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

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]

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

1. An extended abstract of this paper appears at IEEE CSF 2023. This is the full version.

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 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[2], 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 multiple appendices.

2. Background: The Tamarin Prover

We now introduce the theoretical background needed for our extensions.

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

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.

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: $\mathtt{Fr}(\sim n)$ to sample fresh random values such as nonces and keys, $\mathtt{Out}(t)$ and $\mathtt{In}(t)$ are used for outputs to and inputs from the attacker-controlled network, and $\mathtt{K}(t)$ 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.*

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

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

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

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 $(l\text{-}[a]\!\!\rightarrow r)$ S' is possible when $l\text{-}[a]\!\!\rightarrow r$ is a valid instantiation of a rule $l_x\text{-}[a_x]\!\!\rightarrow r_x$ from RU, i.e., there exists a substitution σ from variables to ground terms such that $l\text{-}[a]\!\!\rightarrow r = l_x\sigma\text{-}[a_x\sigma]\!\!\rightarrow r_x\sigma$, $l \subset S$ 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:

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

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*

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

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.*

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.

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

Figure 1: The $S_{\approx}$ rule

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 $x \mapsto \mathit{fun}(s)$ 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 $s \sqsubset \mathit{fun}(s)$ 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 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 $\mathit{fst}(\langle x, y \rangle) \rightarrow_{R,AC} x$ 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 $x \mapsto \mathit{fun}(s)$ 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 is proved in Appendix C.

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

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 Appendix A.

AC-RECURSE:

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

- *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.

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 Appendix B.

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], [29], [30]. 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

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 $\texttt{Fr}(s)$,*
- *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

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 $+ : \mathsf{nat} \times \mathsf{nat} \rightarrow \mathsf{nat}$. 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 [31].

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 $\exists x.\ n + x = m$.

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 $\texttt{Fr}(s)$. We describe below informally two variants of this first rule, and provide the full rule as well as soundness and completeness proofs in Appendix D.

"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 Appendix E.

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* $\texttt{Fact}(id, \ldots)$ *of each rule, there is no other conclusion* $\texttt{Fact}(id, \ldots)$ *with the same first term and*
   a) *either there is a premise* $\texttt{Fr}(id)$
   b) *or there is exactly one premise* $\texttt{Fact}(id, \ldots)$

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.

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 [29]	Authentication, Replay-Resistance	134	4	3	19	1	no	no
PKCS#11 [30]	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 [29]	19	21	**1**	1	1	1
PKCS#11 [30]	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

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).

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 $\mathbin{+\!\!\!+}$ 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 [32], 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, inter alia, 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 [29], 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 [30]. 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 [30] 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 [33] 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
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)**.**

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

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.[3] 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.

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)**.**

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

Similarly to the TreeKEM case, this illustrates how the subterm predicate can be 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.

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

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, Eidgenössische Technische Hochschule 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] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] B. Schmidt, "Formal analysis of key exchange protocols and physical protocols," Ph.D. dissertation, ETH Zurich, 2012.

[35] S. Escobar, R. Sasse, and J. Meseguer, "Folding variant narrowing and optimal variant termination," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 7-8, pp. 898–928, 2012.

[36] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[37] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

Appendix A. Detailed Preliminaries

The foundations of Tamarin are defined in the PhD thesises of Benedict Schmidt [34] and Simon Meier [27]. As they differ slightly, we will stick to Schmidts thesis for notation. For a more detailed introduction to the equational theory, we refer to [35]. In the remainder of this section, we introduce the formal definitions and give an introduction to the foundations of Tamarin.

A.1. Terms and Equations

We use **terms** to represent messages. They are drawn from an order-sorted term algebra with $sorts = \{msg, fresh, pub\}$ where msg is the top sort with all others being incomparable subsorts of it described by the relation $\leq_S$. The sort $fresh$ is used for random values and pub for public names. The functions of the term algebra are given by $\Sigma_{base} = \{h, enc, dec, \langle \cdot, \cdot \rangle, fst, snd, \dots\}$ where Σ_{base} contains the operators for hashing, encryption, pairs, Diffie-Hellman and more., which are defined in [34]. In Tamarin 1.6, all functions have the signature $msg \times \ldots \times msg \to msg$.

The **variables** of the algebra are drawn from the sets V_s where s denotes the sort. Sometimes, we use $V = \bigcup_{s \in sorts} V_s$ for all variables. The names are countably infinite sets N_s and N similarly to the variables. They provide an infinite set of constant values. Summarizing these parts, we call T_{base} or (if V, N not clear from the context) $T_{\Sigma_{base}}(V, N)$ the set of terms and TG_{base} or $T_{\Sigma_{base}}(N)$ the set of ground terms that do not contain variables. When using a variable or name x, we always denote its sort by $x : s$. With this, we define a sort function S, which returns the sort of names and variables $S(x : s) = s$. With the function signatures given in Σ_{base}, we can lift S to terms. A function $f : s \to s'$ can only be applied to a term t if $S(t) \leq_S s$.

A **substitution** $\sigma : V \to T_{base}$ maps variables to terms and can be lifted to terms by replacing each variable in a term by its substitution. A substitution σ is called sort-decreasing if for all $\sigma(x : s) = t$ it holds that $S(t) \leq_S s$. In this paper, all substitutions, we encounter, are sort-decreasing.

An **equation** is an unoriented pair $t = t'$ with $t, t' \in T_{base}$. An equation is sort-preserving if for all substitutions σ it holds that $S(t\sigma) = S(t'\sigma)$. We use equations in two situations. One is to specify security properties where we perform unification. Secondly, to define an equational theory. An equational theory is a set of equations E, which induce a congruence relation $=_E$ on T_{base}.

An E**-unifier** for an equation $t = t'$ is a substitution σ 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. Further explanation of this can be found in [35].

A **rewrite rule** is an oriented equation $l \to r$ with $l, r \in T_{base}$ and $l \notin X$. Intuitively, a rewrite rule replaces a subterm l of a bigger term by r. It is sort-decreasing if for all substitutions σ it holds that $S(r\sigma) \leq_S S(l\sigma)$. A set of rewriting rules R form a rewriting relation $\to_R$. A transition $t \to_R t'$ is possible if there is a position p in t, a substitution σ and a rule $l \to r \in R$ such that $t|_p = l\sigma$ and $t' = t[r\sigma]_p$, i.e., the subterm $t|_p$, which matches $l\sigma$, gets replaced by $r\sigma$. The rewriting relation is terminating if there is no infinite chain $t_1 \to_R t_2 \to_R \cdots$. It is confluent if for all $t_1 \ {}_R^*\!\!\leftarrow t \to_R^* t_2$ there is a t_3 with $t_1 \to_R^* \ t_3 \ {}_R^*\!\!\leftarrow t_2$ where $\to_R^*$ denotes the transitive and reflexive closure of $\to_R$. If we have a confluent and terminating rewriting relation, we can define a unique canonical form of a term t. We do this by executing arbitrary rewriting steps until no more can be applied. Confluence guarantees uniqueness of the resulting term t', which we denote by $t{\downarrow_R}$.

Rewriting modulo is necessary if we have an equational theory for which the rewriting rules hinder confluence and termination. For example, associativity and commutativity $AC = \{(x + y) + z = x + (y + z), x + y = y + x\}$, which are essential in Tamarin. Luckily, we have a practical unification algorithm for AC, which we explicitly use during rewriting. Doing so, we can remove AC from the rewriting rules R and reclaim confluence and termination while handling AC externally with the given unification algorithm. More general, we do rewriting modulo a set of equations Ax (Axioms), which are in our case AC. A modulo rewriting step $t \to_{R,Ax} t'$ is possible if there is a position p in t, a substitution σ and a rule $l \to r \in R$ such that $t|_p =_{Ax} l\sigma$ and $t' = t[r\sigma]_p$. The only difference to normal rewriting is that we check for equality $t|_p =_{Ax} l\sigma$ modulo Ax. We define the same notions of confluence, termination and canonical forms for this relation.

An R, Ax**-variant** of a term t is a pair (t', σ) where t' is the canonical form $t\sigma\downarrow_{R,Ax}$ of t after applying σ. Formally, we define the set of variants $\lceil t \rceil^*_{R,Ax} = \{(t', \sigma) | \sigma \text{ is a substitution and } t\sigma{\downarrow_R} =_{Ax} t'\}$. If R, Ax is clear, we write $\lceil t \rceil^*$. Similarly to the unifiers, we do not need the whole (usually infinite) set of variants as most of them are instances of some most general variants. Thus, we only consider *complete* subsets of them, which is explained in [35]. These are denoted $\lceil t \rceil_{R,Ax}$ without a star. In Tamarin, we require that the rewriting theory R, Ax has the finite variant property (FVP), i.e., there is a finite complete set of variants. These variants can be used to compute unifiers as follows:

Lemma 2 (Variant-based Unification). *A substitution σ is a unifier for $t = t'$ iff $\exists(\_, \sigma) \in \lceil t \rceil^* \cup \lceil t' \rceil^*$.*

Analogously, we can compute a complete set of unifiers with a complete set of variants [35]. As we assume the FVP for our equational system, we can compute the complete set of variants and thus the CSU.

We will often talk about **multisets**, which are similar to sets but can contain elements multiple times. Given a set A, we denote the set of multisets of A with $A^{\#}$, which is analogous to the powerset $\mathcal{P}(A)$ for normal sets. Similarly, we can lift several operators to multisets, which we know from normal sets. For instance, we define $\subseteq^{\#}$, $\cup^{\#}$, and $\setminus^{\#}$ for multiset-subset, -union, and -difference in the straightforward way.

A.2. Protocols

We model protocols with multiset rewriting rules over facts. Facts are built of terms in T_{base} and have names from a set Σ_{base}, which is partitioned into two types called persistent and linear facts. From these names, we build the set of facts $\mathcal{F} = \{F(t_1, \ldots, t_n) \mid t_i \in T_{base}, F \in \Sigma_{base}\}$. If we restrict ourselves to ground terms in TG_{base}, we call the set $\mathcal{G}$.

