of an arbitrary size, and be able to go through all of them to update it;
- the group key is produced through an infinitely growing hash chain.

We address the first challenge by using a Tamarin model of ordered lists that relies on our implementation of natural numbers, and the second one by using the subterm predicate.

Our model. From the version proposed in [24], we take the following parts to model a single group with an unbounded number of participants:

1) a rule to create a group with one participant,
2) rules for a new participant to join a group,
3) rules for a participant to update their secrets, and
4) a theorem proving forward secrecy.

The natural numbers allowed us efficiently model the storage of the path of secrets from the leaf to the root of the tree as an ordered list. Tamarin does not directly support this data structure, so it has to be built out of smaller primitives. Without our extension, a natural choice would be to model it with pairs, e.g., the list

$[a, b, c, d]$ would be represented as $\langle a, \langle b, \langle c, d\rangle\rangle\rangle$. However, this has the disadvantage that elements at the end or in the middle of the list can only be accessed by a loop which deconstructs the structure. A smarter way is to use a multiset with indices, i.e., $\langle 1, a\rangle + \langle 2, b\rangle + \langle 3, c\rangle + \langle 4, d\rangle$. There, we can access any element by pattern matching on it with the index n: $\langle n, \mathit{elem}\rangle + \mathit{rest}$ and iterate trough the list by incrementing n. With such a model of the data structure, we were able to encode all the loops that need to go through the list, e.g., to update the values, in an efficient way.

Proving Forward Secrecy. We specify forward secrecy formally as follows: the group key gk of participant id cannot be known by the attacker if no participant of the group is compromised in a state where their group key gk_2 was a predecessor of gk. We write the corresponding Tamarin lemma as:

Tamarin Lemma 1 (TreeKEM Forward Secrecy)**.**

$$\begin{aligned}\forall\, id, gk, i.\ &\texttt{GroupKey}(id, gk)@i \\ &\Rightarrow \neg\exists\ id_2, gk_2, j.\ (\texttt{Leak}(id_2, gk_2)@j \\ &\qquad\qquad \&\ gk_2 \sqsubset gk) \\ &\quad\Rightarrow \neg(\exists\ l.\ \texttt{K}(gk)@l)\end{aligned}$$

Note that it does not make sense to refer to time-wise constraints like "if there was no compromise before, the attacker will never know gk". With that, it could be that the attacker compromises a client in the future that did not receive updates and is thus in an old state. Instead, we use the group key to express the progress of a client. As group keys are computed from old group keys with a key derivation function $\mathit{gkNew} = \mathit{kdf}(\mathit{gkOld}, \mathit{rootSecret})$, it is the case that $\mathit{gkOld} \sqsubset \mathit{gkNew}$, even for arbitrarily old group keys. This is an interesting setting where the subterm predicate is not only a proof technique but actually needed to specify the security property in some intelligible and straightforward way.

When first trying to prove forward secrecy for TreeKEM, Tamarin found a trace that contradicted the security property. The attack came from the fact that in the original specification, the first group key is initialized with a public constant instead of a fresh random variable, and as long as no update was performed and only new group members were added, the protocol would not provide forward secrecy. By comparing with the current draft of the MLS standard, we saw that the issue had already been discovered manually and fixed in draft 10 of the MLS standard.[2] After applying the fix, we were able to prove the forward secrecy property. It required writing four helper lemmas, which is relatively low, and the proof runs in a few seconds.

Limitations. Note that we do not model deletion of group members. Moreover, we only have a restricted join operation that inserts new participants always on the right side of the tree which yields non-balanced binary trees. The most severe limitation of our protocol is that we were not able to prove PCS. There are two challenges to a PCS proof for TreeKEM, that we are not sure how to address yet, and consider out of scope of this paper:

- PCS relies on an invariant over the whole tree structure which is unfortunately distributed over arbitrarily many clients. Expressing such an invariant over a structure which is abstract and never explicitly occurs inside a state is complex in Tamarin.
- In addition, TreeKEM clients in the same group can be widely out of sync for many steps, which makes it a challenge to express a meaningful notion of PCS while accounting for the temporal dependencies.

