Automated Symbolic Proofs of Observational Equivalence

David Basin
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
basin@inf.ethz.ch

Jannik Dreier
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
jannik.dreier@inf.ethz.ch

Ralf Sasse
Inst. of Information Security
Dept. of Computer Science
ETH Zurich, Switzerland
ralf.sasse@inf.ethz.ch

ABSTRACT

Many cryptographic security definitions can be naturally formulated as observational equivalence properties. However, existing automated tools for verifying the observational equivalence of cryptographic protocols are limited: they do not handle protocols with mutable state and an unbounded number of sessions. We propose a novel definition of observational equivalence for multiset rewriting systems. We then extend the TAMARIN prover, based on multiset rewriting, to prove the observational equivalence of protocols with mutable state, an unbounded number of sessions, and equational theories such as Diffie-Hellman exponentiation. We demonstrate its effectiveness on case studies, including a stateful TPM protocol.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods; K.4.4 [**Electronic Commerce**]: Security

General Terms

Security, Verification

Keywords

Protocol verification, observational equivalence, symbolic model

1. INTRODUCTION

Security protocols are the backbone of secure communication in open networks. It is well known that their design is error-prone and formal proofs can increase confidence in their correctness. Most tool-supported proofs have focused on trace properties, like secrecy as reachability and authentication as correspondence. But observational equivalence has received increasing attention and it is frequently used to express security properties of cryptographic protocols. Examples include stronger notions of secrecy and privacy-related properties of voting and auctions [12, 14, 15, 16], game-based notions such as ciphertext indistinguishability [5], and authenticated key-exchange security [6, 18].

Our focus in this paper is on symbolic models [4] for observational equivalence. The key advantage of using a symbolic model is that it enables a higher degree of automation in tools [9, 11, 8, 7, 25] for protocol analysis. These tools can quickly find errors in protocols or demonstrate their correctness with respect to symbolic abstractions. Moreover, they do not require a manual, tedious, and error-prone proof for each protocol. Unfortunately, none of the above tools are capable of analyzing protocols with mutable state for an unbounded number of sessions with respect to a security property based on observational equivalence. Note that mutable state is a key ingredient for many kinds of protocols and systems, for example to specify and analyze security APIs for hardware security modules [19].

In this paper, we develop a novel and general definition of observational equivalence in the symbolic setting of multiset rewriting systems. We present an algorithm suitable for protocols with mutable state, an unbounded number of sessions, as well as equational properties of the cryptographic operations, such as Diffie-Hellman exponentiation. Our algorithm is sound but not complete, yet it succeeds on a large class of protocols. We illustrate this through case studies using our implementation of the algorithm in the TAMARIN prover.

As case studies we verify the untraceability of an RFID protocol, and find an attack on the TPM_Envelope protocol when using deterministic encryption. Note that some protocols, such as TPM_Envelope, have been analyzed before with symbolic methods [13]. However, their analyses were carried out with respect to weaker trace-based security properties such as the unreachability of a state where the adversary can derive secrets. Formulating security properties in terms of observational equivalence is much closer to the properties used in game-based cryptographic proofs than trace properties are. For example, game-based protocol analysis often uses the standard test [21] of distinguishing *real–or–random*, where the adversary is unable to distinguish the real secret from an unrelated randomly generated value.

Contribution. We give a novel definition of observational equivalence in the multiset rewriting framework and an associated algorithm which is the first that supports mutable state, an unbounded number of sessions, and Diffie-Hellman exponentiation. We implement this algorithm in an extension of the TAMARIN prover and we demonstrate its practicality in different case studies that illustrate its features. The resulting proofs are largely automated, with limited

.

manual input needed to select proof strategies in some cases.

Structure. We introduce our general system model based on multiset rewriting in Section 2. We motivate and define observational equivalence for multiset rewrite systems in Section 3. We show how to prove observational equivalence in Section 4 and sketch its implementation in the TAMARIN prover. Afterwards we present our case studies in Section 5. We close with related work and draw conclusions in Section 6.

2. PRELIMINARIES AND MODEL

Let S^* denote the set of sequences over S. For a sequence s, we write s_i for its i-th element, $|s|$ for its length, and $idx(s) = \{1, \ldots, |s|\}$ for the set of its indices. We use $[]$ to denote the empty sequence, $[s_1, \ldots, s_k]$ to denote the sequence s of length k, and $s \cdot s'$ to denote the concatenation of the sequences s and s'.