A protocol rule is a tuple $ri = (id, l, a, r)$ written $id : l\text{--}[a]\!\!\rightarrow r$ where id is a unique name and $l, a, r \in \mathcal{F}^{\#}$. To extract properties from this tuple, we define $name(ri) = id$ for the name, $prems(ri) = l$ for the premises, $acts(ri) = a$ for the actions, and $concs(ri) = r$ for the conclusions. The actions a are later used for specifying security properties. An E-instance of a rule r is a rule r' for which there is a substitution σ with $r' =_E r\sigma$ where $r\sigma$ treats rules as terms to define the substitution application. For a set of rules P, which we denote a protocol, we define the set of ground instances $groundi_E(P)$ as all E-instances of $ri \in P$ that are ground, i.e., do not contain variables. Lastly, we can define the labeled transition relation $\to_P \ \subseteq \ \mathcal{G}^{\#} \times \mathcal{G}^{\#} \times \mathcal{G}^{\#}$, which is the backbone of Tamarin:

$$\frac{ri = id : l\text{--}[a]\!\!\rightarrow r \in_E groundi_E(P) \qquad linear(l) \subseteq^{\#} S \qquad pers(l) \subseteq S}{S \xrightarrow{\mathrm{a}}_P ((S \setminus^{\#} linear(l)) \ \cup^{\#} \ r)}$$

where S is the current state and $linear(l)$ and $pers(l)$ are the linear/persistent facts of l. Note that the relation operates modulo E, which is why we use $\in_E$. In general, a rule transforms a state S to a successor state S' by removing the linear facts of l and adding the facts r. Here, we see that linear facts get consumed by applying a rule while persistent facts always remain in the state once added. For getting familiar with these rules, we define the built-in message deduction rules *MD*:

$$\begin{aligned}
MD = \{ & \mathit{Fresh} : \emptyset \negmedspace-\negmedspace[] \negmedspace\rightarrow \mathtt{Fr}(x : \mathit{fresh}), \\
& \mathit{Output} : \mathtt{Out}(x) \negmedspace-\negmedspace[] \negmedspace\rightarrow K(x), \\
& \mathit{Input} : K(x) \negmedspace-\negmedspace[K(x)] \negmedspace\rightarrow \mathtt{In}(x), \\
& \mathit{FreshK} : \mathtt{Fr}(x : \mathit{fresh}) \negmedspace-\negmedspace[] \negmedspace\rightarrow K(x : \mathit{fresh}), \\
& \mathit{Public} : \emptyset \negmedspace-\negmedspace[] \negmedspace\rightarrow K(x : \mathit{pub}) \} \\
\cup \{ & \mathit{Construct}_f : K(x_1), \ldots, K(x_n) \negmedspace-\negmedspace[] \negmedspace\rightarrow \\
& K(f(x_1, \ldots, x_n) \mid f \in \Sigma_{base} \text{ with arity } n \}
\end{aligned}$$

These rules introduce some facts, which we will use frequently. $\mathtt{Fr}()$ denotes a fresh random variable and can never appear on the right side of a rule except for the *Fresh* rule. $\mathtt{In}()$ and $\mathtt{Out}()$ facts model receiving/sending on network with a Dolev-Yao adversary [36], which learns all messages and can insert arbitrary messages. This is modeled by the rules *Output* and *Input* where the fact K represents the knowledge of the attacker. Note that other types of networks with weaker adversaries can be modeled as well. Additional capabilities of the attacker are generating random variables (*FreshK*), knowing public constants (*Public*) and building terms out of known subterms ($\mathit{Construct}_f$).

With the labeled transition relation from these rules, we define a labeled transition system whose initial state is the empty multiset $S_0 = \emptyset$ and which transforms a multiset S according to the relation. An execution in this transition system is a sequence of states S_i, which are connected by rules:

$$S_0 \ (l_1 \negmedspace-\negmedspace[a_1] \negmedspace\rightarrow r_1) \ S_2 \ \ldots \ S_{n-1} \ (l_n \negmedspace-\negmedspace[a_n] \negmedspace\rightarrow r_n) \ S_n$$

Here $(S_{i-1}, (l_i \negmedspace-\negmedspace[a_i] \negmedspace\rightarrow r_i), S_i)$ must be a valid step. The trace associated with this execution is $a_1, \ldots, a_n$. We denote the set of all traces of a protocol P as $\mathit{traces}(P)$. Over these traces, we define a security property in TAMARIN.

A.3. Construction and Deconstruction

The rules *MD* allow for many alternative deductions of specific K-facts. Consider for example the decryption $\mathit{dec}(\mathit{enc}(x, k), k) = x$. The attacker can, given a term $K(t)$, construct $K(\mathit{enc}(t, t))$ and then, using $K(t)$ again, $K(\mathit{dec}(\mathit{enc}(t, t), t)$, which is the same as $K(t)$ from the start. This is not a problem of the model but can become a problem for efficiency and termination of the search algorithm. To avoid this, we split the attacker into a deconstruction and then construction phase. Deconstruction concerns retrieving subterms from more complex terms such as a t out of an $\mathit{enc}(t, \mathit{key})$. Once all deconstruction is finished, the attacker continues with constructing terms by combining the resulting minimal terms from the deconstruction. With this strategy, it is not possible to alternatingly construct and destruct.

The formalization of this uses $K^{\downarrow}$ facts for deconstruction and $K^{\uparrow}$ for construction instead of the generic attacker knowledge K. For simplicity, we omit the exponentiation rules K^{exp} and refer to [34]. To work with this separation between $K^{\downarrow}$ and $K^{\uparrow}$, we need the rule *Coerce*, which transforms $K^{\downarrow}$ to $K^{\uparrow}$ (but not the other way!). All in all, the normal deconstruction rules *ND* look as follows:

$$\begin{aligned}
ND = \{ & \mathit{Fresh} : \emptyset \negmedspace-\negmedspace[] \negmedspace\rightarrow \mathtt{Fr}(x : \mathit{fresh}), \\
& \mathit{Output} : \mathtt{Out}(x) \negmedspace-\negmedspace[] \negmedspace\rightarrow K^{\downarrow}(x), \\
& \mathit{Input} : K^{\uparrow}(x) \negmedspace-\negmedspace[K(x)] \negmedspace\rightarrow \mathtt{In}(x), \\
& \mathit{FreshK} : \mathtt{Fr}(x : \mathit{fresh}) \negmedspace-\negmedspace[] \negmedspace\rightarrow K^{\uparrow}(x : \mathit{fresh}), \\
& \mathit{Public} : \emptyset \negmedspace-\negmedspace[] \negmedspace\rightarrow K^{\uparrow}(x : \mathit{pub}), \\
& \mathit{Coerce} : K^{\downarrow}(x) \negmedspace-\negmedspace[] \negmedspace\rightarrow K^{\uparrow}(x), \\
& \mathit{Decrypt} : K^{\downarrow}(\mathit{enc}(x, k)), K^{\uparrow}(k) \negmedspace-\negmedspace[] \negmedspace\rightarrow K^{\downarrow}(x) \} \\
\cup \{ & \mathit{Construct}_f : K^{\uparrow}(x_1), \ldots, K^{\uparrow}(x_n) \negmedspace-\negmedspace[] \negmedspace\rightarrow \\
& K^{\uparrow}(f(x_1, \ldots, x_n) \mid f \in \Sigma_{base} \text{ with arity } n \}
\end{aligned}$$

Of course, these rules are not complete. Each equation can add - possibly multiple - deconstruction rules. For example, the rule *Decrypt* was synthesized from the equation $\mathit{dec}(\mathit{enc}(x, k), k) = x$. The process of synthesizing rules from equations is explained in [34]. Note that the *Decrypt* has the *key* as a $K^{\uparrow}$ fact, which was constructed. At least one (usually the first) premise of a deconstruction rule, though, must always be a $K^{\downarrow}$ fact to prevent looping between deconstruction rules, coerce and construction rules.

A.4. Dependency Graphs

A dependency graph modulo E is a compact representation of an execution of a protocol P. It is a tuple (I, D) with $I \in \mathit{groundi}_E(P \cup \mathit{MD})^*$ a sequence of rules and $D \in \mathbb{N}^2 \to \mathbb{N}^2$ representing edges between facts of these rules, e.g., the edge $(i, u) \rightarrowtail (j, v) \in D$ denotes a dependency from the u-th conclusion of rule I_i to the v-th premise of the rule I_j. The following axioms must always hold:

DG1 For $(i, u) \rightarrowtail (j, v) \in D$ it holds that $i < j$ and conclusion (i, u) is syntactically equal to premise (j, v).
DG2 Every premise has exactly one incoming edge.
DG3 Every linear conclusion has at most one outgoing edge.
DG4 The *Fresh* instances are unique.

For a protocol P we write $\mathit{dgraphs}_E(P)$ as the set of all possible dependency graphs.

The trace of a dependency graph $dg = (I, D)$ with $I = [l_1 \text{-}[a_1]\!\!\rightarrow r_1, \ldots, l_k \text{-}[a_k]\!\!\rightarrow r_k]$ is the sequence $[a_1, \ldots, a_n]$. The traces of a set of dependency graphs DG is denoted $\mathit{traces}(DG)$ In the following lemma, we will see that this definition coincides with the original definition of traces.

Lemma 3. *For all protocols P, we have $\mathit{traces}_E(P) = \mathit{traces}(\mathit{dgraphs}_E(P))$.*

We now integrate the concept of construction and deconstruction to dependency graphs and call them *normal dependency graphs*. To do so, we replace MD by ND to get $I \in \mathit{groundi}_E(P \cup \mathit{ND})^*$. Additionally, we need a set of thirteen normal form conditions **N1-N13**, joining the old conditions **D1-D4**. We will explain them in Section B.2. These normal form conditions are used in the algorithm to cut down branches that are equivalent to already explored branches. This greatly improves efficiency. Additionally, they can ease termination, though, never guarantee it as the problem itself is undecidable.

A.5. Security Properties

As the security properties refer to the actions on the traces, they are called trace properties. When talking about an action fact f, we always connect it to a time point i by writing $f@i$. This enables us to relate the time points of two facts $f@i$, $g@j$ with a relation $\prec$. $i \prec j$ states that the timepoint i was before j, i.e., f is in the trace before g. As domain for time points, we choose $\mathbb{Q}$ (and not $\mathbb{N}$). This has the advantage that we can always squeeze a time point in between two (distinct) others. Taking $\mathbb{Q}$, we build an assignment function θ, which maps time points to $\mathbb{Q}$ and all other variables in a sort-decreasing way to ground terms TG_{base}. With this assignment function, we can define the satisfaction relation $(tr, \theta) \models_E \phi$ stating that for the equational theory E, the trace tr satisfies the trace property ϕ with the valuation θ.

