ETH zürich Research Collection

Master Thesis

Adding support for user-defined sorts and sorted function symbols to Tamarin

Author(s):
Staub, Cedric

Publication Date:
2013

Permanent Link:
https://doi.org/10.3929/ethz-a-009959880

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.

ETH Library

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master thesis

Adding support for user-defined sorts and sorted function symbols to Tamarin

Cedric Staub
cstaub@ethz.ch

Supervisor: Dr. Ralf Sasse
Professor: Prof. Dr. David Basin
Issue date: March 1, 2013
Submission date: September 1, 2013

Abstract

A security protocol is an exchange of messages between multiple parties with the intent of achieving certain security properties, such as confidentiality of message contents or authentication. Security protocol verification tools are used to formally verify the properties these protocols claim. These tools are often limited by the type of protocols they can model appropriately. Recently, a new tool called Tamarin was developed which allows for the modeling of security protocols that make use of Diffie-Hellman exponentiation, as well as allowing for advanced security properties, such as compromising adversaries.

We extend the theory behind Tamarin to allow for the modeling of protocols which could not be properly modeled before. Our changes include support for user-defined sorts and function symbols, built-in natural numbers, and iterated function application. We argue how our changes fit with the original theory behind Tamarin. We implement our modifications and show examples of protocols modeled with the new capabilities. We present an improvement in the modeling of existing protocols in the Tamarin security protocol verification tool. We exemplify part of our improvements by a protocol working with a simple encrypted counter value, called the `Counter` protocol. We also show a protocol which could not be properly modeled in Tamarin before: we present a protocol based on a minimal hash chain based on our iterated function application extension.

Contents

1 Introduction **7**

1.1 Protocol verification 7

1.2 Tamarin prover 7

1.3 Contributions 8

2 Preliminaries **9**

2.1 Notational preliminaries 9

2.2 Security protocol model 10

3 User sorts and functions **13**

3.1 Motivation 13

3.2 Theory 15

3.3 Implementation 16

3.4 Discussion 17

4 Natural numbers **21**

4.1 Theory 21

4.2 Implementation 21

5 Iterated functions **23**

5.1 Motivation 23

5.2 Theory 23

5.3 Implementation 25

5.4 Case study 26

6 Related work **29**

7 Conclusions **31**

A Implementation details **35**

A.1 User-defined sorts and function symbols 35

A.2 Built-in natural numbers 36

A.3 Iterated function application 37

1 Introduction

We start with a short overview of topics relevant to this thesis. We will give a short introduction into protocol verification, look at the verification tool this thesis is based on, and list the relevant contributions.

1.1 Protocol verification

A security protocol is an exchange of messages between multiple parties, with the intent of achieving certain security properties. These include, among others, confidentiality, integrity, authentication, availability and key establishment. Traditionally, these types of protocols have been defined only as a specification of the message exchange between multiple parties participating in the protocol. Protocol designers would then argue which security properties the message exchange fulfills. However, these arguments were often informal in their nature, and turned out to be wrong in many cases. Thus, many protocols had flaws that were claimed not to exist. Because of this, the need for formal verification tools for security protocols arose.

These tools can be a great help in formally verifying the correctness of protocols, but they are often limited by the type of protocols and the security properties they can model appropriately. Recently, David Basin, Benedikt Schmidt, Simon Meier and Cas Cremers developed a security protocol verification tool called Tamarin prover [16, 25, 20]. Tamarin prover allows for the modeling of protocols that make use of Diffie-Hellman exponentiation with advanced security properties, such as compromising adversaries. This thesis describes how we extend both the theory and implementation of the Tamarin security protocol verification tool, to enable us to model and verify new types of protocols which could not be properly modeled using Tamarin before.

1.2 Tamarin prover

Tamarin prover is a security protocol verification tool developed by the Institute of Information Security at ETH Zurich in Switzerland [16]. Tamarin supports both falsification (including attack-finding) and verification of security protocols, which are modeled as multiset rewriting systems [25, 20]. The message theory in Tamarin allows for the modeling of protocols which make use of Diffie-Hellman exponentiation, in combination with a user-defined subterm-convergent rewriting theory [24, 18]. Tamarin has an interactive user interface which allows for dynamic proof construction [26]. It uses Maude [9, 4, 5] as a unification backend [16].

The execution of a protocol in Tamarin is modeled as a labeled transition system (see [24], section 3.1). The protocol, as well as the capabilities of the adversary, are

jointly specified as a multiset rewriting system on the state of the protocol execution. Security properties are expressed as trace properties on the resulting transition system. This models the interaction between the protocol and the adversary as updates to the shared protocol execution state. The protocols are verified in a symbolic setting, reasoning about the structure of messages and the information the adversary can extract from the network.

1.3 Contributions

We extend the theoretical framework behind Tamarin prover to introduce user-defined sorts and sorted function symbols. We explain how our modifications are consistent with the original framework. We extend the implementation to make user-defined sorts and sorted function symbols usable, and present a practical example that benefits from our changes. Furthermore, we extend Tamarin to make it possible to mark user-defined functions as associative and commutative. We then present a model for natural numbers based on these improvements.

Additionally, we extend the theory and implementation to provide built-in natural numbers supporting addition and a neutral element (zero). We enhance Tamarin to provide a nice, built-in syntax for these natural numbers and show how protocol models can be improved by using the new functionality.

Finally, based on our addition of built-in natural numbers, we extend Tamarin to support iterated function application. This allows for the modeling of protocols which make use of iterated function application, such as protocols based on hash chains (for example TESLA [21]). We show a simplified, minimal hash chain example and how it can be modeled and verified using the new functionality, but do not consider TESLA.

2 Preliminaries

The notation used in this thesis is largely the same as the notation used in [1], as well as [24] and [18]. All relevant definitions have been reproduced below.

2.1 Notational preliminaries

Order-sorted term algebras. An order-sorted signature is a three-tuple $\mathbf{\Sigma} = (S, \leq, \Sigma)$ consisting of a set of sorts S, a partial order $\leq$, and a set Σ of function symbols. Every function symbol in Σ is associated with sorts such that two properties hold: **(1)** for every $s \in S$, the connected component C of s in $(S, \leq)$ has a greatest element (the *top-sort*), denoted by $top(s)$ and **(2)** for every $f : s_1 \times \cdots \times s_k \to s$, $f \in \Sigma$ with $k \geq 1$, there is an $f : top(s_1) \times \cdots \times top(s_k) \to s$ in Σ referred to as the *top-sort overloading* of f. In some cases, $\mathbf{\Sigma}$ can be substituted by Σ if the sort hierarchy is clear from the context.

We assume that there exists a countably infinite set of variables $\mathcal{V}_s$ for every sort $s \in S$ and define $\mathcal{V} = \biguplus_{s \in S} \mathcal{V}_s$. A single variable $x \in \mathcal{V}_s$ is denoted by the short-hand $x : s$. We denote the set of well-sorted terms of sort s constructed over $\Sigma \cup \mathcal{V}'$ for a signature $\mathbf{\Sigma}$ (where $\mathcal{V}' \subseteq \mathcal{V}$) as $\mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V}')_s$. The set of *all* well-sorted terms is denoted as $\mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V}')$.

Positions, subterms and variables. For a term $t \in \mathcal{T}_{\Sigma}(\mathcal{V}')$ we define the notion of a *position* in the term as a sequence of natural numbers. For some term t and a position p inside that term, we denote the *subterm* of t at position p as $t|_p$. Let $[]$ denote the empty sequence, $[i]$ the singleton sequence with element i and let $p_1 \cdot p_2$ be the concatenation of two sequences p_1 and p_2. Then, we define $t|_p$ formally by the equations **(1)** $t|_{[]} = t$, **(2)** $f(t_1, \cdots, t_k)|_{[i] \cdot p'} = t_i|_{p'}$ (with $i \leq k$) and **(3)** undefined otherwise.

A position p is valid for a term t if $t|_p$ is not undefined. The term $valid\,(t, p)$ is true iff the position p is valid for the given term t. The set of *subterms of* t, $St\,(t)$, is defined as $\{t|_p \mid valid(t, p)\}$. Additionally, we use $root\,(t)$ to denote $root\,(f(t_1, \cdots, t_k)) = f$ for all $f \in \Sigma$, and $root\,(t) = t$ otherwise. The substitution of a *subterm* at a given position p in a term t by term s is denoted as $t[s]_p$. We define $vars\,(t) = St\,(t) \cap \mathcal{V}$, and we call a term t *ground* if $vars\,(t) = \emptyset$.