We specify properties of functions by *equations*. Given a signature Σ_{Fun} of functions and a set V of variables, $T_{\Sigma_{Fun}}(V)$ denotes the set of terms built using functions from Σ_{Fun} and variables from V. Terms without variables are called ground terms and denoted $T_{\Sigma_{Fun}}$. An equation over the signature Σ_{Fun} is an unordered pair of terms $s, t \in T_{\Sigma_{Fun}}(V)$, written $s \simeq t$. An *equational presentation* is a pair $\mathcal{E} = (\Sigma_{Fun}; E)$ of a signature Σ_{Fun} and a set of equations E. The corresponding *equational theory* $=_{\mathcal{E}}$ is the smallest Σ_{Fun}-congruence containing all instances of the equations in E. We often leave the signature Σ_{Fun} implicit and identify the equations E with the equational presentation $\mathcal{E}$. Similarly, we use $=_E$ for the equational theory $=_{\mathcal{E}}$. We say that two terms s and t are equal modulo E iff $s =_E t$. We use the subscript E to denote the usual operations on sets, sequences, and multisets where equality is modulo E instead of syntactic equality. For example, we write $\in_E$ for set membership modulo E.

EXAMPLE 1. *To model symmetric key encryption, let Σ_{Fun} be the signature consisting of the functions $enc(\cdot,\cdot)$ and $dec(\cdot,\cdot)$ together with the equation $dec(enc(x,k),k) \simeq x$.*

We model systems using *multiset rewrite rules*. These rules manipulate multisets of *facts* which model the current state of the system, with *terms* as arguments. Formally, given a signature Σ_{Fun} and a (disjoint) set of fact symbols Σ_{Fact}, we define $\Sigma = \Sigma_{Fun} \cup \Sigma_{Fact}$, and we define the set of facts as $\mathcal{F} = \{F(t_1, \ldots, t_n) | t_i \in T_{\Sigma_{Fun}}, F \in \Sigma_{Fact} \text{ of arity } n\}$. We assume that Σ_{Fact} is partitioned into *linear* and *persistent* fact symbols; a fact $F(t_1, \ldots, t_n)$ is called linear if its function symbol F is linear, and persistent if F is persistent. Linear facts model resources that can only be consumed once, whereas persistent facts can be consumed as often as needed. We denote by $\mathcal{F}^\sharp$ the set of finite multisets built using facts from $\mathcal{F}$, and by $\mathcal{G}^\sharp$ the set of multisets of ground facts.

The system's possible state transitions are modeled by *labeled multiset rewrite rules*. A labeled multiset rewrite rule is a tuple (id, l, a, r), written $id : l -[\, a \,]\!\!\rightarrow r$, where $l, a, r \in \mathcal{F}^\sharp$ and $id \in \mathcal{I}$ is a unique identifier. Given a rule $ri = id : l -[\, a \,]\!\!\rightarrow r$, $name(ri) = id$ denotes its *name*, $prems(ri) = l$ its *premises*, $acts(ri) = a$ its *actions*, and $concs(ri) = r$ its *conclusions*. Finally $ginsts(R)$ denotes the ground instances of a set R of multiset rewrite rules, $lfacts(l)$ is the multiset of all linear facts in l, and $pfacts(l)$ is the set of all persistent facts in l. We use $mset(s)$ to highlight that s is a multiset, and we use $set(s)$ for the interpretation of s as a set, even if it is a multiset. We use regular set notation $\{\cdot\}$ for multisets as well, whenever it is clear from the context whether it is a set or a multiset.

EXAMPLE 2. *The following multiset rewrite rules describe a system that constructs terms containing nested applications of the functions $one(\cdot)$ and $two(\cdot)$ inside a fact built with the symbol M using the first three rules below. Using the final rule, E_{check}, the system can compare a constructed term with the value stored in the $\mathsf{In}_{Env}(\cdot)$ fact.*

$$\begin{aligned} Env = \{\ & E_{null} : -[]\!\!\rightarrow \mathsf{M}(null), \\ & E_{one} : \mathsf{M}(x) -[]\!\!\rightarrow \mathsf{M}(one(x)), \\ & E_{two} : \mathsf{M}(x) -[]\!\!\rightarrow \mathsf{M}(two(x)), \\ & E_{check} : \mathsf{M}(x), \mathsf{In}_{Env}(x) -[]\!\!\rightarrow \mathsf{Out}_{Env}(true) \quad \} \end{aligned}$$