$(tr, \theta) \models_E i \prec j$	if $\theta(i) < \theta(j)$
$(tr, \theta) \models_E i \approx j$	if $\theta(i) = \theta(j)$
$(tr, \theta) \models_E t \approx t'$	if $t\theta =_E j\theta$
$(tr, \theta) \models_E f@i$	if $f \in tr[\theta(i)]$
$(tr, \theta) \models_E \neg\phi$	if $(tr, \theta) \models_E \phi$ does not hold
$(tr, \theta) \models_E \phi \wedge \phi'$	if $(tr, \theta) \models_E \phi \wedge (tr, \theta) \models_E \phi'$
$(tr, \theta) \models_E \phi \vee \phi'$	if $(tr, \theta) \models_E \phi \vee (tr, \theta) \models_E \phi'$
$(tr, \theta) \models_E \exists x : s.\ \phi$	if there is a term u of sort s such that $(tr, \theta[x \mapsto u]) \models_E \phi$

In TAMARIN, we can use properties either in a universal or an existential context. In the universal context (validity), we require for all traces $tr \in \mathit{traces}(P)$ that $\forall\theta.\ (tr, \theta) \models_E \phi$ while in the existential context (satisfiability), we require for at least one trace in $\mathit{traces}(P)$ that $\exists\theta.\ (tr, \theta) \models_E \phi$. Note that, in the satisfiability context, the existential quantification of θ implies an existential quantification for free variables, should they appear in ϕ.

A.6. Constraint Solving

The ultimate aim of TAMARIN is to check these validity and satisfiability properties with respect to a protocol. As usual, we reduce validity to satisfiability by negating the formula. Therefore, we only have to answer satisfiability queries of the form "is there a trace, such that the property holds?". With Lemma 3, we can transform this to "is there a normal dependency graph on which the property holds?" This dependency graph can be constructed incrementally by the so-called constraint solving.

The idea of constraint solving is to have a set of constraints, which we transform with constraint solving rules. Constraints are ground rules (nodes of a dependency graph), edges or formulas. We start with the set containing only the formula to prove. Subsequently, we transfer the formula parts into information about a dependency graph, creating nodes and edges. After having processed all formula parts, we have a dependency graph left. During all these steps, we take care that a dependency graph, satisfying the formulas and containing the nodes and edges in the multiset, still does so after one step of the constraint solving relation, i.e., we do not loose or add solutions. More specifically, we require soundness and completeness.

Going into more detail, constraints can also be a "provides" $i \triangleright f$, which denotes that a node i provides the fact f and deconstruction chain $i \twoheadrightarrow (j, v)$, which is discussed extensively in [34] in addition to the already mentioned nodes $i : ri$, edges $(i, u) \rightarrowtail (j, v)$ and formulas ϕ. With this notation, we can define the satisfaction relation $(dg, \theta) \Vdash C$. A normal dependency graph $dg = (I, D)$ satisfies a set of constraints Γ if there is a valuation function θ such that $(dg, \theta) \Vdash C$ for all $C \in \Gamma$.

$$\begin{array}{ll}
(dg,\theta) \Vdash_E i : ri & \text{if } \theta(i) \in \mathit{indices}(I) \\
 & \text{ and } ri\theta =_{AC} I_{\theta(i)} \\
(dg,\theta) \Vdash_E (i,u) \rightarrowtail (j,v) & \text{if } (\theta(i),u) \rightarrowtail (\theta(j),v) \in D \\
(dg,\theta) \Vdash_E \phi & \text{if } (\mathit{trace}(dg),\theta) \models_{AC} \phi \\
(dg,\theta) \Vdash_E i \; \triangleright f & \text{if } \theta(i) \in \mathit{indices}(I) \\
 & \text{ and } (\mathit{concs}(I_{\theta(i)}))_1 =_{AC} f\theta \\
(dg,\theta) \Vdash_E i \twoheadrightarrow (j,v) & \text{if } \theta(i) \twoheadrightarrow (\theta(j),v)
\end{array}$$

A constraint rewriting rule is a rule from a constraint set to a finite set of constraint sets $\Gamma \rightsquigarrow_P \{\Gamma_1, \dots, \Gamma_k\}$. This is to allow for case distinctions. We can abuse notation and define the reflexive, transitive closure $\Gamma \rightsquigarrow_P^* \{\Gamma_1, \dots, \Gamma_k\}$. An application of a rule $\Gamma_i \rightsquigarrow_P \{\Gamma_{i,1}, \dots, \Gamma_{i,k}\}$ to a set $\Delta = \{\Gamma_1, \dots, \Gamma_k\}$ then intuitively removes Γ_i and inserts $\Gamma_{i,1}, \dots, \Gamma_{i,k}$ into Δ.

To ease the definition of rules we divide them into two types: modification and insertion rules. While the latter are only allowed to add new constraints, modification rules can also change existing constraints where it is necessary.

$$\frac{c_1, \dots, c_l}{\Delta_1 \mid \dots \mid \Delta_k}\,\mathcal{I} \qquad \frac{\Gamma}{\Gamma'_1 \mid \dots \mid \Gamma'_k}\,\mathcal{M}$$

Insertion rules marked with $\mathcal{I}$ denote that $\Gamma \rightsquigarrow_P \{\Gamma \cup \Delta_1, \dots, \Gamma \cup \Delta_k\}$ if $\{c_1, \dots, c_l\} \subseteq_{AC} \Gamma$. And modification rules marked with $\mathcal{M}$ denote that $\Gamma \rightsquigarrow_P \{\Gamma'_1, \dots, \Gamma'_k\}$. With this notation, we define the most important rules of Tamarin in Fig. 5. The first rule $\mathcal{S}_@$ considers a fact f, which needs to be solved, and makes a case distinction on the rules that can solve it. Solving means adding a rule whose actions contain a fact g where $g \approx f$ can be unified. The second rule $\mathcal{S}_\approx$ solves an equality constraint $s \approx t$ by considering all unifiers σ_i. For each of them, we do a case distinction and apply it to all constraints we have so far, using a modification rule. As the free variables are considered existential, it is easy to solve an existential constraint with $\mathcal{S}_\exists$. For universal quantification, we use the fact that Tamarin restricts itself to guarded trace formulas (see [34] for more information). Intuitively, this means that in negation normal form, each universal quantification has a "guard", which is a negative atomic formula, i.e., either $\neg(f@i)$ or $\neg(s \approx t)$, in a disjunction with the rest of the formula ϕ. If we find a substitution σ such that this guard is wrong, we know that $\phi\sigma$ must hold. This is exactly the way how $\mathcal{S}_{\forall,@}$ and $\mathcal{S}_{\forall,\approx}$ work. In a straightforward way, $\mathcal{S}_\vee$ does a case distinction and $\mathcal{S}_\wedge$ adds both subformulas as individual formulas. Finally, the two rules $\mathcal{S}_{\not\approx}$ and $\mathcal{S}_{\neg @}$ add the unsatisfiable $\perp$ to the constraints if they are not satisfiable. This $\perp$ then leads via $\mathcal{S}_\perp$ to the empty set $\emptyset$, which implies unsatisfiability (given the other case distinctions are unsatisfiable, too). For the sake of brevity, we skip the other rules from [34]. They cover aspects like adding edges, deducing messages and ensuring normality of the resulting dependency graph.

Theorem 1. *The relation $\rightsquigarrow_P$ is sound and complete for any P. This means that if $\Gamma \rightsquigarrow_P \{\Gamma_1, \dots, \Gamma_k\}$, then the set of models (dg, θ) of Γ is the same as the union of the models of all Γ_i.*

With this relation, we search for either $\{\phi\} \rightsquigarrow_P \emptyset$ or $\{\phi\} \rightsquigarrow_P \Delta$ where there is a $\Gamma \in \Delta$ for which we can easily construct a dg that satisfies Γ. In the first case, we know that there cannot be any normal dependency graphs satisfying a $\Gamma \in \emptyset$. Because of soundness and completeness, we know that the same holds for $\{\phi\}$, i.e., there is no normal dependency graph satisfying ϕ. In the second case, however, we can construct a dg satisfying a $\Gamma \in \Delta$ and thus we know that a *trace*(dg) satisfies ϕ. To easily define when this construction of a dependency graph is possible, we define the notion of a *solved* constraint system.

$$\mathcal{S}_{@}:\ \frac{f@i}{i : ru_1, g_1 \approx f \mid \ldots \mid i : ru_l, g_l \approx f}\ \mathcal{I} \qquad \text{if } \{(ru_1, g_1), \ldots, (ru_l, g_l)\} = \{(ru, g) \mid ru \in \lceil P \cup ND \rceil_{\mathcal{RDH}_e, AC} \wedge g \in acts(ru)\}$$

$$\mathcal{S}_{\approx}:\ \frac{s \approx t, \Gamma}{\Gamma\sigma_1 \mid \ldots \mid \Gamma\sigma_l}\ \mathcal{M} \qquad \text{if } \{\sigma_1, \ldots, \sigma_l\} = CSU_{AC}(s = t)$$

$$\mathcal{S}_{\exists}:\ \frac{\exists \vec{x}.\ \phi}{\phi\{\vec{y}/\vec{x}\}}\ \mathcal{I} \qquad \text{if } \vec{y} \text{ freshly chosen variables such that } S(x_i) = S(y_i)$$

$$\mathcal{S}_{\forall,@}:\ \frac{\forall \vec{x}.\ \neg(f@i) \vee \phi}{\phi\sigma}\ \mathcal{I} \qquad \text{if } g@j \in as(\Gamma), \sigma \in CSU_{AC}(g@j, f@i)$$

$$\mathcal{S}_{\forall,\approx}:\ \frac{\forall \vec{x}.\ \neg(s \approx t) \vee \phi}{\phi\sigma}\ \mathcal{I} \qquad \text{if } \sigma \in CSU_{AC}(s = t) \text{ and } dom(\sigma) \subseteq \vec{x}$$