Substitutions. We define a substitution as a well-sorted function $\sigma : \mathcal{V} \to \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$, with the constraint that σ must correspond to the identity function, except on a finite set of variables denoted by $dom(\sigma) \subseteq \mathcal{V}$. The image of $dom\,(\sigma)$ under the function σ is denoted as $range\,(\sigma)$. We can extend σ to an endomorphism on $\mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$ and simply write $t\sigma$ to denote the application of σ to the term t. The substitution σ with $dom\,(\sigma) = \{x_1, \cdots, x_k\}$ and $x_i\sigma = t_i$ $(i \leq k)$ is denoted by $\{t_1/x_1, \cdots, t_k/x_k\}$. The identity substitution is denoted by id.

Equational theories and unification. We define an equational theory to be a tuple

$$\frac{l_1 \;\cdots\; l_k}{r_1 \;\cdots\; r_k}\,[a_1 \;\cdots\; a_k]$$

Table 1: A *multiset rewriting rule* in inference notation.

$\mathcal{E} = (\mathbf{\Sigma}, E)$ with an order-sorted signature $\mathbf{\Sigma}$ and a set of equations E. An equation is simply an unordered pair of terms $t \simeq t'$ with $t, t' \in \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$. Given $\mathcal{E}$, we define the *equational theory* on $\mathcal{E}$ (denoted by $=_{\mathcal{E}}$) to be the smallest possible $\mathbf{\Sigma}$-congruence with all instances of equations in E. We call an equation $t \simeq t'$ *regular* if $vars(t) = vars(t')$ and likewise an equation is *sort-preserving* if $t\sigma \in \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})_s$ if and only if $t'\sigma \in \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})_s$, for all substitutions σ. The notions of *regular* and *sort-preserving* equations can be extended to equational theories, so an equational theory is *regular* (or *sort-preserving*) if all equations in E are *regular* (or *sort-preserving*). An $\mathcal{E}$-unifier is a substitution σ for two terms t and t' such that $t\sigma =_{\mathcal{E}} t'\sigma$.

2.2 Security protocol model

Message terms. Tamarin uses an order-sorted term algebra to model message terms (see section 2.1). We have the order-sorted signature $\mathbf{\Sigma} = (S, \leq, \Sigma)$ with three sorts $\{\texttt{Msg}, \texttt{Fr}, \texttt{Pub}\} \subseteq S$. The sort hierarchy is given by the relations $\texttt{Fr} \leq \texttt{Msg}$ and $\texttt{Pub} \leq \texttt{Msg}$. We have `Msg` as a fixed top-sort, meaning that $top(\texttt{Fr}) = \texttt{Msg}$ and $top(\texttt{Pub}) = \texttt{Msg}$.

Tamarin also has various built-in functions for building message terms, modeling symmetric and asymmetric encryption, hashing, multisets, bilinear pairing, and more. These can be enabled using the `builtins` keyword.

Facts and transitions. The state of the transition system is modeled as finite multisets of *facts* from a set of *facts* denoted by $\mathcal{F}$. This set of facts is built from an unsorted signature $\Sigma_{\mathcal{F}}$, such that $F(t_1, \cdots, t_k) \in \mathcal{F}$ for all $F \in \Sigma^k_{\mathcal{F}}$ and $t_i \in \mathcal{T}_{\mathbf{\Sigma}}(\mathcal{V})$, $1 \leq i \leq k$. We use multiset rewriting rules to express possible transitions on the system. A rule is defined as a three-tuple (l, a, r) where $l, a, r \in \mathcal{F}^*$. We denote such a rule either by $l \,{-}[a]{\rightarrow}\, r$ or using inference notation (see [24], section 3.1.2). A rule in inference notation would be rendered as shown in Table 1. Note that the square brackets can be left out if a is empty. In practice, we tend to use distinct sets of facts for l, r when compared to a in order to avoid potential confusion of the protocol specifier and reader.

Protocol rules. Multiset rewriting rules are used to model the protocol. In order to model the protocol interaction with the network, there are two special *facts* used in protocol rules. The first is the `Out` fact, which is used to denote terms that the protocol has sent to the network. The second is the `In` fact, which is used to denote terms that the adversary has sent and can be received by the network. For instance, the rule

`In` (x) $-[]\rightarrow$ `Out` (`h(x)`) would receive an arbitrary message term x from the network and then send out $h(x)$.

Tamarin uses the Dolev-Yao [10] model to prove properties of protocols. In the Dolev-Yao model, the adversary controls the network. This means that in Tamarin, all `Out` facts are received by the attacker and all `In` facts are sent by the attacker. So the adversary can control messages sent over the network, intercept them and construct new ones. The adversary is only constrained by the use of cryptographic methods in the protocol. Unlike in the real world, the adversary is unable to modify messages by manipulating the bit representations of messages (for example, flipping bits of an encrypted message). However, the adversary can symbolically manipulate messages and make use of algebraic properties of the underlying cryptographic system.

Additionally, a special fact `Fr` is used to generate *fresh names*, which are guaranteed to be unique. A protocol may also define an arbitrary number of protocol-specific facts. There are certain restrictions as to what constitutes a valid protocol rule which are not reproduced here, but can be found in [24], under definition 3.4 ("Protocol rules").

Intruder rules. The multiset rewriting rules are used to model adversary capabilities and message deduction. There are multiple special facts used to model message deduction and adversary capabilities. The fact `K` is used to denote adversary knowledge of a message term. To model message deduction, we use the facts $\mathtt{K}^{\Uparrow}$ (K-up, rendered as `KU` in Tamarin) and $\mathtt{K}^{\Downarrow d}$ (K-down, rendered as `KD` in Tamarin). The $\mathtt{K}^{\Uparrow}$ fact is used to model message construction (from smaller terms) and the $\mathtt{K}^{\Downarrow d}$ fact is used to model message destruction (to yield smaller terms). This distinction is necessary to enforce normal message deductions (see [24], section 3.2.3). There is an additional $\mathtt{K}^{\Downarrow e}$ fact for messages that have been deduced by modifying an exponent of a message, but it is not relevant for this thesis.

3 User sorts and functions

In this section, we are extending Tamarin to add support for user-defined sorts and sorted function symbols. We will show a motivating example, look at some theoretical background, and then talk about our implementation.

3.1 Motivation

```
theory Counter begin

builtins: multiset, symmetric-encryption

rule Create:
    [ Fr(~s) ]
  --[ Start(~s) ]->
    [ Counter(~s, '1') ]

rule Inc:
    [ Counter(~s, x) ]
  --[ Counter(~s, x) ]->
    [ Counter(~s, x ++ '1'), Out(senc(x, ~s)) ]

lemma Lesser_Enc_Secret[use_induction]:
  "All x y #i #j s.
      Counter(s, x) @ i & K(senc(y, s)) @ j
      ==> (#i < #j | Ex z. y ++ z = x)"

end
```

Table 2: A simple example protocol based on multisets.

We begin with an example in order to illustrate why user-defined sorts and sorted functions are useful. Consider a very simple protocol which implements a counter (see Table 2) based on Tamarin's existing support for unsorted multisets. We have a `Create` rule for initializing the counter and an `Inc` rule to increment the counter. Additionally, every time we increment the counter we send out the encryption of the current value of the counter x using a counter-specific key s. Now, we would like to prove certain properties of the protocol.

The main property we want to prove is expressed in the lemma `Lesser_Enc_Secret`: We want to show that if the attacker knows the encryption of the counter value y, and the counter is currently at state x, then y must be smaller than x. This formalizes the notion that the attacker can not produce encryptions for values the counter has not yet reached.