In our semantics of multiset rewriting, we associate each fact F with a recipe $recipe(F)$, representing how this fact was derived. This will be important for defining observational equivalence later. Specifically, we define a sequence of the premises $seq^{\leq}(l)$ and conclusions $seq^{\leq}(r)$ of a rule $id : l -[\, a \,]\!\!\rightarrow r$ by ordering all facts under the total order $\leq$. Usually, $\leq$ will just be the lexicographic order, where if the same fact symbol appears repeatedly, we order the instances of each such fact lexicographically by the terms inside the fact. If these terms are also identical, the facts can appear in any order. Given a rule $id : l -[\, a \,]\!\!\rightarrow r$, for a fact $F \in r$, where k is the index of F in $seq^{\leq}(r)$, and $l_1, \ldots, l_n = seq^{\leq}(l)$, we have

$$recipe(F) = id_k(newvars(F), [recipe(l_1), \ldots, recipe(l_n)]),$$

where $newvars(F)$ denotes the list of new variables. New variables are those that appear in F but not in any of the premises. Thus, we include their instantiations, e.g., $[\{a/x\}]$ for the list containing the new variable x instantiated with a. This list is ordered by the appearance of the new variables inside F. This definition requires computing the recipes for the facts $l_1, \ldots, l_n$ recursively. Moreover, by abuse of notation, we define the recipe of a rule id as

$$\begin{aligned} recipe(id) = \ & id([newvars(r_1), \ldots, newvars(r_m)], \\ & [recipe(l_1), \ldots, recipe(l_n)]), \end{aligned}$$

where $r_1, \ldots, r_m = seq^{\leq}(r)$. It consists of the list of lists of new variables and the list of all recipes of the premises. We denote by ρ the set of all recipes of rules.

The semantics of a set of multiset rewrite rules P are given by a *labeled transition relation* $\rightarrow_P \subseteq \mathcal{G}^\sharp \times (\mathcal{G}^\sharp \times \rho) \times \mathcal{G}^\sharp$, defined by the transition rule:

$$\frac{ri = id : l -[\, a \,]\!\!\rightarrow r \in_E ginsts(P) \qquad lfacts(l) \subseteq^\sharp S \qquad pfacts(l) \subseteq S}{S \xrightarrow[recipe(id)]{set(a)}_P ((S \setminus^\sharp lfacts(l)) \cup^\sharp mset(r))}$$

Note that the initial state of an LTS derived from multiset rewrite rules is the empty set of facts $\emptyset$. Each transition transforms a multiset of facts S into a new multiset of facts, according to the rewrite rule used. Moreover each transition is labeled by the actions a of the rule, as well as the rule's recipe $recipe(id)$. These labels are used in our definition of observational equivalence below, for example that each interface transition must be simulated by the same transition. Since we perform multiset rewriting modulo E, we use $\in_E$

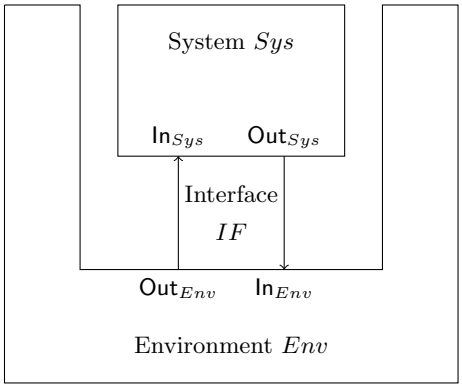

Figure 1: System model

for the rule instance. As linear facts are consumed upon rewriting, we use multiset inclusion, written $\subseteq^\sharp$, to check that all facts in $\mathit{lfacts}(l)$ occur sufficiently often in S. For persistent facts, we only check that each fact in $\mathit{pfacts}(l)$ occurs in S. To obtain the successor state, we remove the consumed linear facts and add the generated facts.