$$\mathcal{S}_{\vee}:\ \frac{\phi_1 \vee \phi_2}{\phi_1 \mid \phi_2}\ \mathcal{I} \qquad \mathcal{S}_{\wedge}:\ \frac{\phi_1 \wedge \phi_2}{\phi_1\ ,\ \phi_2}\ \mathcal{I}$$

$$\mathcal{S}_{\not\approx}:\ \frac{\neg(t \approx t)}{\bot}\ \mathcal{I} \qquad \mathcal{S}_{\bot}:\ \bot, \Gamma\ \mathcal{M} \qquad \text{(no solutions)}$$

$$\mathcal{S}_{\neg @}:\ \frac{\neg(f@i), \Gamma}{\bot}\ \mathcal{I} \qquad \text{if } f@i \in_{AC} as(\Gamma)$$

$$\mathcal{S}_{Prem}:\ \frac{i : ri}{j : ru_1, (j, v_1) \rightarrowtail (i, u) \mid \ldots}\ \mathcal{I} \qquad \text{if } prems(ri)_u \text{ is no } K^{\uparrow/\downarrow}/Fr\text{-fact and } \{(ru_1, v_1), \ldots\} = \{(ru, v) \mid ru \in \lceil P \cup ND \rceil_{\mathcal{RDH}_e, AC} \wedge v \text{ is index in } concs(ru)\} \text{ and } j \text{ is freshly chosen}$$

$$\mathcal{S}_{\rightarrowtail}:\ \frac{i : ri, j : ru, (i, u) \rightarrowtail (j, v)}{concs(ri)_u \approx prems(ru)_v}\ \mathcal{I}$$

Figure 5: Most important rules of Tamarin. For the complete set of rules, see [34].

Definition 6 (Redundancy)**.** *An application of a constraint solving rule is considered* redundant *if it does not add any new constraints except for trivial equalities $s \approx t$ where $s =_{AC} t$ or one of the following holds:*

- *The rule is $\mathcal{S}_{@}$ and $f@i \in_{AC} as(\Gamma)$.*
- *The rule is $\mathcal{S}_{\exists}$ and there are terms $\vec{t}$ such that $\phi\{\vec{t}/\vec{x}\} \in_{AC} \Gamma$*
- *The rule is $\mathcal{S}_{\vee}$ and $\phi_1 \in_{AC} \Gamma$ or $\phi_2 \in_{AC} \Gamma$*

For some further rules in [34], there are individual conditions like the above.

We say that a constraint system is solved if all rule applications are redundant. Additionally to this redundancy, we define some well-formedness conditions, which clearly hold for all constraint systems we encounter during the construction. These well-formedness conditions are:

WF1 All node constraints are AC-instances of rules from $\lceil P \rceil_{\mathcal{RDH}_e, AC}$ or ND except for multiplication rules.
WF2 For all node constraints of the form $i : ps \text{--}[]\!\!\rightarrow K^{\uparrow}(m)$, there is a constraint $i \triangleright K^{\uparrow}(m')$ with $m =_{AC} m'$.
WF3 All provides constraints are of the form $i \triangleright K^{\uparrow}(m)$.
WF4 For all edge constraints $(i, u) \rightarrowtail (j, v)$, there are node constraints $i : ri$ and $j : ru$ such that u and v are valid conclusion and premise indices, respectively.
WF5 For all chain constraints $i \twoheadrightarrow (j, v)$, there are node constraints $i : ri$ and $j : ru$ such that $concs(ri)_1$ and $prems(ru)_v$ are of the form $K^{\downarrow}(t)$ for some t.
WF6 There are no fresh names in Γ.

Theorem 2. *Every non-empty, well-formed constraint system Γ for P that is solved with respect to $\rightsquigarrow_P$ has at least one dependency graph that satisfies it.*

The proof in [34] basically picks the nodes and edges from the constraint system and proves that they fulfill the properties of a normal dependency graph. With this construction, we have a (not necessarily) terminating algorithm to determine satisfiability or unsatisfiability of a formula with respect to a protocol. That concludes this section.

Appendix B. Adding Natural Numbers

In this appendix, we will formally define the natural number of Tamarin and prove correctness. In contrast to the other extensions, we do not only add new rules and constraints but touch the very essentials of the equational theory. Thus, many constraint rewriting rules are changed and we have to prove that the wellformedness conditions still hold. All in all, this is a rather lengthy proof in which nothing exciting happens.

First, we enumerate the changes to the original Tamarin, which we will often refer to as "our changes":

1) Add a new sort $nat \leq_S msg$ with a constant name $N_{nat} = \{1\}$, which is treated public like the sort pub (but still has $nat \cap pub = \emptyset$).
   This means also that the rule Public can now also deduce terms of sort nat. The choice of publicness reflects the purpose of nat. They are supposed to be small counters which are guessable and thus considered public knowledge.
2) Add a function $+ : nat \times nat \rightarrow nat$, which is associative and commutative.
   This function is the addition on nat. It explicitly disallows any other sort than nat in the signature. That means that no non-public knowledge can be in it and thus, no deconstruction rule is needed. This is the important difference to the multiset-encoding of counters as it has been done without our extension.
3) Do not add construction rules for $+$ and 1.
   Not having a construction rule for public values makes sense as they can be directly deduced by the Public rule. Note that all public inputs can be directly deduced, even if there are variables inside.

Actually, these changes are quite specific and we want to capture more general changes to Tamarin such that easily new sorts and sorted functions can be added without the need for new proofs.

4) Allow new sorts with arbitrary hierarchy $\leq_S$.
5) Allow functions to have arbitrary signatures instead of only $msg \times \ldots \times msg \rightarrow msg$ if the output is not atomic $(pub, fresh)$.
   *We also disallow functions to have sort nat as output except for $+$. Ther reason is that functions having potentially non-public inputs, but result in something public are unnatural and cause complications in the attacker deduction, especially in **N4**. The reason is that we use this assumption in the proof for the subterm rules concerning natural numbers.*
6) Allow equational theories to refer to different sorts than msg if they are sort-decreasing.
   The sort-decreasingness is important to prove properties regarding the equational theory. Also it makes sense intuitively: If we reduce a term to another one, we do not expect it to change the sort. Though, it can specialize, e.g., $dec(enc(x : nat)) = x : nat$ changes the sort from msg to nat which is fine as $nat \leq_S msg$.

To justify these changes, we have to do the following:

1) Show how to adapt the equational theory to cope with the changes.
2) Prove that the normal form conditions still hold.
3) Prove that completeness & soundness of the constraint solving rules still hold.
4) Prove that the well-formedness conditions still hold.
5) Prove that we can still construct a normal dependency graph from a solved constraint system.

B.1. Equational Theory

Adapting the equational theory is fairly easy as the underlying tool Maude [37], which is used for variant generation, already supports sorts with arbitrary hierarchies and even function signatures. I.e., it is not hard-wired to support only exactly three sorts $(msg, fresh, pub)$. Thus, the unification process, which changes with this sort hierarchy is easily adapted. The slight problem is that for function signatures, it creates so-called error-supersorts which are applied if the original signature is violated.

Example 6. *Consider a function $f : nat \rightarrow nat$. As the input of the function is a number, it cannot be of sort msg or even fresh. Only terms of sort nat (or subsorts of nat) create a signature-respecting term when inserted into f. If something else is inserted, the signature $f : [msg] \rightarrow [msg]$ with error-supersorts is used in Maude.*

Luckily, we can prove that we never encounter these error-supersorts during variant generation if we restrict ourselves to sort-decreasing rewriting rules.

Example 7. *To understand the trouble involving non-sort-decreasing rewriting rules remember the function $f : nat \rightarrow nat$. Additionally, consider the equation $f(x : nat) =$ "string" $: pub$. If we apply this non-sort-decreasing rewrite rule to the inner term of $f(f(x : nat))$, we get $f($"string" $: pub)$ which clearly violates the signature of f.*

Thus, we have to restrict the rewriting rules R, i.e., the equational theory to be sort-decreasing, i.e., for each rule $l \to r$ it has to hold that $S(r) \leq_S S(l)$. This rules out the equation stated before. Note that this does not rule out a strict $S(r) <_S S(l)$, which makes sense as it cannot violate a functions signature, e.g., inputting a number into a hash function $h : \mathit{msg} \to \mathit{msg}$ is totally fine as the signature also accepts numbers $\mathit{nat} \leq_S \mathit{msg}$ as input.

The same argument applies to substitutions and valuations, which are, fortunately, already sort-decreasing in Maude. For associative and commutative equations, we even need sort-preservingness, which makes sense as they do not have a canonical direction. With these restrictions, we can prove the following new well-formedness condition in Section B.4:

WF7 The terms in all node constraints and formulas are signature-respecting.

B.2. Normal Form Conditions

Here, we will look at all normal form conditions, explain them, and prove that they still hold with our changes. We use the numbering from [34] as far as possible and group them by topic. First, we consider the conditions that are not targeted towards a specific theory. These are the ones that have the potential to be violated by our changes, which become especially present with the public values and in the unification process.

N1 All rules in I are $\downarrow_{\mathcal{RDH}_e}$-normal.
This ensures that only reduced terms are used in the dependency graph. Intuitively, this rule allows to prune constraint systems with terms that could be reduced. $\mathcal{RDH}_e$ is the set of rewriting rules used by [34]. Clearly, our changes do not impact this condition.

N3 If there are two conclusions c and c' with conclusion facts $K^x(m)$ and $K^{x'}(m')$ such that $m =_{AC} m'$ and either $x = x' = \uparrow$ or $x = x' = \downarrow$ then $c = c'$.
[weakened to N3' and N3" in obs. eq.] This rule ensures that no attacker knowledge is constructed twice. Clearly, our changes do not impact this condition.