However, if we try to prove the properties of this procotol using Tamarin, we run into

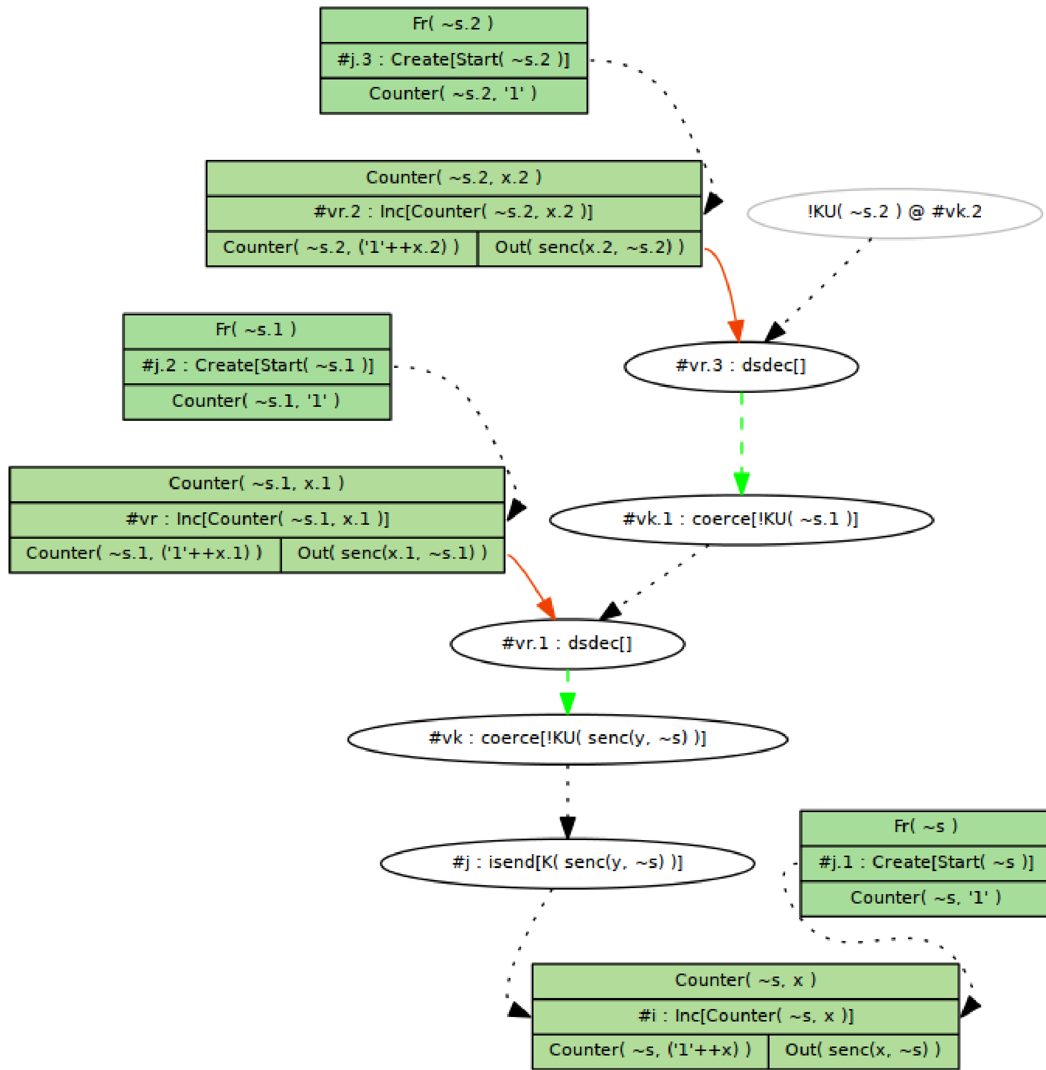

Figure 1: Problematic trace when working with the `Counter` theory.

problems. If we look at Figure 1, we can see the problematic trace: the prover ends up using the output of the `Inc` rule as an oracle for arbitrary values. As we can see in the trace, the output of the rule `Inc` at node `#vr.2` is decrypted. The decrypted counter value `x.2` is then used as a key to decrypt the output of rule `Inc` at node `#vr`. This is not directly incorrect, because it's not impossible that the encrypted value could match the encryption key for the other counter from the point of view of the prover. If we have two terms `x:Msg` and `s:Fr` we can unify them with the unifier $\{\mathtt{x} \rightarrow \mathtt{z}, \mathtt{s} \rightarrow \mathtt{z}\}$ for some `z:Fr`, since $\mathtt{Fr} \leq \mathtt{Msg}$. We could make this example work, if it was possible to restrict `x` to a subsort of `Msg` to prevent unification with a `Fr` value.

Our goal is to solve this using the built-in sorting system of Tamarin, by allowing the user to define custom sorts and functions on those sorts. This would allow us to build a better counter and make the example `Counter` theory work.

3.2 Theory

We recall the sort hierarchy used in Tamarin (see section 2.2), with the three sorts $\{\mathtt{Msg}, \mathtt{Fr}, \mathtt{Pub}\} \subseteq S$ and the relations $\mathtt{Fr} \leq \mathtt{Msg}$ and $\mathtt{Pub} \leq \mathtt{Msg}$. Remember that `Msg` is the fixed top-sort, meaning that $top\left(\mathtt{Fr}\right) = \mathtt{Msg}$ and $top\left(\mathtt{Pub}\right) = \mathtt{Msg}$. So we have a sort for fresh values (`Fr`), a sort for public values (`Pub`), and a fixed top-sort (`Msg`).

In order to be able to add new user-defined sorts, we need to decide how to modify the sort hierarchy used in Tamarin. We would like to make the changes as non-intrusive as possible to make integration into the existing theory easier, and to make sure existing theory files don't have to be changes unless absolutely necessary. That means we would like to keep at least the fixed top-sort, so we need to make the user-defined sorts subsorts of `Msg`. Another question that poses itself is how to treat the new user-defined sorts. To solve the dilemma of the motivating example of the previous section, we would need a user-defined sort that would allow us to model a counter. Something which is essentially a public value, but distinct from other sorts.

Taking those constraints into account, we propose the splitting of the sort `Pub` into multiple disjunct subsets $\mathtt{Pub}_0$ (the original sort `Pub`) and $\mathtt{Pub}_1, \cdots, \mathtt{Pub}_k$ (one per user-defined sort). We treat all user-defined sorts essentially as public values, except with the restriction that unification is not possible between them. This means that the terms $\mathtt{x} : \mathtt{Pub}_0$ and $\mathtt{y} : \mathtt{Pub}_1$ can not be unified. This way we can restrict unwanted unification. Other than this, the relationships between the sorts remains intact. So we still have the fixed top-sort, meaning that for all new sorts we still have $\mathtt{Pub}_k \leq \mathtt{Msg}$ and $top\left(\mathtt{Pub}_k\right) = \mathtt{Msg}$.

We argue that the split of public sorts leaves the original theory behind Tamarin intact. For all intents and purposes except unification the new public sorts are treated equally. Any arguments and constructs on the original hierarchy can be applied to the new hierarchy by simply letting $\mathtt{Pub} = \cup_{i=0}^{k} \mathtt{Pub}_i$. We only add new restrictions, mean-

ing that any unification that is possible in our model is also possible in the original model. That is, the set of possible unifications in our model is a subset of the possible unifications in the original model.

In the original theory, the split into fresh values and public values is made to allow for the modeling of random values such as secret keys. These are re-generated on every run of the protocol, and in general are not known by the adversary. Fresh values are guaranteed to be unique. For fresh values, unification means that the value must have been known from elsewhere (that is, must be the same value) - they could not have been constructed by the adversary. Although the adversary can generate new fresh values, by the uniqueness property they will be different. This property is preserved by our changes. Furthermore, our changes preserve the flat sort hierarchy the original correctness proof depends on - we do not add new subsorts to either `Fr` or `Pub`. As the original correctness proof is not concerned with unification between different public values, we are free to partition the set of public values into multiple subsets without affecting other parts of the theory behind Tamarin.

In more detail, the original proof makes the assumptions that **1)** the sort hierarchy is flat, **2)** we have three sorts `Msg`, `Fr` and `Pub`, **3)** the sorts `Fr` and `Pub` are incomparable, and **4)** the sort `Msg` is the top-sort of all other sorts. With our changes, these assumptions are preserved. The hierarchy stays flat because all new sorts are parallel to the sort public. We can still consider the theory to have three sorts by the argument that $\texttt{Pub} = \cup_{i=0}^{k} \texttt{Pub}_i$. The sorts `Fr` still remains incomparable with any sort $\texttt{Pub}_i$. And finally, the sort `Msg` is still the top-sort.