EXAMPLE 3 (PAIRS). *Consider two systems, where the first system outputs a pair of identical values*

$$S_A = \{ \quad A : -[]\rightarrow \mathsf{Out}_{Sys}((x,x)) \quad \}$$

and the second system may output two different values

$$S_B = \{ \quad B : -[]\rightarrow \mathsf{Out}_{Sys}((x,y)) \quad \}.$$

In S_A, we have that

$$\emptyset \xrightarrow[A([\{m/x\}],[])]{} \{\mathsf{Out}_{Sys}((m,m))\}.$$

In S_B, we can either take a similar transition

$$\emptyset \xrightarrow[B([\{m/x,m/y\}],[])]{} \{\mathsf{Out}_{Sys}((m,m))\}$$

or alternatively

$$\emptyset \xrightarrow[B([\{m/x,n/y\}],[])]{} \{\mathsf{Out}_{Sys}((m,n))\}.$$

3. OBSERVATIONAL EQUIVALENCE

Observational equivalence expresses that two systems appear the same to the environment. This can be used to specify security properties such as the inability of an attacker to distinguish between two instances of a protocol. It also has applications in system verification, for example in formalizing that the environment sees no difference between interacting with an ideal system or a concrete implementation. To define observational equivalence, we must model the system, the environment, and their interface.

In our model, depicted in Fig. 1, we model both the System Sys and the environment Env using multiset rewrite rules. We require that the sets of facts and rules used by the system and the environment are disjoint, and that their signatures provide "communication facts" Out_{Sys}, In_{Sys}, Out_{Env}, and In_{Env} as an interface for interaction. Their interaction is described by the following interface rules.

$$\begin{aligned} \mathit{OUT} &= \{\mathit{OUT} : \mathsf{Out}_{Sys}(M) -[\ \mathsf{O}\ ]\rightarrow \mathsf{In}_{Env}(M)\} \\ \mathit{IN} &= \{\mathit{IN} : \mathsf{Out}_{Env}(M) -[\ \mathsf{I}\ ]\rightarrow \mathsf{In}_{Sys}(M)\} \\ \mathit{IF} &= \mathit{OUT} \cup \mathit{IN} \end{aligned}$$

The OUT rule forwards the system's output to the environment's input and the IN rule forwards the environment's output to the system's input.

In our interface rules, each input and output is labeled using the action O, for system output, or I, for system input. We model that the environment can only observe these interactions, but not the internal state or transitions within the system, which should be invisible to the environment. We reflect this in the recipes by defining the recipe of the $\mathsf{In}_{Env}(M)$ fact as a conclusion of the OUT-rule differently from other facts in the system or environment rules. Namely, we define $\mathit{recipe}(\mathsf{In}_{Env}(M)) = \mathit{OUT}_1([],x)$, where x is a new variable. Similarly we define the recipe of the rule as $\mathit{recipe}(\mathit{OUT}) = \mathit{OUT}([],x)$. This replaces the recipe of the $\mathsf{Out}_{Sys}(M)$ fact, which is considered to be internal to the system, with a variable. Note that this replacement makes the book-keeping of recipes inside the system unnecessary; however we keep them in our formalization as it simplifies the definition of the LTS as we therefore do not need to distinguish between system and environment transitions.

EXAMPLE 4 (PAIRS REVISITED). *Consider the two systems from Example 3 and the following environment, which can check whether the two values in a pair are equal:*

$$\mathit{Env} = \{ \quad C : \mathsf{In}_{Env}(x,x) -[]\rightarrow \mathsf{Out}_{Env}(\mathit{true}) \quad \}.$$

Then in $S_A \cup IF \cup Env$ we have

$$\begin{aligned} \emptyset &\xrightarrow[A([\{m/x\}],[])]{} \{\mathsf{Out}_{Sys}((m,m))\} \\ &\xrightarrow[\mathit{OUT}([],z)]{O} \{\mathsf{In}_{Env}((m,m))\} \\ &\xrightarrow[C([],[\mathit{OUT}_1([],z)])]{} \{\mathsf{Out}_{Env}(\mathit{true})\}. \end{aligned}$$

In $S_B \cup IF \cup Env$ we have similar transitions:

$$\begin{aligned} \emptyset &\xrightarrow[B([\{m/x,m/y\}],[])]{} \{\mathsf{Out}_{Sys}((m,m))\} \\ &\xrightarrow[\mathit{OUT}([],z)]{O} \{\mathsf{In}_{Env}((m,m))\} \\ &\xrightarrow[C([],[\mathit{OUT}_1([],z)])]{} \{\mathsf{Out}_{Env}(\mathit{true})\}. \end{aligned}$$

Note that the first transition has a different recipe as we also must instantiate y, but the output replaces this recipe with a new variable z, which is the same on both sides. This hides from the environment how the term was constructed. However, in $S_B \cup IF \cup Env$ we also have the following transitions:

$$\begin{aligned} \emptyset &\xrightarrow[B([\{m/x,n/y\}],[])]{} \{\mathsf{Out}_{Sys}((m,n))\} \\ &\xrightarrow[\mathit{OUT}([],z)]{O} \{\mathsf{In}_{Env}((m,n))\}. \end{aligned}$$

Note that the first transition has a different recipe as we instantiate y differently, but again the output replaces this recipe with a new variable z, which is the same on both sides.