N4 All conclusion facts $K^\uparrow(f(t_1, \ldots, t_n))$ where f is an invertible function are conclusions of the construction rule for f.
An invertible function is a function that has deconstruction rules to extract its argument terms. These are, for example, pairs or multisets. **N4** *intuitively enforces that the terms of invertible functions are first deconstructed before the rule Coerce is used. Note that $+$ is not invertible. Moreover, for this rule it is important that it does not apply to functions resulting in a public value as they do not have a construction rule. This is a problem which arises with our changes as the public sort nat can be non-atomic.*

N5 If a node i has a conclusion $K^\downarrow(m)$ and a node j has a conclusion $K^\uparrow(m')$ with $m =_{AC} m'$, then $i < j$ and either $\mathit{root}(m)$ is invertible or the node j is an instance of *Coerce* or *Public*.
Together with **N3**, *this rule ensures uniqueness of knowledge, i.e., that it is not deconstructed and then constructed again (except for invertible functions). However, this rule did not take into account public values, which is why we had to add the "or Public" at the end. This change is not specific to our changes and should have been there before. We consider it a mistake in the theory of [34]. Note that this rule is disabled if* TAMARIN *is used in diff-mode to prove observational equivalence properties [12]*

The next two rules are specifically targeted to the Diffie-Hellman constructions. Thus, our changes do not impact these conditions.

N2 There is no multiplication rule that has a premise fact of the form $K^\uparrow(s * t)$ and all conclusion facts of the form $K^\uparrow(s * t)$ or $K^\downarrow(s * t)$ are conclusions of a multiplication rule.
This condition forces the attacker to construct products by multiplying their components. If TAMARIN *is used in diff-mode, this property is weakened by removing the second part (after the and). Our changes do not impact this condition as it is only connected to multiplication rules.*

N6 There is no node $[K^{\downarrow d}(a), K^\uparrow(b)]\!-\![]\!\!\rightarrow K^{\downarrow e}(c^d)$ where c does not contain any fresh names and $\mathit{nifactors}(d) \subseteq_{ACC} \mathit{nifactors}(b)$.
In this rule, we see the attacker knowledge facts $K^{\downarrow d}$ and $K^{\downarrow e}$ which are an even more fine-grained subclasses of $K^\downarrow$ responsible for Diffie-hellman exponentiation [34]. With nifactors(), which extracts the (non-inverse) factors from a multiplication term, we can ensure that no exponentiation rule is used if we can alternatively directly construct the term c^d. Our changes do not impact this condition as it is only connected to exponentiation rules.

The rules **N7** and **N8** cope with the multiset operator and are, thus, independent of our changes. One might think that we need such rules for $+$ as well, because it is associative and commutative as the multiset operator. However, this is not necessary as $+$ neither has construction nor deconstruction rules.

N7 There is no construction rule for $+\!\!+$ that has a premise of the form $K^\uparrow(s +\!\!+ t)$ and all conclusion facts of the form $K^\uparrow(s +\!\!+ t)$ are conclusions of a construction rule for $+\!\!+$.

The operation $\#$ is used for the multiset-extension of Tamarin. **N7** *ensures that the construction rules for multisets are not chained but instead, directly a n-ary $\#$ is constructed. Our changes do not impact this condition as it is only connected to multisets.*

N8 The conclusion of a deconstruction rule for $\#$ is never of the form $K^{\downarrow d}(s \# t)$.
Connected to **N7***, this rule ensures that no multisets are extracted from a multiset. Our changes do not impact this condition as it is only connected to multisets.*

The next three rules are specific to bilinear pairing.

N9 There is no node $[K^{\downarrow d}(a), K^{\uparrow}(b)]\text{-}[]\!\!\rightarrow K^{\downarrow e}([d]c)$ such that c does not contain any fresh names and $nifactors(d) \subseteq_{ACC} nifactors(b)$.
This rule is similar to ***N6***. *Our changes do not impact this condition as it is only connected to bilinear pairings.*

N10 There is no node i labeled with $[K^{\downarrow d}([t_1]p), K^{\downarrow d}([t_2]q)]\text{-}[]\!\!\rightarrow K^{\downarrow d}(\hat{e}(p,q)^c)$ such that there is a node j labeled with $[K^{\downarrow d}(\hat{e}(p,q)^c), K^{\uparrow}(d)]\text{-}[]\!\!\rightarrow K^{\downarrow d}(\hat{e}(p,q)^e)$, an edge $(i,1) \rightarrowtail (j,1)$, $nifactors(ti) \subseteq_{ACC} nifactors(d)$ for $i=1$ or $i=2$, and $\hat{e}(p,q)$ does not contain any fresh names.
Our changes do not impact this condition as it is only connected to bilinear pairings.

N11 There is no node $[K^{\downarrow d}([a]p), K^{\downarrow d}([b]q)]\text{-}[]\!\!\rightarrow K^{\downarrow d}(\hat{e}(p,q)^{a*b})$ such that the send-nodes of the first and second premises are labeled with ru_1 and ru_2 and $fsyms(ru_2) <_{fs} fsyms(ru_1)$ where $<_{fs}$ is a total order on sequences of fact symbols.
Our changes do not impact this condition as it is only connected to bilinear pairings.

In later extensions, the papers [9] and [7] add two more conditions which are targeted to solve problems of non-subterm-convergent theories and exclusive or:

N12 There is no chain of nodes repeatedly instantiating a rule of the form $[K^{\downarrow}(l|_p), K^{\uparrow}(t_1), \ldots, K^{\uparrow}(t_i)]\text{-}[]\!\!\rightarrow K^{\downarrow}(r)$ of length at least equal to the number of subterms of $l|_p$, if $l|_p$ and r are unifiable.
This condition is necessary for adding non-subterm-convergent equational theories. It prevents infinite chains of deconstruction rules for these theories. Our changes do not impact this condition as they do not introduce new deconstruction rules.

N13 There is no chain of repeated instantiations of the deconstruction rules for XOR.
This condition is necessary for adding the equational theory of XOR. It prevents infinite chains of deconstruction rules for this theory. Our changes do not impact this condition as they are not connected to the XOR rule.

B.3. Soundness & Completeness

Here, we will prove that we can maintain completeness and soundness of the constraint solving rules with our changes. Moreover, we will add a constraint solving rule $\mathcal{S}_{\triangleright,pub}$ which was forgotten in the previous papers describing Tamarin and adapt the rules $\mathcal{S}_{\triangleright,K^{\uparrow}}$ and $\mathcal{S}_{\triangleright,inv}$.

As our changes are very non-invasive, only very few rules change while most of them address completely different topics. As a first step, we consider the rules concerning unification and variant generation, as our changes introduce a slight change there. The relevant rules are $\mathcal{S}_{@}$ and $\mathcal{S}_{Prem}$ for variant generation as well as $\mathcal{S}_{\approx}$, $\mathcal{S}_{\forall\approx}$ and $\mathcal{S}_{\forall @}$ for AC-unification and -matching.

First, we consider soundness. For this, we only need to consider the modification-rules, not the insertion-rules as the latter only add constraints and thus never increase the set of models. The only modification-rule relevant for us is $\mathcal{S}_{\approx}$. Here, the $\approx$ constraint is removed and the unification σ is applied to the constraint system. Assuming we have a P-solution (dg, θ) of Γ, then $(dg, \theta \circ \sigma)$ is a P-solution of $\Gamma\sigma$ as in $\Gamma\sigma$ the equality constraint is automatically resolved and σ is also applied to θ. This proof does not change with the special function signature of the AC-operator $+$. This concludes the proof of soundness.

For completeness, we will look at the relevant rules individually, too. For each rule we assume that for $dg = (I, D)$, the tuple (dg, θ) is a P-model of a constraint system Γ. Then, we show that there are l and θ' such that (dg, θ') is a model for one of the resulting Γ_l after applying the rule.

$\mathcal{S}_{@}$: From the precondition $f@i$, we know that $f \in_{AC} trace(dg)[\theta(i)]$. This means that there is a rule $ru \in \lceil P \cup ND \rceil_{\mathcal{RDH}_e,AC}$ which is in dg at position $I_{\theta(i)}$ with σ as a grounding substitution: $I_{\theta(i)} = \sigma(ru)$. And which has an action fact g with $\sigma(g) \in acts(ru)$ with $\sigma(g) =_{AC} f$. This means that the constraints $i : ru$ and $g \approx f$ are fulfilled by $(dg, \theta\sigma)$. These are exactly the constraints from a Γ_l. Therefore, there is a Γ_l such that $(dg, \theta\sigma) \Vdash \Gamma_l$. In this proof, we assume that the variants $\lceil P \cup ND \rceil_{\mathcal{RDH}_e,AC}$ are signature respecting. With our changes, we can assume this because of **WF7**. Moreover, note that all substitutions and valuations are sort-decreasing, i.e., yield only signature-respecting terms. With those two observations, the proof remains correct.

$\mathcal{S}_{Prem}$: From the precondition $i : ri$, we know that $ri\theta =_{AC} I_{\theta(i)}$. From **DG1,DG2**, we know that there is an edge $(k, v) \rightarrowtail (i\theta, u)$ with a rule $k : ru' \in ginsts(\lceil P \cup ND \rceil_{\mathcal{RDH}_e,AC})$ for the premise $prems(ri)_u$. As stated, ru' is a ground instance from a rule $ru \in \lceil P \cup ND \rceil_{\mathcal{RDH}_e,AC}$, i.e., there is a σ with $ru' = \sigma(ru)$. We now construct the substitution $\theta' = \theta\sigma[j \to k]$. Clearly, (dg, θ') fulfills the constraints $j : ru$ and $(j, v) \rightarrowtail (i, u)$ as well as the old Γ. This is exactly one of the Γ_l. As in the last proof, it is important that the variants are signature-respecting terms which is given by **WF7**. Moreover, note that all substitutions and valuations are sort-decreasing, i.e., yield only signature-respecting terms. With those two observations, the proof remains correct.

$\mathcal{S}_{\approx}$: From the precondition $t_1 \approx t_2$, we know that $t_1\theta =_{AC} t_2\theta$. Thus, there is a unifier σ with a sort-decreasing valuation θ', which leaves variables $x \in vars(\Gamma)\backslash dom(\sigma)$ while for $x \in dom(\sigma)$ it holds that $\theta(x) =_{AC} \theta'(\sigma(x))$. And for which holds $(dg, \theta') \models \Gamma\sigma$ which is one of the resulting Γ_l from the rule application. Note that this works because θ and σ are sort-decreasing.