3.3 Implementation

In order to add the new functionality to Tamarin, we need to make several changes. First, we need to give the user a syntax for actually defining their own user-defined sorts and sorted function symbols. Second, we need to augment the internal data structures in Tamarin to be able to deal with custom sorts. And third, we need to tell Maude [4, 5] (which is used by Tamarin as a unification oracle) about the new custom sorts.

To let the user define their own sorts, we add a new keyword to the domain-specific language for Tamarin. The new keyword is simply `usersorts`, and allows the user to list one or more sort names. Furthermore, we extend the `functions` section of a theory file to make it possible to give function signatures to user-defined functions. The syntax is similar to the way operators are declared in Maude (see [4], section 3.4). The extended `Counter` protocol with user-defined sorts using the proposed syntax is shown in Table 3.

Internally, the implementation was adapted to keep track of the new user-defined sorts when handling message terms and handle them properly when parsing and

serializing terms. Since Tamarin also uses Maude to perform certain operations, the code for doing so also had to be adapted to take into account the new sort hierarchy. More details can be found in the Appendix A.1.

```
theory Counter_Sorted begin

builtins: symmetric-encryption

usersorts: NAT                // Define custom sort

functions: zero: -> NAT,      // Zero constant
           succ: NAT -> NAT   // Successor function

rule Create:
    [ Fr(~s) ]
  --[ Start(~s) ]->
    [ Counter(~s, zero) ]

// Increment counter, with sort restrictions on x.
rule Inc:
    [ Counter(~s, x:NAT) ]
  --[ Counter(~s, x:NAT) ]->
    [ Counter(~s, succ(x)), Out(senc(x:NAT, ~s)) ]

end
```

Table 3: The Counter_Sorted protocol with user-sorts to restrict x.

3.4 Discussion

The changes allow us to model a counter more nicely than before. To fix the original Counter theory, we simply defined a new user-defined sort NAT with two functions zero and succ. Thus, we model the counter using Peano numbers. However, the attentive reader may have noticed that the Counter_Sorted theory does not implement addition for NAT values, making it impossible to express our original security property from the Counter theory in Table 2.

This stems from a limitation in Tamarin which postulates that the user-defined rewrite theory must be subterm convergent. This is to ensure that the rewrite theory has the finite-variant property, which in turn guarantees that efficient unification is possible [15]. Unfortunately, addition on Peano numerals does not have the finite-variant property [14], and an alternative model is therefore required.

As that alternative, we propose to represent natural numbers by defining a constant function *one* and an associative/commutative function *plus*. The canonical representation of the number three would then be $plus(one, one, one)$ for example. This does not

allow us to have a neutral element (zero) in our model, but it is good enough for the purposes of modeling a counter. More so, this model has the finite-variant property and can be used in Tamarin.

```
theory Counter_Sorted_AC
begin

builtins: symmetric-encryption

usersorts: NAT

functions: one: -> NAT,                // Constant 'one'
           plus: NAT NAT -> NAT [AC]   // Addition function

rule Create:
    [ Fr(~s) ]
  --[ Start(~s) ]->
    [ Counter(~s, one) ]

// Increment counter, with sort restrictions on x.
rule Inc:
    [ Counter(~s, x:NAT) ]
  --[ Counter(~s, x:NAT) ]->
    [ Counter(~s, plus(x:NAT, one)), Out(senc(x:NAT, ~s)) ]

// New lemma with user-defined addition function.
lemma lesser_senc_secret[use_induction]
  "All x:NAT y:NAT #i #j s.
      Counter(s, x:NAT) @ i & K(senc(y:NAT, s)) @ j
      ==> (#i < #j | Ex z:NAT. plus(y:NAT, z:NAT) = x:NAT)"

end
```

Table 4: The `Counter_Sorted_AC` protocol with a user-defined AC function *plus*.

Tamarin can already deal with associative/commutative functions, and uses Maude for delayed AC-unification. This functionality is not exposed to the user at the moment, but with some modifications we change this and allow the user to mark functions as being associative and commutative. A function can only be marked as being both associative and commutative, but not commutative or associative individually. Tamarin uses Maude to deal with delayed unification of AC terms [16], so all we need to do is to tell Maude which user-defined functions are associative/commutative. To avoid certain problems arising from having both functions that are associative/commutative and rewrite rules involving those functions in Maude (see [4], section 2.9.5, “Traps for the Unwary”), we do not allow a function to be marked as AC and having rewrite rules defined for it at the same time.

We can easily extend Tamarin to make it possible to annotate user-defined functions as `AC`, with some small internal changes to keep track of which functions are `AC` and

which ones are not. For the user, we simply extend the syntax of our sorted function symbols by allowing a function to be annotated with `[AC]` (see Table 4).

With all these additions, we can now model a counter which uses user-defined sorts to restrict unification, as well as having a simple model of natural numbers using a user-defined AC function. The result is the `Counter_Sorted_AC` theory shown in Table 4. Further improvements to this are possible by improving the built-in support for natural numbers in Tamarin, as shown in section 4.

4 Natural numbers

In order to make it possible to use natural numbers with a neutral element, we want to directly implement natural numbers as a new built-in for Tamarin. This also makes using natural numbers more convenient, and allows us to define a specialized syntax for natural numbers.

4.1 Theory

To model natural numbers, we introduce a new sort into the sort system in Tamarin called `Nat` with $\texttt{Nat} \leq \texttt{Msg}$ and $top\,(\texttt{Nat}) = \texttt{Msg}$, the same as a user-defined sort.

$$\frac{}{\texttt{K}^{\Uparrow}(0{:}nat)} \qquad \frac{}{\texttt{K}^{\Uparrow}(1{:}nat)} \qquad \frac{\texttt{K}^{\Uparrow}(x{:}nat) \quad \texttt{K}^{\Uparrow}(y{:}nat)}{\texttt{K}^{\Uparrow}(x{+}y)}$$

Table 5: Intruder rules for the natural numbers extension.

We list the proposed intruder rules for the natural numbers extension in Table 5. This includes three basic rules, one rule each for the constants zero and one, as well as a construction rule for addition. The canonical representation of any natural number $n > 1$ is simply the sum of n-times the constant one. So, the canonical representation of the number three would be $1 + 1 + 1$. Zero is the neutral element and can be discarded when normalizing a term.

Note that, for addition both x and y are sort-restricted to the sort Nat. This constraint guarantees that we can only construct valid natural numbers. Otherwise, it would be possible to (for example) construct the addition of a fresh value and a public name, which would be meaningless. Furthermore, the constants for zero and one are of sort `Nat`.

4.2 Implementation

In order to implement the model we have chosen for natural numbers into Tamarin, there are several parts of the code we need to extend. First, we need to implement a new built-in sort for natural numbers and extend the code for parsing and pretty-printing to deal with the new sort much like we did for user-defined sorts. Second, we need to extend the Maude module Tamarin generates in order to tell Maude about our natural numbers model. And third, we need to implement the intruder rules and make sure our normal form conditions hold when applying them.

The new sort we add to Tamarin for natural numbers is named `Nat`, and we extend the `LSort` data type used for representing sorts by a new constructor `LSortNat`. The new constructor does not have any parameters. Functions for parsing and pretty-printing theories were extended to deal with the new built-in natural numbers. A more detailed explanation of our changes can be found in the Appendix A.2.

```
theory Counter_Builtin_Nat
begin

builtins: symmetric-encryption, natural-numbers

rule Create:
    [ Fr(~s) ]
  --[ Start(~s) ]->
    [ Counter(~s, 0:nat)]

rule Inc:
    [ Counter(~s, x:nat) ]
  --[ Counter(~s, x:nat) ]->
    [ Counter(~s, (x:nat + 1)), Out(senc(x:nat,  ~s)) ]

lemma lesser_senc_secret[use_induction]:
  "All x:nat y:nat #i #j s.
      Counter(s,x:nat) @ i & K(senc(y:nat,s)) @ j
      ==> (#i < #j | Ex z:nat. (y:nat + z:nat) = x:nat)"

end
```

Table 6: The `Counter_Builtin_Nat` protocol using built-in natural numbers.