2. https://github.com/mlswg/mls-protocol/pull/385

4.4. S/Key

The S/Key protocol uses a hash chain to provide a One Time Password (OTP) authentication scheme [25] integrated into the Linux kernel. The user first generates the hash chain $h^n(\mathit{password})$, keeps all the iterations, and provides only the first element $h^n(\mathit{password})$ to the server through a secure channel. Then, at step i, $h^{n-i}(\mathit{password})$ is given to the server which can check if the hash of the given value matches its stored value and keep this hash as the new stored value.

While it is one of the most classical OTP schemes, it was never automatically analyzed before due to the complexity of the arbitrary large – first increasing and then decreasing – hash chain. This is a case for which writing down the protocol is deceptively simple and only takes a dozen of lines, yet the proof is involved. For this protocol, we prove the authentication property that specifies that the server will only accept a token for which the user explicitly revealed a previous value. If we denote by $\texttt{User}(x)$ the action raised when a user is using the value x from the chain to try to authenticate (thus explicitly revealing it), and by $\texttt{Server}(x)$ the action raised by the server accepting a chain value, we prove authentication for S/Key:

Tamarin Lemma 2 (S/Key Authentication)**.**

$$\begin{aligned}\forall\, x, i.\ &\texttt{Server}(x)@i \Rightarrow \\ &\exists\, y, j.\ \texttt{User}(y)@j\ \&\ j < i\ \&\ (x = y \mid y \sqsubset x)\end{aligned}$$

Similarly to the TreeKEM case, this illustrates how the subterm predicate can help express complex security properties. We prove the previous security property (lifted to an unbounded number of sessions) in a few seconds with one helper lemma.

4.5. TESLA Scheme 2

TESLA Scheme 2 [26] is a stream authentication protocol. From a high-level point of view, it can be seen as using the basic S/Key concept as a building block, but turned into a full-fledged system where each part of the hash chain authenticates a sequence of messages. The expected security is the authenticity of each message accepted by the server, which is expressed as:

Tamarin Lemma 3 (Tesla Authentication)**.**

$$\forall\ m, i.\ \texttt{Accept}(m)@i \Rightarrow \exists\ j.\ \texttt{Sent}(m)@j\ \&\ j < i$$

A first Tamarin model of it was proposed in 2012 [27], as an example of things that Tamarin was not able to prove. The reason for that is the complex construction with an inverse hash chain which might skip an arbitrary number of intermediate steps. With subterms, we were able to express such a skip and write helper lemmas, which especially expressed monotonicity and uniqueness. Here, the subterms are only used as an intermediate proof technique to prove a generic and subterm independent property. We were able to prove authentication as well as some additional secrecy requirements in about 5 seconds with 5 helper lemmas.

5. Conclusions

We extend the Tamarin prover with a subterm predicate and multiple associated proof techniques, as well as a dedicated support for natural numbers. We illustrate on multiple case studies how this improves the automation and the scope of the tool, providing both speed-ups of old models and novel case studies.

Our extensions have significant impact on our case studies: our techniques can enable verification which had not succeeded before (getting rid of non-termination in, e.g., TreeKEM), removing the need for manually-specified helper lemmas or oracles, and we observed a speed-up factor of over 30x in one case (CH'07 RFID).

While our techniques were initially developed for Tamarin, they appear to be rather generic and it should be possible to, e.g., introduce a general subterm predicate to ProVerif that can be used in lemmas and queries.

References

[1] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A Comprehensive Symbolic Analysis of TLS 1.3," in *ACM Conference on Computer and Communications Security*. ACM, 2017, pp. 1773–1788.

[2] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5G authentication," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 1383–1396.

[3] C. Cremers, B. Kiesl, and N. 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 20)*, 2020, pp. 1–17.

[4] D. Basin, R. Sasse, and J. Toro-Pozo, "The EMV standard: Break, fix, verify," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1766–1781.

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

[6] C. Cremers and D. Jackson, "Prime, order please! Revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 78–7815.

[7] J. Dreier, L. Hirschi, S. Radomirovic, and R. Sasse, "Automated unbounded verification of stateful cryptographic protocols with exclusive or," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 359–373.