$\mathcal{S}_{\forall @}$: From the precondition $\forall \vec{x}.\ \neg(f@i) \vee \phi$, we know that for all substitutions σ which only range over the quantified variables ($dom(\sigma) \subseteq \vec{x}$) the following holds: $(dg, \theta) \models (f@i)\sigma \Rightarrow \phi\sigma$. As we furthermore know that for $(g@j) \in as(\Gamma)$ it holds that $g@j =_{AC} (f@i)\sigma$, the conclusion $\phi\sigma$ must hold as well, i.e., $(dg, \theta) \models \phi\sigma$. What might be affected in this proof is the AC-matching. However upon closer inspection, the AC-property of $+$ does not change anything, we rely on. Thus, there is no difference in the proof and as σ is already sort-decreasing, only signature-respecting terms can arise.

$\mathcal{S}_{\forall\approx}$: This proof is analogous to the one for $\mathcal{S}_{\forall @}$.

This proves soundness and completeness of the constraint solving rules concerning unification and variant generation. The second big change, we introduce, is that public values (of sort *nat*) can be non-atomic. That implies that we have to change the constraint rewriting rules $\mathcal{S}_{\triangleright,K^\uparrow}$ and $\mathcal{S}_{\triangleright,inv}$. More specifically, we add the condition "and $S(m)$ is not public". This was not necessary before as public values m were atomic and could never satisfy $m = f(t_1, \dots, t_k)$. Note that this implies that nothing changes if Tamarin is used without the features we add.

$$S_{\triangleright,K^\uparrow} : \frac{i \triangleright K^\uparrow(m)}{i : [K^\uparrow(t_1), \dots, K^\uparrow(t_k)]\!-\![]\!\!\rightarrow\! K^\uparrow(m) \quad | \qquad i : K^{\downarrow y}(m)\!-\![]\!\!\rightarrow\! K^\uparrow(m)}\ \mathcal{I} \qquad \begin{array}{l}\text{if } m = f(t_1, \dots, t_k) \\ \text{and } f \text{ is not invertible and } f \neq * \\ \text{and } S(m) \text{ is not public}\end{array}$$

$$\mathcal{S}_{\triangleright,inv} : \frac{i \triangleright K^\uparrow(m)}{i : [K^\uparrow(t_1), \dots, K^\uparrow(t_k)]\!-\![]\!\!\rightarrow\! K^\uparrow(m)}\ \mathcal{I} \qquad \begin{array}{l}\text{if } m = f(t_1, \dots, t_k) \\ \text{and } f \text{ is invertible} \\ \text{and } S(m) \text{ is not public}\end{array}$$

Additionally, we have to add a constraint rewriting rule which should be present in [34]. We call this rule $\mathcal{S}_{\triangleright,pub}$ because it processes public values. The rule is similar to $\mathcal{S}_{\triangleright,fresh}$ and looks as follows:

$$\mathcal{S}_{\triangleright,pub} : \frac{i \triangleright K^\uparrow(m)}{i : \emptyset\!-\![]\!\!\rightarrow\! K^\uparrow(m)}\ \mathcal{I} \qquad \text{if } S(m) \text{ is public}$$

We need this rule to resolve premises of the form $K^\uparrow(m)$ where m is a public value. Clearly, this rule is sound as it is an insertion rule. Moreover, completeness holds as well: Assume, there is a (dg, θ) which is a model for Γ and $\mathcal{S}_{\triangleright,pub}$ is applicable. Then we know from the premise that there is a rule at $I_{\theta(i)}$ and for its first conclusion c holds that $c =_{AC} K^\uparrow(m)$. As there are no construction rules for public values (including *nat*) except for the rule *Public*, it has to be precisely this rule. Fortunately, this fits exactly the pattern $\emptyset\!-\![]\!\!\rightarrow\! K^\uparrow(m)$ which we require. Thus, (dg, θ) also satisfies $\Gamma_1 = \{i : \emptyset\!-\![]\!\!\rightarrow\! K^\uparrow(m)\} \cup \Gamma$ which concludes the proof of completeness.

Adding $\mathcal{S}_{\triangleright,pub}$ fixes the handling of public values which is not documented in the theory part of [34]. It could have been fixed in another way as follows: Not adding $\mathcal{S}_{\triangleright,pub}$ but instead defining a constraint system as already solved even if $i \triangleright K^\uparrow(m : pub)$ constraints are not satisfied. Then, the *Public* rule instances must be added in the process of building the dependency graph. It would be similar to the handling of fresh variables. However, fresh variables have the additional constraint that they must be unique, which justifies doing it separately. For the Public values, it can be included in the constraint rewriting relation which is cleaner.

B.4. Well-Formedness

Having a closer look at the proofs of the well-formedness conditions in [5], we see that they are completely independent of our changes. Given their correctness, we will prove the new well-formedness condition **WF7** discussed earlier in this chapter.

WF7 The terms in all node constraints and formulas are signature-respecting.

Proof. For the node constraints, we know by **WF1** that they are instances of $\lceil P \cup ND \rceil_{\mathcal{RDH}_e,AC}$. Therefore, it suffices to show that an instance of a variant is signature-respecting. We know that the initial term t, which the user inputs, is signature-respecting. From this, the transformation to an instance of a variant is applying a substitution, applying some rewriting rules, and a final substitution application. As the substitutions and rewriting rules are all sort-decreasing, the subterms t that get replaced by t' via the application of one of them have the property $S(t') \leq S(T)$. Thus, no function signature can be violated. Also, the modifying constraint solving rule $\mathcal{S}_{\approx}$ only applies sort-decreasing substitutions which automatically respect function signatures. This proves the first part of **WF7**.

For checking whether formulas are signature-respecting, we consider the rules $\mathcal{S}_{\exists}, \mathcal{S}_{\forall @}, \mathcal{S}_{\forall\approx}, \mathcal{S}_{\rightarrowtail}$ as they are the only rules which add non-trivial formulas to Γ. The proof for the rules $\mathcal{S}_{\exists}, \mathcal{S}_{\forall @}, \mathcal{S}_{\forall\approx}$ follows from the fact that a subterm of a signature-respecting formula is signature-respecting as well. The $\forall$-rules just apply a sort-decreasing substitution to the subformula and the $\exists$-rule replaces variables while preserving the sort. The last rule to consider is $\mathcal{S}_{\rightarrowtail}$ which adds conclusions and premises of rules. From the first part of **WF7**, we know that these rules only contain signature-respecting formulas. Additionally, we know from the premise $(i, u) \rightarrowtail (j, v)$ that $concs(ri)_u =_{AC} prems(ru)_v$, i.e., $S(concs(ri)_u) = S(prems(ru)_v)$ as $=_{AC}$ is sort-preserving. With this, we conclude the proof of **WF7**. □

B.5. Construction of a Dependency Graph

The last part to prove is that we can construct a P-model for every solved constraint system. The only relevant part is where we instantiate variables with constants. Variables of sort *msg* and *fresh* are instantiated with distinct fresh constants and variables of sort *pub* are instantiated with distinct public constants. For *nat*, we instantiate all variables with 1. After that, the proof in [5] has two steps:

1) Defining a set of constraint properties **CS1-6** which are invariant under $\rightsquigarrow_P$. These properties do not interfere with our changes except for **CS6**: *All trace formulas ϕ are guarded trace formulas.* This holds with our changes because we ensure with **WF7** that our terms are well-sorted.
2) Assuming **CS1-6**, a P-model is constructed. This is rather technical and does not involve the equational theory, publicness or anything else which could be affected by our changes.

This concludes the proof that adding natural numbers does not invalidate the previous theory of Tamarin.

Appendix C. Adding Subterms

For adding subterms, we introduce the predicate $\sqsubset$. We give it the semantics

$$(tr, \theta) \models s \sqsubset t \quad \textit{if } s\theta \sqsubset_{R,AC} t\theta$$

and add the following constraint rewriting rules in the notation from [34]

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 neither AC nor a reducible operator

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*
- *if $\mathbin{+\!\!\!+}$ is an ac-operator and neither reducible nor the addition from natural numbers*
- *if t_i don't have $\mathbin{+\!\!\!+}$as top operator (flatness)*

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*
- *if x_i is syntactically in $t_{(i+1)\%(n+1)}$ and not below a reducible operator*

NEG-RECURSE:

$$\frac{\neg t \sqsubset f(t_1, \ldots, t_n)}{(t \neq t_1 \wedge \neg t \sqsubset t_1) \qquad \ldots \qquad (t \neq t_n \wedge \neg t \sqsubset t_n)}\ \mathcal{I}$$

if f is neither AC nor a reducible operator

NEG-AC-RECURSE:

$$\frac{\neg t \sqsubset t_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} t_n}{\forall x.\ t \mathbin{+\!\!+} x \neq t_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} t_n \qquad \neg t \sqsubset t_1 \ \cdots \ \neg t \sqsubset t_n}\ \mathcal{I}$$

- *where x is a new variable*
- *if $\mathbin{+\!\!+}$ is an ac-operator and neither reducible nor the addition from natural numbers*
- *if t_i don't have $\mathbin{+\!\!+}$ as top operator (flatness)*

NEG:

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

NAT:

$$\frac{s \sqsubset t \qquad \Gamma}{\exists x.\ s : nat + x = t \qquad \Gamma[s \mapsto s : nat]}\ \mathcal{M}$$

- *where x is a new variable*
- *if s is a term of sort nat or a var of sort msg*
- *if t has sort nat*

NEG-NAT:

$$\frac{\neg s \sqsubset t}{t \sqsubset s + 1}\ \mathcal{I}$$

if s and t have sort nat

INVALID:

$$\frac{s \sqsubset t}{\bot}\ \mathcal{I}$$

if the sorts of (s, t) are one of

- $(\mathit{fresh}, \mathit{nat})$
- $(\mathit{pub}, \mathit{nat})$
- $(\mathit{any}, \mathit{pub})$ *where any is an arbitrary sort*
- $(\mathit{any}, \mathit{fresh})$ *where any is an arbitrary sort*

UTVPI:

$$\frac{s_0 \sqsubset t_0 \qquad \cdots \qquad s_n \sqsubset t_n}{a_1 = b_1 \qquad \cdots \qquad a_m = b_m}\ \mathcal{I}$$

- *where the equations $a_i = b_i$ are the result of the UTVPI-algorithm for the constraints $s_i \sqsubset t_i$*
- *if s_i and t_i are terms of sort nat*

Lemma 4. *The rules for subterms are sound and complete.*

Proof. Soundness is trivial for all rules except for NAT as they are insertion rules, so we only show completeness for the remaining rules by assuming that for $dg = (I, D)$, the tuple (dg, θ) is a P-Model of a constraint system Γ. Then, we will show that there is a θ' such that (dg, θ') is a model for the resulting Γ' after applying the rule.