One small adaption that needs to be made concerns the new operator for addition of natural numbers. Currently, Tamarin is already using the + symbol for representing union on multisets. In order to avoid confusion for new users, we instead change the + operator to be addition over natural numbers and implement a new, distinct operator for the multiset union. We propose the new operator ++ as a replacement for the multiset union.

With these changes, we can now take our original `Counter` theory from the previous section and re-write it to use natural numbers. The result is shown in Table 6. As we can see, the addition of built-in natural numbers makes the model more readable when compared to `Counter_User_AC` (see Table 4). We can now also (once again) initialize the counter with zero, since Tamarin (and Maude) now know about the neutral element.

5 Iterated functions

In this section we will use our new built-in natural number support in Tamarin to enable us to model iterated function application. We will look at a motivating example, explain the theory behind our iterated functions model, and show a new protocol which can now be verified as a result of our changes.

5.1 Motivation

We would like to model protocols based on hash chains in Tamarin, as well as have the ability to reason about such hash chains. This has previously not been possible, as illustrated by the `Minimal_Hash_Chain` theory listed in Table 7, which is motivated by TESLA (we will talk in more detail about the hash chain in section 5.4).

Although it was possible to model the generation and checking of the hash chain itself, it was not possible to prove the security of such a model. In the old model, repeated function application was simply represented by repeatedly applying a function f, yielding terms such as $f(f(f(m)))$. This works, but makes it impossible to express a lemma which holds for an arbitrary number of applications of f. We can only express lemmas on a constructible term of fixed size, for example $f(f(m))$, but not on a class of terms. If we wanted to express a lemma over an arbitrary number of applications of the function f, we would require an infinite number of lemmas (one lemma for every number n of applications of f). This makes it impossible to prove certain properties of chained function applications, since we can never make a statement about the full chain, but only about a single member.

5.2 Theory

To allow the modeling of iterated functions, we propose to make it possible to mark a function with the signature `Nat Msg` $\rightarrow$ `Msg` as an iterated function, whereby the first argument counts the number of function applications and the second argument represents the actual argument. For example, applying the function f on a message term m three times yields $f\,(1+1+1, m)$.

We list the proposed intruder rules for the iterated function application extension in Table 8. We have three rules for each iterated function f. We have two construction rules. First, a rule for initially applying a function f to a plain value m any number of times. Second, a rule for taking an instance of f applied on m and further iterating the hash any number of times. And last, we have a simple destruction rule which models the degenerate case of applying the iterated function f zero times on a value. Again, we need a new normal form condition:

```
theory Minimal_HashChain begin

functions: f/1

// Beginning of hash chain generation.
rule Gen_Start:
    [ Fr(seed) ]
  --[ ChainKey(seed) ]->
    [ Gen(f(seed)), Out(seed) ]

rule Gen_Step:
    [ Gen(k) ]
  --[ ChainKey(k) ]->
    [ Gen(f(k)) ]

// At some point the sender decides to stop the hash-chain
// precomputation. We take note of the final key kZero.
rule Gen_Stop:
    [ Gen(kZero) ]
  --[ ChainKey(kZero) ]->
    [ !Final(kZero) ]

// Start checking an arbitrary key. Use a loop-id to allow connecting
// different statements about the same loop.
rule Check_Start:
    [ In(kOrig), Fr(loopId) ]
  --[ Start(loopId, kOrig) ]->
    [ Loop(loopId, f(kOrig), kOrig) ]

rule Check_Step:
    [ Loop(loopId, k, kOrig) ]
  --[ Loop(loopId, k, kOrig) ]->
    [ Loop(loopId, f(k), kOrig) ]

// If we construct kZero starting from kOrig, we have successfully
// checked that kOrig is part of the hash chain.
rule Check_Stop:
    [ Loop(loopId, kZero, kOrig), !Final(kZero) ]
  --[ Success(loopId, kOrig) ]->
    []
```

Table 7: The minimal hash chain theory.

$$\frac{\mathtt{K}^{\Uparrow}(x{:}nat)\quad \mathtt{K}^{\Uparrow}(m)}{\mathtt{K}^{\Uparrow}(f(x,m))} \qquad \frac{\mathtt{K}^{\Uparrow}(x{:}nat)\quad \mathtt{K}^{\Uparrow}(f(y,m))}{\mathtt{K}^{\Uparrow}(f(x+y,m))} \qquad \frac{\mathtt{K}^{\Downarrow d}(f(0{:}nat,m))}{\mathtt{K}^{\Downarrow d}(m)}$$

Table 8: Intruder rules for the iterated functions extension (per iterated f).

- **N1**. There is no node $\mathtt{K}^{\Uparrow}(x : nat),\ \mathtt{K}^{\Uparrow}(m) -[]\rightarrow \mathtt{K}^{\Uparrow}(f(x, m))$ such that premise m has the function symbol f as its root.
- **N2**. There is no node $\mathtt{K}^{\Uparrow}(x : nat),\ \mathtt{K}^{\Uparrow}(m) -[]\rightarrow \mathtt{K}^{\Uparrow}(f(x, m))$ such that premise $x : nat$ reduces to zero.

The conditions are necessary to make sure we only construct valid terms containing the iterated function symbol f. In normal form, the first argument must always be of sort `Nat`, and all terms must be of the normal form $f(x : nat,\ m)$ rather than $f(x : nat,\ f(y : nat,\ m))$. Without the normal form condition **N1**, it would be possible to apply the first rule listed in Table 8 to construct a term of the latter form. Furthermore we do not want the intruder to be able to construct terms of the form $f(0, m)$, which is why we require the normal form condition **N2**.

However, since we need rules in a protocol model to always maintain the normal form of a term, we have to be careful when dealing with iterated function terms now. In particular, the prover will not instantiate any terms of the form $f(x, f(y, m))$. This means that if we have a rule of the form $In(x : nat), In(m) -[]\rightarrow f(x : nat, m)$ it will only be instantiated for message terms m that do *not* have f at their root. If we want the rule to be applicable to those terms, we need to create a duplicate rule of the form $In(x : nat), In(f(y : nat, m)) -[]\rightarrow f((x : nat + y : nat), m)$. This is slightly tedious, but it guarantees that terms always stay in their normal form.

With the new representation, the number of applications is counted using the new built-in natural numbers. So, we can now express a lemma on a term such as $f(x, m)$ for an arbitrary x : `Nat` and therefore an arbitrary number of applications of f.

5.3 Implementation

In order to implement iterated function application in Tamarin, we will have to change the way Tamarin instantiates intruder rules for user-defined functions. Currently, for every public user-defined function, Tamarin will instantiate a standard intruder rule for function application. For example, if we define a function h for hashing, Tamarin will instantiate an intruder rule $x -[]\rightarrow h(x)$ for constructing a single application of that function. However, for user-defined functions marked as iterated, we will have to change this to instantiate our new intruder rules for iterated functions instead.

Internally, we can represent iterated function symbols like regular user-defined functions. We simply add an additional field to the respective data type that allows us to keep track of which functions are iterated, and which ones are not. Additionally, we need to adapt the code in Tamarin which determines if a term is in normal form to make sure we exclude terms of the form $f(x, f(y, m))$. A more detailed explanation can be found in the Appendix A.3.

5.4 Case study

As a case study, we present a minimal hash chain example inspired by TESLA, shown in Table 7. In our minimal hash chain model, we would like to generate a chain of keys based on repeatedly hashing an initial seed. For this, we have the `Gen_Start`, `Gen_Step` and `Gen_Stop` rules. The start rule creates a fresh seed, marks it as a chain key (with the `Chain_Key` annotation) and hashes it once. The step rule takes a key, marks it as a chain key, and hashes it one additional time. The stop rule takes a key, marks it as a chain key, and declares it as the final key. Using the start/step/stop rules, we can create an arbitrarily long chain of keys.

To verify if a key is part of the chain, we have the `Check_Start0`, `Check_Start1`, `Check_Step` and `Check_Stop` rules. The check rule receives an arbitrary value, and starts a loop for verifying it. On every iteration of the loop, we hash the value once. At some point, the value should match the final key. If this is the case, the original value we received must have been part of the original chain. This is ultimately what we want to prove. Now, in our first hash chain model, we are unable to prove the correctness of this model. The prover will get stuck in an infinite loop, trying to track down the end of the chain. However, since the chain can be arbitrarily large, this won't work.