3.1 Definition

We formalize observational equivalence in the context of labeled transition systems (LTS) induced by two multiset rewrite systems. Since the environment does not know with which system it is interacting, each transition caused by an environment rule must be matched by the same rule operating on facts with the same recipes, ensuring that the environment makes the same choices in both cases. Otherwise the environment could trivially distinguish any two systems by choosing transitions depending on which system it interacts with. Similarly, all interface rules must be matched by

themselves, ensuring that an input can only be matched by an input, and an output only by an output.

In contrast to the above, transitions inside one system can be matched by any number of transitions in the other system since the environment cannot observe these steps. This is reflected by the OUT rule, where the recipe of a system output is removed, ensuring that the environment does not know how the term was constructed. Note that this still allows the environment to distinguish both systems if their behavior on a given input differs, or if they output terms that can be distinguished by the environment. An example of the latter is when a transition in the environment requiring the equality of two terms with the same recipes (i.e., deduced using the same steps from the same outputs) is possible in one system, but not in the other.

Definition 1 (Observational Equivalence $\approx$). *Two sets of multiset rewrite rules S_A and S_B are* observational equivalent *with respect to an environment given by a set of multiset rewrite rules Env, written $S_A \approx_{Env} S_B$, if, given the LTS defined by the rules $S_A \cup IF \cup Env$ and $S_B \cup IF \cup Env$, there is a relation $\mathcal{R}$ containing the initial states, such that for all states $(\mathcal{S}_A, \mathcal{S}_B) \in \mathcal{R}$ we have:*

1. *If $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$ and r is the recipe of a rule in $Env \cup IF$, then there exists actions $l' \in \mathcal{F}^\sharp$ and $\mathcal{S}'_B \in \mathcal{G}^\sharp$ such that $\mathcal{S}_B \xrightarrow[r]{l'} \mathcal{S}'_B$, and $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$.*

2. *If $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$ and r is the recipe of a rule in S_A, then there exist recipes $r_1, \ldots, r_n \in \rho$ of rules in S_B, actions $l_1, \ldots, l_n \in \mathcal{F}^\sharp$, $n \geq 0$, and $\mathcal{S}'_B \in \mathcal{G}^\sharp$ such that $\mathcal{S}_B \xrightarrow[r_1]{l_1} \ldots \xrightarrow[r_n]{l_n} \mathcal{S}'_B$, and $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$.*

Additionally, we have the same in the other direction:

3. *If $\mathcal{S}_B \xrightarrow[r]{l} \mathcal{S}'_B$ and r is the recipe of a rule in $Env \cup IF$, then there exists actions $l' \in \mathcal{F}^\sharp$ and $\mathcal{S}'_A \in \mathcal{G}^\sharp$ such that $\mathcal{S}_A \xrightarrow[r]{l'} \mathcal{S}'_A$, and $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$.*

4. *If $\mathcal{S}_B \xrightarrow[r]{l} \mathcal{S}'_B$ and r is the recipe of a rule in S_B, then there exist recipes $r_1, \ldots, r_n \in \rho$ of rules in S_A, actions $l_1, \ldots, l_n \in \mathcal{F}^\sharp$, $n \geq 0$, and $\mathcal{S}'_A \in \mathcal{G}^\sharp$ such that $\mathcal{S}_A \xrightarrow[r_1]{l_1} \ldots \xrightarrow[r_n]{l_n} \mathcal{S}'_A$, and $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$.*

3.2 Examples

We now illustrate this definition on several examples.

Example 5 (Pairs). *Consider the two systems and the environment from Example 4. In $S_B \cup IF \cup Env$ we have*

$$\emptyset \xrightarrow[B([\{m/x,n/y\}],[])]{} \{\mathsf{Out}_{Sys}((m,n))\} \xrightarrow[OUT([],z)]{O} \{\mathsf{In}_{Env}((m,n))\}.$$

The only way for $S_A \cup IF \cup Env$ to simulate this would be

$$\emptyset \xrightarrow[A([\{m/x\}],[])]{} \{\mathsf{Out}_{Sys}((m,m))\} \xrightarrow[OUT([],z)]{O} \{\mathsf{In}_{Env}((m,m))\}.$$

and potentially further transitions using rule A, adding more $\mathsf{Out}_{Sys}((m,m))$ facts to the state. Note that there can only be one $\mathsf{In}_{Env}((m,m))$-fact in the resulting state as the output transition can only be used once. This implies that for the resulting state S we have $(\{S\}, \{\mathsf{In}_{Env}((m,n))\}) \in \mathcal{R}$. However we have

$$S \xrightarrow[C([],[OUT_1([],z)])]{} \{\mathsf{Out}_{Env}(true)\},$$

but in state $\{\mathsf{In}_{Env}((m,n))\}$ no transition with the same recipe is possible, hence $S_A \not\approx_{Env} S_B$.