[8] B. Schmidt, R. Sasse, C. Cremers, and D. Basin, "Automated verification of group key agreement protocols," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 179–194.

[9] J. Dreier, C. Duménil, S. Kremer, and R. Sasse, "Beyond subterm-convergent equational theories in automated verification of stateful protocols," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 117–140.

[10] V. Cortier, S. Delaune, and J. Dreier, "Automatic generation of sources lemmas in Tamarin: towards automatic proofs of security protocols," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 3–22.

[11] D. Jackson, C. Cremers, K. Cohn-Gordon, and R. Sasse, "Seems legit: Automated analysis of subtle attacks on protocols that use signatures," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2165–2180.

[12] D. Basin, J. Dreier, and R. Sasse, "Automated symbolic proofs of observational equivalence," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1144–1155.

[13] J. A. Clark and J. L. Jacob, "A survey of authentication protocol literature: Version 1.0," 1997.

[14] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016.

[15] V. Cheval, V. Cortier, and M. Turuani, "A little more conversation, a little less action, a lot more satisfaction: Global states in ProVerif," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 344–358.

[16] B. Blanchet, V. Cheval, and V. Cortier, "Proverif with lemmas, induction, fast subsumption, and much more," in *42nd IEEE Symposium on Security and Privacy (S&P'22)*, 2022.

[17] "ProVerif Manual, version 2.04," https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf, visited May 2022.

[18] C. Cremers, "The Scyther Tool: Verification, falsification, and analysis of security protocols," in *International conference on computer aided verification*. Springer, 2008, pp. 414–418.

[19] J. D. Guttman, "Shapes: Surveying crypto protocol runs," in *Formal Models and Techniques for Analyzing Security Protocols*. IOS Press, 2011, pp. 222–257.

[20] M. D. Liskov, J. D. Ramsdell, and J. D. Guttman, "CPSA: A Cryptographic Protocol Shapes Analyzer," *The MITRE Corporation*, 2016.

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

[22] C. Staub, "Adding support for user-defined sorts and sorted function symbols to Tamarin," Master's thesis, ETH Zürich, 2013.

[23] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," *Journal of Computer Security*, vol. 24, no. 5, pp. 583–616, 2016.

[24] K. Bhargavan, R. Barnes, and E. Rescorla, "TreeKEM: Asynchronous decentralized key management for large dynamic groups," p. 20, 2018, https://hal.archives-ouvertes.fr/hal-02425247/.

[25] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, vol. 24, no. 11, pp. 770–772, 1981.

[26] A. Perrig, R. Canetti, J. D. Tygar, and D. Song, "Efficient authentication and signing of multicast streams over lossy channels," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE, 2000, pp. 56–73.

[27] S. Meier, "Advancing automated security protocol verification," Ph.D. dissertation, ETH Zurich, 2013.

[28] C. Cremers, C. Jacomme, and P. Lukert, "Extended Tamarin-prover and case-studies," https://cispa.saarland/group/cremers/tamarin/subterm/index.html.

[29] ——, "Subterm-based proof techniques for improving the automation and scope of security protocol analysis (extended version)," Cryptology ePrint Archive, Paper 2022/1130. [Online]. Available: https://eprint.iacr.org/2022/1130

[30] R. Künnemann and G. Steel, "YubiSecure? Formal security analysis results for the Yubikey and YubiHSM," in *International Workshop on Security and Trust Management*. Springer, 2012, pp. 257–272.

[31] A. Dax, R. Künnemann, S. Tangermann, and M. Backes, "How to Wrap it up-A Formally Verified Proposal for the use of Authenticated Wrapping in PKCS# 11," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 62–6215.

[32] S. K. Lahiri and M. Musuvathi, "An efficient decision procedure for UTVPI constraints," in *International Workshop on Frontiers of Combining Systems*. Springer, 2005, pp. 168–183.

[33] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in WPA2," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1313–1328.

[34] H.-Y. Chien and C.-W. Huang, "A lightweight RFID protocol using substring," in *International Conference on Embedded and Ubiquitous Computing*. Springer, 2007, pp. 422–431.