With our changes enabling the use of iterated functions, we can modify the model, as shown in Table 9. Here, we have adapted the rules in the model to use our new iterated function support. Note that the rules `Gen_Step` and `Check_Step` pattern-match on the iterated function application. We can do this without loss of generality, since the `Gen_Start` and `Check_Start0`/`Check_Start1` rules always emit a value that has at least one application of the iterated function.

In our modified model, we can now prove a lemma such as the `Success_chain` lemma shown in Table 9. We can add a lemma called `Chain_key_end` thanks to our changes, which allows us to show the existence of the end of the hash chain. This allows a proof for the `Success_chain` lemma to be generated, and prevents the prover from endlessly looping while trying to track down the end of the chain.

```
theory Minimal_HashChain_Iterated begin

builtins: natural-numbers
functions: f: Nat Msg -> Msg [iterated]

// Beginning of hash chain generation.
rule Gen_Start:
    [ Fr(seed) ]
  --[ ChainKey(seed) ]->
    [ Gen(f(1:nat, seed)), Out(seed) ]

rule Gen_Step:
    [ Gen(f(x:nat, seed)) ]
  --[ ChainKey(f(x:nat, seed)) ]->
    [ Gen(f((x:nat + 1), seed)) ]

rule Gen_Stop:
    [ Gen(kZero) ]
  --[ ChainKey(kZero) ]->
    [ !Final(kZero) ]

// Split of Check_Start rule into multiple cases.
rule Check_Start0:
    [ In(f(x:nat, k)), Fr(loopId) ]
  --[ Start(loopId, f(x:nat, k)) ]->
    [ Loop(loopId, f((x:nat + 1), k), f(x:nat, k)) ]

rule Check_Start1:
    [ In(kOrig), Fr(loopId) ]
  --[ Start(loopId, kOrig) ]->
    [ Loop(loopId, f(1:nat, kOrig), kOrig) ]

rule Check_Step:
    [ Loop(loopId, f(x:nat, kOrig), kOrig) ]
  --[ Loop(loopId, f(x:nat, kOrig), kOrig) ]->
    [ Loop(loopId, f((x:nat + 1), kOrig), kOrig) ]

rule Check_Stop:
    [ Loop(loopId, kZero, kOrig), !Final(kZero) ]
  --[ Success(loopId, kOrig) ]->
    []

// Lemma: If f(x:nat, k) is a key, k must also be a key.
lemma Chain_key_end [use_induction, reuse]:
  "All x:nat k #i . ChainKey(f(x:nat, k)) @ i ==> Ex #j . ChainKey(k) @ j"

// Lemma: A successful check implies that the key was a chain key.
lemma Success_chain:
  "All lid k #i. Success(lid, k) @ i ==> Ex #j. ChainKey(k) @ j"
```

Table 9: The minimal hash chain theory, with iterated functions.

6 Related work

ProVerif is a security protocol verification tool developed by Bruno Blanchet, Vincent Cheval, Xavier Allamigeon and Ben Smyth [3, 2]. It verifies protocols in a symbolic Dolev-Yao model. Protocols are modeled using Horn clauses, and automatically verified to have certain properties such as secrecy and authentication. It is not possible to define arbitrary security properties. The tool can handle an unbounded number of sessions and messages in the protocol. It also supports attack reconstruction, and will try to show how an attack is possible if a certain property can not be verified. ProVerif has support for many different cryptographic primitives, such as symmetric and asymmetric cryptography, digital signatures and hash functions, as well as Diffie-Hellman key agreement [17]. There is a separate graphical user interface for ProVerif called the ProVerif Editor developed by Joeri de Ruiter [22].

Maude-NPA is a security protocol verification tool that can take into account algebraic properties of cryptographic systems. It is developed by Santiago Escobar, Catherine Meadows and José Meseguer [13, 12]. The tool operates in a symbolic setting. It can deal with many advanced algebraic properties such as Abelian groups, exponentiation, and has been extended to deal with homomorphic encryption in [11]. For verification, the analyzer starts from a final state and uses backwards search through narrowing to determine if the state is reachable. It allows for the modeling of theories with commutative and associative properties. There is also a graphical user interface for Maude-NPA, the Maude-NPA GUI, developed by Sonia Santiago, Carolyn L. Talcott, Santiago Escobar, Catherine Meadows and José Meseguer [23].

Scyther is a security protocol verification tool developed by Cas Cremers [6, 7, 8], which can automatically verify various security properties. Protocols are modeled using linear role scripts. It can handle an unbounded number of sessions and nonces, and has the ability to characterize protocols, producing a finite representation of the possible behaviours of a protocol. This means that Scyther can yield a finite representation of all possible protocol behaviours. The tool can automatically find attacks and visualize them. The algorithm underlying Scyther was extended to *Scyther-Proof* by Simon Meier, Cas Cremers and David Basin, which can generate machine-checkable proofs [19]. Scyther has a built-in graphical user interface.

7 Conclusions

We presented various protocols which presented difficulties when modeling them in the Tamarin security protocol verification tool. We examined the problems arising from these protocols, and came up with extensions to the underlying theory of Tamarin to allow for the proper modeling of these protocols. The changes were implemented in Tamarin, and we showed the adapted protocol models and how they were improved.

The first protocol we looked at was the `Counter` protocol, where a counter is repeatedly incremented and it's encrypted value sent out to the network. In the original model we were unable to prove a desired secrecy property, because the prover ended up using the encrypted value as an oracle. We added user-defined sorts and sorted function symbols to Tamarin, which allowed us to add sort restrictions to the model. Through this, we were able to restrict the prover and exclude the incorrect trace from being considered.

Then, we extended Tamarin with support for user-defined associative/commutative function symbols. Through this, users are now able to model more complex structures in Tamarin. We extended the `Counter` theory from before to use natural numbers based on user-defined associative/commutative functions. Through both these changes, we were now able to prove the original secrecy property. The incorrect trace can now be excluded using sort restrictions, and natural numbers can now be properly modeled using user-defined associative/commutative functions.

Furthermore, we decided to take our model for natural numbers and directly built support for it into Tamarin. This makes the use of natural numbers and theories requiring them more natural and easier to read. Again, we showed our original `Counter` theory with the improvements based on our changes.

And finally, we used the new built-in natural number support in Tamarin to model iterated functions. This allows user-defined functions to be marked as iterated, allowing them to be applied repeatedly, and having Tamarin keep track of the repeated function applications. This way, repeated applications of a single function can be counted, allowing us to express security properties on an arbitrary chain of repeated function applications. Thanks to these changes, we were able to model and prove the security properties of a minimal hash chain example inspired by TESLA, which was previously not possible in Tamarin.

References

[1] Marc Bezem, Jan Willem Klop, Roel de Vrijer, and eds. *Term Rewriting Systems*. Cambridge University Press, Apr. 2003. ISBN: 0-521-39115-6. URL: www.cs.vu.nl/~terese/.

[2] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. Cape Breton, Nova Scotia, Canada: IEEE Computer Society, June 2001, pp. 82–96.

[3] Bruno Blanchet, Vincent Cheval, Xavier Allamigeon, and Ben Smyth. *ProVerif: Cryptographic protocol verifier in the formal model*. Aug. 2013. URL: http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.6)*. Jan. 2011. URL: http://maude.cs.illinois.edu/maude2-manual/maude-manual.pdf.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-71940-3.

[6] Cas Cremers. *Scyther tool*. Aug. 2013. URL: http://people.inf.ethz.ch/cremersc/scyther/index.html.

[7] Cas Cremers and Sjouke Mauw. *Operational Semantics and Verification of Security Protocols*. Information Security and Cryptography. Springer, 2012. ISBN: 978-3-540-78636-8.

[8] C.J.F. Cremers. "Scyther - Semantics and Verification of Security Protocols". Ph.D. dissertation. Eindhoven University of Technology, 2006. ISBN: 978-90-386-0804-4.

[9] University of Illinois at Urbana-Champaign Dept. of Comp. Sci. *The Maude System*. Aug. 2013. URL: http://maude.cs.illinois.edu/.