This simple example illustrates that if the environment can do something on one side, but not on the other, then the two sides are distinguishable and therefore not observationally equivalent. The next example illustrates the importance of recipes in our definition of observational equivalence.

Example 6. *Consider the two systems from Example 4, but a different environment Env':*

$$Env' = \{ \begin{array}{l} E_{fst} : \mathsf{In}_{Env}((x,y)) \multimap\!\!\!\!\rightarrow \mathsf{M}(x), \\ E_{snd} : \mathsf{In}_{Env}((x,y)) \multimap\!\!\!\!\rightarrow \mathsf{M}(y), \\ E_{cmp} : \mathsf{M}(x), \mathsf{M}(x) \multimap\!\!\!\!\rightarrow \mathsf{Out}_{Env}(true) \end{array} \},$$

where $\mathsf{M}(\cdot)$ is a persistent fact. Intuitively we would expect that this environment can distinguish S_A and S_B, as it can compare the first and second value of the tuple. We now try to apply the same reasoning as in Example 5. Consider

$$\begin{aligned} \emptyset &\xrightarrow[B([\{m/x,n/y\}],[])]{} \{\mathsf{Out}_{Sys}((m,n))\} \\ &\xrightarrow[OUT([],z)]{O} \{\mathsf{In}_{Env}((m,n))\} \\ &\xrightarrow[E_{fst}([],[OUT_1([],z)])]{} \{\mathsf{M}(m)\} \\ &\xrightarrow[E_{snd}([],[OUT_1([],z)])]{} \{\mathsf{M}(m), \mathsf{M}(n)\}. \end{aligned}$$

In $S_A \cup IF \cup Env'$ this can be simulated as follows:

$$\begin{aligned} \emptyset &\xrightarrow[A([\{m/x\}],[])]{} \{\mathsf{Out}_{Sys}((m,m))\} \\ &\xrightarrow[OUT([],z)]{O} \{\mathsf{In}_{Env}((m,m))\} \\ &\xrightarrow[E_{fst}([],[OUT_1([],z)])]{} \{\mathsf{M}(m)\} \\ &\xrightarrow[E_{snd}([],[OUT_1([],z)])]{} \{\mathsf{M}(m), \mathsf{M}(m)\}. \end{aligned}$$

Moreover, we can compare the first and second value of the tuple with

$$\{\mathsf{M}(m), \mathsf{M}(m)\} \xrightarrow[r_1]{} \{\mathsf{M}(m), \mathsf{M}(m), \mathsf{Out}_{Env}(true)\},$$

where

$$r_1 = E_{cmp}([], [E_{fst,1}([], [OUT_1([], z)]), E_{snd,1}([], [OUT_1([], z)])]).$$

This transition cannot be matched by $S_B \cup IF \cup Env'$. Note however that the following transition is *possible for $S_B \cup IF \cup Env'$:*

$$\{\mathsf{M}(m), \mathsf{M}(n)\} \xrightarrow[r_2]{} \{\mathsf{M}(m), \mathsf{M}(n), \mathsf{Out}_{Env}(true)\},$$

where

$$r_2 = E_{cmp}([], [E_{fst,1}([], [OUT_1([], z)]), E_{fst,1}([], [OUT_1([], z)])]).$$

The only difference between the two transition is the different recipe: instead of comparing the first and the second value of the tuple, we simply compared the first value to itself, and therefore they are not observational equivalent. This example shows that with a different environment the two systems are still distinguishable.

The next example shows how two different systems can behave in an equivalent way, and how equations can be used to model the equivalence of terms.