RECURSE: We choose $\theta' = \theta$ and assume $t\theta \sqsubset_{R,AC} f(t_1, \ldots, t_n)\theta$ which is $f(t_1\theta, \ldots, t_n\theta)$. It remains to prove that one of $t\theta =_{R,AC} t_1\theta \mid t\theta \sqsubset_{R,AC} t_1\theta \mid \cdots \mid t\theta =_{R,AC} t_n\theta \mid t\theta \sqsubset_{R,AC} t_n\theta$ holds. As f is neither AC nor reducible, this holds because it resembles the definition of $\sqsubset_{synt}$.

AC-RECURSE: We choose $\theta' = \theta$ and assume $t\theta \sqsubset_{R,AC} (t_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} t_n)\theta$ which is $t_1\theta \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} t_n\theta$. It remains to prove that $\exists x.\ t\theta \mathbin{+\!\!+} x =_{R,AC} t_1\theta \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} t_n\theta \mid t\theta \sqsubset_{R,AC} t_1\theta \mid \cdots \mid t\theta \sqsubset_{R,AC} t_n\theta$. We make a case distinction on the validity of the first term. Assuming, the existential equation holds, the disjunction holds because of the first disjunct. Otherwise, we know that $t\theta$ cannot be on the uppermost AC-level of the bigger term and thus has to hold somewhere in lower parts, exploiting the transitive closure of $\sqsubset_{synt}$.

CHAIN: We choose $\theta' = \theta$ and assume that $t_0\theta \sqsubset_{R,AC} x_0\theta$ up to $t_n\theta \sqsubset x_n\theta$ hold and show a contradiction. We will show that the term size of $t_i\theta\downarrow$ is strictly smaller than the one of $t_j\theta\downarrow$ for $j = (i+1)\ \%\ (n+1)$ which is a direct contradiction. As $x_i\theta$ is syntactically in $t_j\theta$ and not below a reducible operator, we have that $x_i\theta\downarrow \sqsubset_{synt} t_j\theta\downarrow$ and thus also $x_i\theta\downarrow \sqsubset_{AC} t_j\theta\downarrow$. And because of the premise constraint, we have $t_i\theta\downarrow \sqsubset_{AC} x_i\theta\downarrow$ which transitively concludes $t_i\theta\downarrow \sqsubset_{AC} t_j\theta\downarrow$. As with AC, the term sizes remain stable, we finally prove that $t_i\theta\downarrow$ is smaller than $t_j\theta\downarrow$.

NEG-RECURSE: This holds analogously to RECURSE. It is the negated form and has thus a conjunction instead of a disjunction in the conclusion of the rule.

NEG-AC-RECURSE: This holds analogously to AC-RECURSE. It is the negated form and has thus a conjunction instead of a disjunction in the conclusion of the rule.

NEG: We choose $\theta' = \theta$ and assume $\neg s\theta \sqsubset_{R,AC} r\theta$ and $t\theta \sqsubset_{R,AC} r\theta$. We show the two conclusions by contradiction.

1) Assume $s\theta \sqsubset_{R,AC} t\theta$. Applying transitivity with the second premise, we get $s\theta \sqsubset_{R,AC} r\theta$ which contradicts the first premise.
2) Assume $s\theta =_{R,AC} t\theta$, then because of the second premise, we have $s\theta \sqsubset_{R,AC} r\theta$ which contradicts the first premise.

NEG-NAT: We choose $\theta' = \theta$ and assume $\neg s\theta \sqsubset_{R,AC} t\theta$. As s and t are of sort nat, we have that $s\theta$ and $t\theta$ are just additions of $1 : nat$ uniquely determined by their respective term sizes l_s and l_t. As we have a equivalence between the less-than comparison of the lengths and $\sqsubset_{R,AC}$, we have $\neg l_s < l_t$. From that follows $l_t \leq l_s$ which is equivalent to $l_t < l_s + 1$ because of discreteness of $\mathbb{N}$. This is equivalent to $t\theta \sqsubset_{R,AC} (s+1)\theta$ which concludes the proof.

NAT: For soundness, the observation suffices that $s\theta$ has to have sort *nat* because t has sort *nat*, so θ can be equally applied to $s : nat$. Thus, $t\theta = t[s \mapsto s : nat]\theta$ for any term t, i.e., the validity of all constraints stay the same.

For completeness, we choose $\theta' = \theta$ and assume $s\theta \sqsubset_{R,AC} t\theta$. Applying the length-construction from the NEG-NAT, we have $l_s < l_t$ and thus, $\exists l_x.\ l_s + l_x = l_t$. We now construct $x = +_{i=1}^{l_x}(1 : nat)$ as l_x ones summed up. This x has the property $s\theta + x\theta = t\theta$ which fulfills the existential conclusion.

INVALID: We choose $\theta' = \theta$ and assume $s\theta \sqsubset_{R,AC} t\theta$. If t has sort *nat*, then $t\theta$ solely consists of $1 : nat$, i.e., $s\theta$ has to have sort *nat* as well. However, *nat* is disjoint from both *fresh* and *pub* which yields a contradiction in this case. A contradiction also follows if t has sorts *pub* or *fresh* as nothing can be a strict subterm of an atomic value.

UTVPI: We choose $\theta' = \theta$. Basically the UTVPI-algorithm guarantees that the equations follow from the subterms because the subterms translate to the less-than relation as s_i and t_i have sort *nat*.

□

Lemma 5. *If the constraint system is solved, there is a dependency graph satisfying all constraints (especially the subterms).*

Proof. We first take the construction of (dg, θ) from the proof without subterms and adapt θ while taking care that we do not invalidate the other constraints. We then look at all subterm constraints $s \sqsubset t$ and filter out the following ones:

1) If t contains reducible operators, we abort the attempt to construct a valid P-model and tell the user that we cannot deal with reducible operators in subterm constraints. We do so as well if we find a reducible operator in a negative subterm constraint.
2) If t is not a variable, we ignore the corresponding subterm as it has been dealt with by the rules [AC-]RECURSE already
3) Now we have two possibilities for the sorts of s and t which are (nat, nat) or (any, msg) because of INVALID. In the first case, the rule NAT already ensured that we fulfill the subterm predicate and we thus filter those ones out.

For the remaining $s \sqsubset t$ we thus have that t is a variable of sort *msg*. Moreover, these constraints form a directed acyclic graph because of CHAIN. Thus, we can satisfy all subterm constraints by the following construction. For each variable t with the subterm constraints $s_1 \sqsubset t, \ldots, s_n \sqsubset t$, we modify θ to substitute $t \mapsto fun(s_1, \ldots, s_n)$ where *fun* is a fresh non-reducible function of arity n not used in the protocol. This is possible because of the acyclicity of the constraints. Now, by construction, all subterms $s_i \sqsubset t$ are satisfied. It remains to prove that this construction did not invalidate other constraints:

- equalities $s = t$ are not affected as θ is applied on both sides.
- inequalities $s \neq t$ cannot be invalidated as we only introduce fresh, unused function symbols and the other variables are mapped to distinct constants
- for equality guarded formulas $\forall x.\ s = t \Rightarrow f$, we have the same as for inequalities
- for negative subterms $\neg s \sqsubset t$, we do the following sort distinction for (s, t):
  - (nat, nat) are ensured correct via NEG-NAT $\rightarrow$ NAT
  - $(nonat, nat)$, (any, pub), and (any, pub) where *nonat* is any sort except for *nat* hold true anyways because of sort incompatibilities and atomicity.
  - for (any, msg) where t is not a variable, we already applied NEG-[AC-]RECURSE as the topmost operator of t is not reducible. With that rule applied, the correctness is assured via the recursive cases.
  - for (any, msg) where t is a variable, we have two cases. *First:* If there is no $s' \sqsubset t$, then t is just instantiated with a fresh constant which makes $\neg s \sqsubset t$ true. *Second:* If there are $s_i \sqsubset t$, then NEG assures that $\neg s \sqsubset s_i$ and $s \neq s_i$ are constraints as well. We can assume that these constraints are satisfied because of induction over the DAG-structure of the $s \sqsubset t$. Therefore, with θ substituting $t \mapsto fun(s_1, \ldots, s_n)$, we know that $\neg s \sqsubset fun(s_1, \ldots, s_n)$.

This concludes the proof. □

Appendix D. Adding Fresh Orderings

Lemma 6. *FRESH-ORDER from Fig. 6 is sound and complete*

Proof. Soundness: This rule only inserts constraints and is thus sound. *Completeness:* We assume that for $dg = (I, D)$, the tuple (dg, θ) is a P-Model of a constraint system Γ. Now, we show that there is a θ' such that (dg, θ') is a model

FRESH-ORDER:

$$\frac{i_0 : f_0 \qquad j : g \qquad t_1 \sqsubset s_1 \; \dots \; t_{n-1} \sqsubset s_{n-1} \qquad i_0 \neq j \; \dots \; i_m \neq j}{i_m < j} \; \mathcal{I}$$

if $\exists u, v, \mathtt{Fact}, s_0, t_n$ *with*

- $\mathit{prems}(f_0)_u = \mathtt{Fr}(s_0)$
- $\mathit{prems}(g)_v = \mathtt{Fact}(t_n)$
- $j : g \notin \mathit{route}(i_0 : f_0, s_0)$
- s_i *is syntactically in* t_{i+1} *and not below a reducible operator*

where $\mathit{route}(i_0 : f_0, s_0)$ *is the maximal list* $[i_0 : f_0, \dots, i_m : f_m]$ *where for two consecutive elements* $i_a : f_a, i_b : f_b$ *holds:*

- f_a *has only one conclusion which is a linear fact (especially not !KU or !KD)*
- $\exists w$ *with* $\mathit{concs}(f_a)_1 = \mathit{prems}(f_b)_w$
- *there is an edge* $(i_a, 1) \mapsto (i_b, w)$

Figure 6: The full fresh order rule with all improvements in the notation style from [34]

for the resulting Γ' after applying the rule. In this case, we choose $\theta' = \theta$. It remains to prove that $\theta(i_m) < \theta(j)$ or $i_0 = j \vee \dots \vee i_m = j$. We know that $t_i\theta$ and $s_i\theta$ are ground instances without variables and reducible operators on the path to the root, i.e., $t_i\theta \sqsubset s_i\theta$ is $t_i\theta \sqsubset_{AC} s_i\theta$ which means that there are $t'_i =_{AC} t_i\theta$ and $s'_i =_{AC} s_i\theta$ such that $t'_i \sqsubset_{synt} s'_i$ holds. Because of the transitivity of $\sqsubset_{synt}$, we get that $s'_0 \sqsubset_{synt} t'_n$ and we moreover know that there are no reducible operators on the path from s'_0 to the root of t'_n. Concluding, we know that $s_0\theta \sqsubset_{synt} t_n\theta$.