[10] Danny Dolev and Andrew Chi-Chih Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650.

[11] Santiago Escobar, Deepak Kapur, Christopher Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, and Ralf Sasse. "Protocol analysis in Maude-NPA using unification modulo homomorphic encryption". In: *PPDP*. 2011, pp. 65–76. DOI: 10.1145/2003476.2003488.

[12] Santiago Escobar, Catherine Meadows, and José Meseguer. "A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties". In: *Theor. Comput. Sci.* 367.1-2 (2006), pp. 162–202.

[13] Santiago Escobar, Catherine Meadows, and José Meseguer. *Maude-NPA*. Aug. 2013. URL: http://maude.cs.illinois.edu/tools/Maude-NPA/.

[14] Santiago Escobar, José Meseguer, and Ralf Sasse. "Effectively Checking the Finite Variant Property". In: *Proceedings of the 19th international conference on Rewriting Techniques and Applications*. RTA '08. Hagenburg, Austria: Springer-Verlag, 2008, pp. 79–93. ISBN: 978-3-540-70588-8. DOI: 10.1007/978-3-540-70590-1_6.

[15] Santiago Escobar, Ralf Sasse, and José Meseguer. "Folding variant narrowing and optimal variant termination". In: *J. Log. Algebr. Program.* 81.7-8 (2012), pp. 898–928. DOI: 10.1016/j.jlap.2012.01.002.

[16] Institute of Information Security at ETH Zurich. *Tamarin Prover*. July 2013. URL: http://www.infsec.ethz.ch/research/software/tamarin.

[17] Ralf Küsters and Tomasz Truderung. "Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation". In: *CSF*. 2009, pp. 157–171. DOI: 10.1109/CSF.2009.17.

[18] Simon Meier. "Advancing automated security protocol verification". PhD thesis. Switzerland: ETH Zurich, 2013. DOI: 10.3929/ethz-a-009790675.

[19] Simon Meier, Cas J. F. Cremers, and David A. Basin. "Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs". In: *CSF*. 2010, pp. 231–245. DOI: 10.1109/CSF.2010.23.

[20] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *CAV*. 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48.

[21] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. "The TESLA Broadcast Authentication Protocol". In: *RSA CryptoBytes* 5.2 (2002), pp. 2–13. DOI: 10.1.1.19.5113.

[22] Joeri de Ruiter. *ProVerif Editor*. Aug. 2013. URL: http://www.cs.ru.nl/~joeri/proverif_editor.html.

[23] S. Santiago, C. Talcott, S. Escobar, C. Meadows, and J. Meseguer. "A Graphical User Interface for Maude-NPA". In: *Electronic Notes in Theoretical Computer Science* 258.1 (2009), pp. 3–20. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.12.002.

[24] Benedikt Schmidt. "Formal analysis of key exchange protocols and physical protocols". PhD thesis. Switzerland: ETH Zurich, 2013. DOI: 10.3929/ethz-a-009898924.

[25] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *2012 IEEE 25th Computer Security Foundations Symposium* (2012), pp. 78–94. ISSN: 1063-6900. DOI: 10.1109/CSF.2012.25.

[26] Cedric Staub. "A user interface for interactive security protocol design". Bachelor's thesis. Switzerland: ETH Zurich, Aug. 2011. DOI: 10.3929/ethz-a-007554462.

A Implementation details

A.1 User-defined sorts and function symbols

Tamarin internally represents sorts as an algebraic data structure `LSort` which can be found in the `Term.LTerm` module. The data structure has the constructors `LSortMsg` for the `Msg` sort, `LSortFresh` for the `Fr` sort, `LSortPub` for the `Pub` sort and `LSortNode` which is used only internally.

```
data LSort = LSortPub           -- ^ Arbitrary public names.
           | LSortFresh         -- ^ Arbitrary fresh names.
           | LSortMsg           -- ^ Arbitrary messages.
           | LSortNode          -- ^ For internal use.
           | LSortUser String   -- ^ Arbitrary user-defined sort.
           deriving( Eq, Ord, Show, Typeable, Data )
```

Table 10: Adapted representation of sorts in Tamarin for user sorts.

```
data ACSym = Union                  -- ^ Multiset union.
           | Mult                   -- ^ Multiplication.
           | UserAC String String   -- ^ User-defined AC symbols.
```

Table 11: Adapted representation of AC symbols in Tamarin for user sorts.

```
sortOfLTerm :: Show c => (c -> LSort) -> LTerm c -> LSort
sortOfLTerm sortOfConst t = case viewTerm2 t of
  Lit2 (Con c)                                -> sortOfConst c
  Lit2 (Var lv)                               -> lvarSort lv
  FAppNoEq (NoEqSym _ _ _ (Just sts) _) _ -> sortFromString (last sts)
  FUserAC _ sort _                            -> sortFromString sort
  _                                           -> LSortMsg
```

Table 12: Adaptation of the `sortOfLTerm` function for user sorts.

We can extend the data structure with a new parametrized constructor `LSortUser`, which takes a `String` as an argument (see Table 10 for the Haskell code). This allows us to represent the new user-defined sorts with an identifier for each sort. Besides this, various functions that operate on sorts had to be adapted to take the new user-defined sorts into account. Maude, the unification backend used by Tamarin, already deals with unification of different sorts correctly. All we need to do is to provide it with the proper sort annotations.

In order to represent the new sorted function symbols, we need to adapt the data

type that represents user-defined functions in Tamarin. The data type used for this is `NoEqSym`, defined in `Term.Term.FunctionSymbols`. We can easily extend the data type to store a list of sorts for each argument (and return value) of a user-defined function. However, we now need to make sure that these sort restrictions are considered during unification. Tamarin already takes care of sorts when unifying, but doesn't know about our new restrictions yet. What we need to do is to adapt the `sortOfLTerm` function in the `Term.LTerm` module, which is responsible for determining the sort of a term. Previously, Tamarin treated all function applications as being of sort `Msg`. So, we simply add a new case distinction for the newly added sorted function symbols (see Table 12).

Internally, we represent user-defined AC functions by extending the `ACSym` data type in the `Term.Term.FunctionSymbols` module by adding a new constructor `UserAC`. The constructor has two parameters, both of type `String`. The first parameter is for the function symbol (for example `plus`) and the second parameter is for storing the sort the symbol was defined on (such as `NAT`). Because of the nature of AC symbols, such a function can only be defined to operate on a single sort. Once again, we need to adapt the `sortOfLTerm` function to take the new user-defined AC functions into account.

A.2 Built-in natural numbers

In order to support built-in natural numbers, we need to extend Tamarin in a similar fashion as we did for user-defined AC symbols. We extend the `LSort` data type to add a new sort for natural numbers, `LSortNat`. We also extend the `ACSym` data type again, and adapt the `sortOfLTerm` function to give us the correct sort for a natural number term. The changes are similar to our previous extensions, except that we have special cases the for built-in natural numbers extension now.

```
data LSort = LSortPub           -- ^ Arbitrary public names.
           | LSortFresh         -- ^ Arbitrary fresh names.
           | LSortMsg           -- ^ Arbitrary messages.
           | LSortNode          -- ^ For internal use.
           | LSortUser String   -- ^ Arbitrary user-defined sort.
           | LSortNat           -- ^ Built-in natural numbers.
           deriving( Eq, Ord, Show, Typeable, Data )
```

Table 13: Adapted representation of sorts in Tamarin for naturals.

Additionally, we need to implement the instantiation of intruder rules for natural numbers. This is necessary to allow the adversary to construct natural numbers directly. We extend the intruder rule code generation module in Tamarin with the code shown in Table 16, which is called whenever the `natural-numbers` built-in is enabled. The code generates intruder rules that allow the adversary to construct the values zero,

```
data ACSym = Union                    -- ^ Multiset union.
           | Mult                     -- ^ Multiplication.
           | NatPlus                  -- ^ Addition on natural numbers.
           | UserAC String String     -- ^ User-defined AC symbols.
```

Table 14: Adapted representation of AC symbols in Tamarin for naturals.

```
sortOfLTerm :: Show c => (c -> LSort) -> LTerm c -> LSort
sortOfLTerm sortOfConst t = case viewTerm2 t of
  Lit2 (Con c)                           -> sortOfConst c
  Lit2 (Var lv)                          -> lvarSort lv
  FAppNoEq (NoEqSym _ _ _ (Just sts) _) _ -> sortFromString (last sts)
  FUserAC _ sort _                       -> sortFromString sort
  FNatPlus _                             -> LSortNat
  _                                      -> LSortMsg
```

Table 15: Adaptation of the `sortOfLTerm` function for naturals.

one, and the addition of two natural numbers.