EXAMPLE 7 (COINS). *Consider a vending machine, in particular the part that returns coins as change when the money inserted was not fully spent. For simplicity we consider only 1 € and 2 € coins, represented by the functions one and two, and a constant null representing no coins. Now suppose we want to return 3 €. The preferred solution would be to return two coins: 1 € and 2 €. Yet returning three 1 € coins is also possible and, moreover, the order of the coins could be permuted.*

Consider again two systems. The first system specifies the optimal behavior of returning just two coins:

$$S_A = \{ \quad A : \multimap \mathsf{Out}_{Sys}(two(one(null))) \quad \}.$$

The second system, representing the actual implementation, may also return other combinations of coins. It is given by

$$\begin{aligned} S_B = \{ \quad & B_1 : \multimap \mathsf{Out}_{Sys}(two(one(null))), \\ & B_2 : \multimap \mathsf{Out}_{Sys}(one(one(one(null)))), \\ & B_3 : \multimap \mathsf{Out}_{Sys}(one(two(null))) \qquad \}. \end{aligned}$$

We now define an environment that checks whether the implementation is correct with respect to the specification. Namely, the vending machine returns the same amount of money using the same coins returned in the same order:

$$\begin{aligned} Env = \{ \quad & E_{null} : \multimap \mathsf{M}(null), \\ & E_{one} : \mathsf{M}(x) \multimap \mathsf{M}(one(x)), \\ & E_{two} : \mathsf{M}(x) \multimap \mathsf{M}(two(x)), \\ & E_{check} : \mathsf{M}(x), \mathsf{In}_{Env}(x) \multimap \mathsf{Out}_{Env}(true) \quad \}. \end{aligned}$$

The environment's test works as follows. Using the first three rules, the environment can build any amount of money from the two kinds of coins. Then, using the E_{check} rule, this can be compared to the system's output. Hence, for $S_A \approx_{Env} S_B$ to hold, both systems must output the same amount of money using the same coins in the same order, otherwise E_{check} is applicable only on one side. We have $S_A \not\approx_{Env} S_B$ as the amount of money returned is the same, but the coins may differ: S_A can only return two coins, while S_B could also return three. More precisely, the environment could build the fact $\mathsf{M}(two(one(null)))$, and try to apply the rule E_{check}. This would work for the system S_A provided an output was made, but not necessarily for the system S_B as the output could, for example, have been $one(one(one(null)))$.

Suppose that we add the equation $two(x) = one(one(x))$, stating that a 2 € coin is equivalent to two 1 € coins. Then $S_A \approx_{Env} S_B$ as the amount of money output by both machines is the same and $two(one(null)) = one(two(null)) = one(one(one(null)))$. Hence the environment successfully checks whether both systems output the same amount of money, independent of the coins used.

Naturally we can also have other environments. Assuming no equations, consider the environment

$$Env' = \{E_{comp} : \mathsf{In}_{Env}(x), \mathsf{In}_{Env}(x) \multimap \mathsf{Out}_{Env}(true)\}.$$

This environment compares whether two system outputs are equal, which is not necessarily the case for S_B, but holds for S_A.

These examples illustrate the generality of our definition of observational equivalence: as it is parametrized by the environment, it can be instantiated in different ways depending on the application context. In protocol verification, this could for example be used to model different types of attackers. Note also that in other process algebras used for protocol verification, such as the applied π-calculus, the environment is typically implicitly defined and cannot be changed.

4. PROVING OBSERVATIONAL EQUIVALENCE

To automate proofs of observational equivalence we introduce the notion of a *bi-system.*

4.1 Bi-Systems

A bi-system is a multiset rewrite system where terms may be built using the special operator $diff[\cdot,\cdot]$, indicating two possible instantiations corresponding to the left and right subterm. This use of *diff* operators was first introduced in PROVERIF [7] where bi-processes are handled in a similar fashion. Using diff-terms, one can specify two systems (left and right) with almost identical rules by one multiset rewriting system, where the only difference is how the diff-terms are instantiated. This simplifies the search for the simulation relation, as we can simply assume that each rule simulates itself, modulo the diff-terms. Nevertheless, this notion is expressive enough to specify many relevant security properties. These include all the examples mentioned in the introduction: our desired real–or–random test, privacy-related properties of voting and auctions, indistinguishability properties such as ciphertext indistinguishability, and authenticated key-exchange security. Moreover, as we show below, all examples from Section 3 can also be expressed this way.

For S a bi-system, we can obtain its *left* instance $L(S)$ by replacing each term $diff[M,N]$ in S with M. Similarly, we can obtain S's *right* instance $R(S)$ by replacing each term $diff[M,N]$ with N. These are both standard multiset rewrite systems. The goal of the algorithm we give is to prove that given a bi-system S, $L(S)$ and $R(S)$ are observationally equivalent.