We now prove that there is a chain of rules from $g\theta$ to $f_m\theta$, i.e., $\theta(i_m) < \theta(j)$ if $g_\theta : j_\theta$ is not in $[i_0 : f_0, \dots, i_m : f_m]\theta$ (*route*) by induction with the following lemma: For any rule $l\theta : r\theta$ with a premise $\mathit{prems}(r\theta)_v = \mathtt{Fact}(t\theta))$ with $s_0\theta \sqsubset_{synt} t\theta$, either $l\theta : r\theta$ is in *route* or there is a rule $l'\theta : r'\theta \notin [i_0 : f_0, \dots, i_{m-1} : f_{m-1}]\theta$ (*routeDropped*) with an edge $(l'\theta, u) \rightarrowtail (l\theta, v) \in D$ and $\mathit{prems}(r'\theta)_{v'} = \mathtt{Fact}(t'\theta))$ with $s_0\theta \sqsubset_{synt} t'\theta$.

Proof of the lemma: As Γ is solved, we have an edge $(l', u) \rightarrowtail (l\theta, v) \in D$ and a rule $l' : r'\theta$ with $\mathit{prems}(r\theta)_v =_{AC} \mathit{concs}(r'\theta)_u = \mathtt{Fact}(t''\theta)$ and $\theta(l') < \theta(l)$. If $r'\theta$ is the Fresh rule, then ${}_F\mathit{act}(t\theta) = \mathtt{Fr}(s0\theta)$. Because of uniqueness of $\mathtt{Fr}$, $l : r\theta = i_0\theta : f_0\theta$ which is in *routeDropped*. Otherwise, variables in a conclusion have to occur in a premise, i.e., because $s_0\theta \sqsubset_{synt} t''\theta$, there has to be a premise $\mathit{prems}(r'\theta)_{v'} = \mathtt{Fact}(t'\theta)$ with $s_0\theta \sqsubset_{synt} t'\theta$. If $l' : r'\theta$ is not in *routeDropped*, the proof is done. Else, $l : r\theta$ has to be in *route* as there is only one conclusion of a non-persistent fact of $l' : r'\theta$ which has to be connected to $l : r\theta$ in *route*.

Inductively applying this lemma to $j\theta : g\theta$ gives us a path of edges to $i_m\theta : f_m\theta$ if $j\theta : g\theta$ is not in $\mathit{route}(i_0 : f_0, s_0)\theta$. Thus, we have $\theta(i_m) < \theta(j)$ and otherwise $i_0 = j \vee \dots \vee i_m = j$ which concludes the proof.

□

Appendix E. Adding Monotonic Injective Facts

Lemma 7. *A fact F detected as injective is actually injective, i.e., there is no execution of the transition system reaching a multiset S_n containing two instances of F with the same first term.*

Proof. We prove a small helper lemma: For an instance of F with id as first term in S_i, there is a $\mathtt{Fr}(id)$ fact consumed in a previous step. Proof by induction over the trace of S_i.

Induction start: Trivial as $S_0 = \emptyset$

Induction step: If $F(id, \dots)$ is in S_i, then two cases can happen for linear facts. First: The exact same fact was in S_{i-1} as well and we apply the induction hypothesis. Second: A rule was applied with $F(id, \dots)$ as conclusion. Then, we have a) there is a $\mathtt{Fr}(id)$ consumed from S_{i-1} (done) or b) there is a $F(id, \dots)$ in S_{i-1} and we can apply the induction hypothesis.

Now we can prove the full lemma by induction over the trace of S_i.

Induction start: Trivial as $S_0 = \emptyset$

Induction step by contradiction: We assume that there are two instances of $F(id, \dots)$ is in S_i. Then at most one of them (f) can be introduced in S_i. The other one (f') has to be in S_{i-1} as well. The introduction comes from an applied rule with f as a conclusion. Thus, we have either b) there is a $F(id, \dots)$ distinct from f' (as it is consumed) in S_{i-1} and we can apply the induction hypothesis (done) or a) there is a $\texttt{Fr}(id)$ in consumed from S_{i-1}. In the latter case, we apply the helper lemma and get a second consumption of $\texttt{Fr}(id)$ from a set S_j before S_{i-1}. As $\texttt{Fr}(id)$ has been consumed two times, it has to be created two times as well. This is a contradiction because of FRESH which concludes the proof. ☐

$$\text{MONOTONIC:}\quad \frac{F(id,\dots,s,\dots)@i \qquad F(id,\dots,t,\dots)@j \qquad \Gamma}{\textit{constraints below}}\ \mathcal{I}$$

where s and t are at the same position p in the injective fact F.
If p is a constant position:

1) *insert $s = t$*

If p is a strictly increasing position:

2) *if $s = t \in \Gamma$ then insert $i = j$*
3) *if $s \sqsubset t \in \Gamma$ then insert $i < j$*
4) *if $i < j \in \Gamma$ or $j < i \in \Gamma$ then insert $s \neq t$*
5) *if $\neg s \sqsubset t \in \Gamma$ and $s \neq t \in \Gamma$ then insert $j < i$*

If p is an increasing position:

3') *if $s \sqsubset t \in \Gamma$ then insert $i < j$*
5') *if $\neg s \sqsubset t \in \Gamma$ and $s \neq t \in \Gamma$ then insert $j < i$*

For decreasing and strictly decreasing, do the inverse of the increasing cases.

Lemma 8. *The rule MONOTONIC is sound and complete.*

Proof. Soundness: This rule only inserts constraints and is thus sound. *Completeness:* We assume that for $dg = (I, D)$, the tuple (dg, θ) is a P-Model of a constraint system Γ. Now, we show that there is a θ' such that (dg, θ') is a model for the resulting Γ' after applying the rule. For this rule, we choose $\theta' = \theta$ and consider all cases from the rule:

1) As F is injective, there is a first occurrence of $F(id,\dots,x,\dots)@k$. We prove that for all instances $F(id,\dots,y,\dots)@l$ that $x\theta = y\theta$. Induction start: It's the same instance ($k\theta = l\theta$) and thus $x\theta = y\theta$. Induction step: Assume that $x\theta = y\theta$ holds. Then it also holds for the next instance $F(id,\dots,y',\dots)@l'$ because in each rule, y is not changed as it is a constant position. With this lemma, we have $s\theta =_{R,AC} x\theta$ and $t\theta =_{R,AC} x\theta$ which yields $s\theta =_{R,AC} t\theta$.
2) For the following four cases, we prove a small helper lemma, targeting a potential six'th case for the monotonic rule $\texttt{SIX}$: *if $i < j \in \Gamma$ then insert $(s \sqsubset t \vee s = t)$* We do not use this case because it introduces subterms which can introduce case splits and other complications in the model. Nevertheless, here is the proof: Because $i\theta < j\theta$ and injectiveness of F, we have a path of rules in I connected with edges in D from $i\theta$ to $j\theta$. We show for each instance $F(id,\dots,x,\dots)@k$ on that path (excluding i and including j) that $s\theta \sqsubset_{R,AC} x\theta$ holds by induction. Induction start: The rule $F(id,\dots,x,\dots)@k$ directly after i has $s\theta \sqsubset_{R,AC} x\theta$ because there is an edge between $i\theta$ and $k\theta$ in D, i.e., the premise of $k\theta$ and conclusion of $i\theta$ are the same while for the premise and conclusion within k holds that the term in the premise is syntactically in the term from the conclusion and not below a reducible operator. The induction step follows the exact same scheme as the induction start. As j is included in the path, we have $s\theta \sqsubset_{R,AC} t\theta$ from this lemma.
   Now we can prove 2) by contradiction: We assume $i\theta \neq j\theta$, i.e., $i\theta < j\theta$ or $j\theta < i\theta$ because of totality of $<$. From the proof of $\texttt{SIX}$ we get $s\theta \sqsubset_{R,AC} t\theta$ or $t\theta \sqsubset_{R,AC} s\theta$. These two possibilities both exclude $s\theta = t\theta$ which concludes the proof by contradiction.
3) Proof by contradiction: We assume $\neg i\theta < j\theta$, i.e., $i\theta = j\theta$ or $j\theta < i\theta$ because of totality of $<$. In the first case, we directly have $s\theta = t\theta$ which contradicts $s\theta \sqsubset_{R,AC} t\theta$. In the second case, we apply the proof of $\texttt{SIX}$ to yield $t\theta \sqsubset_{R,AC} s\theta$ which also contradicts $s\theta \sqsubset_{R,AC} t\theta$.
4) This is exactly the second part of the proof for 2)
5) Proof by contradiction: We assume $\neg j\theta < i\theta$, i.e., $i\theta = j\theta$ or $i\theta < j\theta$ because of totality of $<$. In the first case, we directly have $s\theta = t\theta$ which contradicts $s\theta \neq t\theta$. In the second case, we apply the proof of $\texttt{SIX}$ to yield $s\theta \sqsubset_{R,AC} t\theta$ which contradicts $\neg s\theta \sqsubset_{R,AC} t\theta$.

The cases *3')* and *5')* are proven analogous to *3)* and *5)* as well as the decreasing cases.

☐