For Maude, we model the natural numbers by adding a new sort `TamNat` and the functions `tzero`, `tone` and `tplus`. The `tplus` function is marked as `comm assoc`, with `id:tzero`, giving us a commutative, associative function with a neutral (identity) element (zero). When talking to Maude, Tamarin represents all instances of terms on the new sort `Nat` as their equivalent in Maude. With this, Maude now knows about our changes and can be used for unification for these values.

And finally, we had to make small adaptations to the `unifyRaw` function, which can be found in the `Term.Unification` module of Tamarin. The `unifyRaw` function is responsible for performing the first steps of unification before deferring the rest of the computation to Maude. Here, we had to add new case distinctions to make sure that Tamarin deals with the new zero/one constants for built-in natural numbers correctly and tries to unify them with natural number terms.

A.3 Iterated function application

For our iterated function application extension, we again modify the `NoEqSym` data type defined in `Term.Term.FunctionSymbols`. More specifically, we add a new field which keeps track of which function symbols are supposed to be iterated and which ones are not. For iterated function symbols, we instantiate different intruder rules, but otherwise treat them the same way. Iterated functions are still non-equational

```
natIntruderRules :: [IntrRuleAC]
natIntruderRules =
    [ mkCPlusRule x_var y_var
    , kuRule (ConstrRule natZeroSymString) [] (fAppNoEq natZeroSym [])
    , kuRule (ConstrRule natOneSymString) [] (fAppNoEq natOneSym [])
    ]
  where
    x_var = varTerm (LVar "x" LSortNat 0)
    y_var = varTerm (LVar "y" LSortNat 0)
    kuRule name prems t = Rule name prems [kuFact t] [kuFact t]

mkCPlusRule :: LNTerm -> LNTerm -> IntrRuleAC
mkCPlusRule x_var y_var =
    Rule (ConstrRule natPlusSymString)
         [kuFact x_var, kuFact y_var]
         [kuFact $ fAppAC NatPlus [x_var, y_var]] []
```

Table 16: Code for instantiating intruder rules for natural numbers.

symbols[1], and we do not need to make any special changes to the Maude module generated by Tamarin.

We introduce a new function for instantiating the proper intruder rules for an iterated function. The functions for doing so are reproduced in Table 17. The main function is `iterIntruderRules`, which takes a non-equational symbol and returns a list of intruder rules. Then we have three functions, one per intruder rule. The function `mkC0IterRule` instantiates the intruder rule for first applying an iterated function. The second function, `mkC1IterRule` instantiates the intruder rule for re-applying an iterated function (for example, hashing an already hashed value). And the last function, `mkDIterRule` instantiates the intruder rule for the degenerate case (an iterated function applied zero times on a value).

Furthermore, we need to add certain restrictions to prevent our normal form conditions for iterated functions from being violated. The `nfViaHaskell` function in the `Term.Rewriting.Norm` module can be used to enforce normal forms for terms, which we can use to prevent the construction of terms such as $f(x, f(y, m))$. Additionally, we would like to prevent the intruder from constructing a term such as $f(0, m)$. This is done by adding a new normal form check in the form of the `hasInvalidIter` function shown in Table 18. This is part of the `Theory.Constraint.Solver.Contradiction` module. If an invalid construction is found, we consider it a contradiction because of non-normal terms.

[1] Function symbol that is neither commutative nor associative.

```
-- | Instantiate intruder rules for an iterated function.
iterIntruderRules :: NoEqSym -> [IntrRuleAC]
iterIntruderRules funsym =
    [ mkC0IterRule funsym x_var m_var
    , mkC1IterRule funsym x_var y_var m_var
    , mkDIterRule funsym m_var
    ]
  where
    -- Helper variables
    x_var = varTerm (LVar "x" LSortNat 0)
    y_var = varTerm (LVar "y" LSortNat 0)
    m_var = varTerm (LVar "m" LSortMsg 0)

-- Instantiate construction rule
mkC0IterRule :: NoEqSym -> LNTerm -> LNTerm -> IntrRuleAC
mkC0IterRule sym@(NoEqSym fun _ _ _ _) x_var m_var =
    Rule (ConstrRule $ B.concat [fun, BC.pack "0"])
         [kuFact x_var, kuFact m_var]
         [kuFact $ fAppNoEq sym [x_var, m_var]]
         [kuFact $ fAppNoEq sym [x_var, m_var]]

-- Instantiate iteration rule
mkC1IterRule :: NoEqSym -> LNTerm -> LNTerm -> LNTerm -> IntrRuleAC
mkC1IterRule sym@(NoEqSym fun _ _ _ _) x_var y_var m_var =
    Rule (ConstrRule $ B.concat [fun, BC.pack "1"])
         [kuFact x_var, kuFact $ fAppNoEq sym [y_var, m_var] ]
         [kuFact $ fAppNoEq sym [fAppAC NatPlus [x_var, y_var], m_var]]
         [kuFact $ fAppNoEq sym [fAppAC NatPlus [x_var, y_var], m_var]]

-- Instantiate destruction rule
mkDIterRule :: NoEqSym -> LNTerm -> IntrRuleAC
mkDIterRule sym@(NoEqSym fun _ _ _ _) m_var =
    Rule (DestrRule fun)
         [kdFact $ fAppNoEq sym [fAppNoEq natZeroSym [], m_var]]
         [kdFact m_var] []
```

Table 17: Code for instantiating intruder rules for iterated functions.

```
hasInvalidIter :: System -> Bool
hasInvalidIter sys =
    any isInvalidIter $ M.elems $ L.get sNodes sys
  where
    -- Check for invalid instantiation of rule
    isInvalidIter ru = fromMaybe False $ do
      -- Only applies to intruder rules
      guard $ isIntruderRule ru
      -- Verify premise and conclusion
      [p1, _] <- return $ L.get rPrems ru
      [conc]  <- return $ L.get rConcs ru
      -- Check number of applications, output
      (UpK, count) <- kFactView p1
      (UpK, out)   <- kFactView conc
      -- Don't allow zero as number of applications
      return (isZero count && isIter out)

    -- Check for iterated function
    isIter conc =
      case viewTerm2 conc of
        FAppNoEq (NoEqSym _ _ _ _ True) _ -> True
        _                                 -> False

    -- Check for invalid number of applications
    isZero count =
      case viewTerm2 count of
        FAppNoEq fs [] | fs == natZeroSym -> True
        FNatPlus [t1, t2]                 -> isZero t1 && isZero t2
        _                                 -> False
```

Table 18: Code for restricting intruder rules for iterated function application.

List of Figures

1 Problematic trace when working with the `Counter` theory. 14

List of Tables

1 A *multiset rewriting rule* in inference notation. 10

2 A simple example protocol based on multisets. 13

3 The `Counter_Sorted` protocol with user-sorts to restrict x. 17

4 The `Counter_Sorted_AC` protocol with a user-defined AC function $plus$. 18

5 Intruder rules for the natural numbers extension. 21

6 The `Counter_Builtin_Nat` protocol using built-in natural numbers. . . . 22

7 The minimal hash chain theory. 24

8 Intruder rules for the iterated functions extension (per iterated f). . . . 25

9 The minimal hash chain theory, with iterated functions. 27

10 Adapted representation of sorts in Tamarin for user sorts. 35

11 Adapted representation of AC symbols in Tamarin for user sorts. 35

12 Adaptation of the `sortOfLTerm` function for user sorts. 35

13 Adapted representation of sorts in Tamarin for naturals. 36

14 Adapted representation of AC symbols in Tamarin for naturals. 37

15 Adaptation of the `sortOfLTerm` function for naturals. 37

16 Code for instantiating intruder rules for natural numbers. 38

17 Code for instantiating intruder rules for iterated functions. 39

18 Code for restricting intruder rules for iterated function application. . . . 40