We now revisit the Examples 3 and 7, starting with the tuple example.

EXAMPLE 8 (TUPLES WITH DIFF). *Using diff-terms we can define a single bi-system S that combines S_A and S_B as*

$$S = \{ \quad AB : \multimap \mathsf{Out}_{Sys}((x, diff[x,y])) \quad \},$$

where $L(S) = S_A$ and $R(S) = S_B$, as in Example 3.

EXAMPLE 9 (COINS WITH DIFF). *We create a bi-system S that merges S_A and S_B. The left-hand side of each diff-term is the specification, while the right-hand side is the implementation:*

$$\begin{aligned} S = \{ \quad & AB_1 : \multimap \mathsf{Out}_{Sys}(diff[two(one(null)), \\ & \qquad two(one(null))]), \\ & AB_2 : \multimap \mathsf{Out}_{Sys}(diff[two(one(null)), \\ & \qquad one(one(one(null)))]), \\ & AB_3 : \multimap \mathsf{Out}_{Sys}(diff[two(one(null)), \\ & \qquad one(two(null))]) \quad \}. \end{aligned}$$

Keeping the environment Env identical to Example 7 results in the bi-system S not satisfying observational equivalence. But, if we add the equation two(x) = one(one(x)), then S satisfies observational equivalence.

4.2 Dependency Graph Equivalence

To simplify reasoning, our algorithm works with *dependency graphs* rather than with the labeled transition system. Dependency graphs are a data structure that formalize the entire structure of a system execution, including which facts originate from which rules, similar to recipes. We have several reasons for using dependency graphs. First, by capturing the entire system state, they are well-suited for automated analysis using constraint solving. Second, this representation is already implemented and supported in the TAMARIN prover [24, 22], which we extend. Finally, dependency graphs naturally give rise to an equivalence relation that implies observational equivalence; however, it is substantially simpler to verify.

DEFINITION 2 (DEPENDENCY GRAPH). *Let E be an equational theory, R be a set of labeled multiset rewriting protocol rules, and Env an environment. We say that the pair $dg = (I, D)$ is a* dependency graph modulo E for R *if $I \in_E ginsts(R \cup IF \cup Env)^*$, $D \in \mathcal{P}(\mathbb{N}^2 \times \mathbb{N}^2)$, and dg satisfy the three conditions below. To state these conditions, we define the following notions. Let I be a sequence of rule instances whose indices, $idx(I)$, represent the nodes of dg. We call D the edges of dg and write $(i, u) \rightarrowtail (j, v)$ for the edge $((i, u), (j, v))$. A conclusion of dg is a pair (i, u) such that i is a node of dg and $u \in idx(concs(I_i))$. The corresponding conclusion fact is $(concs(I_i))_u$. A premise of dg is a pair (j, v) such that j is a node of dg and $v \in idx(prems(I_j))$. The corresponding premise fact is $(prems(I_j))_v$. A conclusion or premise is linear if its fact is linear.*

DG1 *For every edge $(i, u) \rightarrowtail (j, v) \in D$, it holds that $i < j$ and the conclusion fact of (i, u) is equal modulo E to the premise fact of (j, v).*

DG2 *Every premise of dg has exactly one incoming edge.*

DG3 *Every linear conclusion of dg has at most one outgoing edge.*

We denote the set of all dependency graphs of R modulo E by $dgraphs_E(R)$. Moreover, by $state(dg)$ we denote the set of all conclusion facts in dg that are either persistent or (if they are linear) do not have an outgoing edge. This intuitively corresponds to the state of the LTS after all transitions in the dependency graph have been executed.

Figures 2 and 3 contain dependency graphs corresponding to evaluations based on Examples 6 and 7, respectively.

Using dependency graphs, we can define a stronger version of observational equivalence, which is used by our algorithm. For this, we define the *dependency graphs of a rule*, which intuitively corresponds to the set of all dependency graphs having the rule as root. Given a rule $r \in R \cup IF \cup Env$, its dependency graphs $dgraphs_E(r)$ contain all dependency graphs where the last node, i.e., the node (i, u) with maximal i, is an instance of the rule r. Moreover, by new diff-variables we mean the new variables of a rule that only appear in one of its two diff-variants, e.g., y in the case of a rule $\mathsf{Out}((x, diff[x, y]))$, where x and y are new variables.

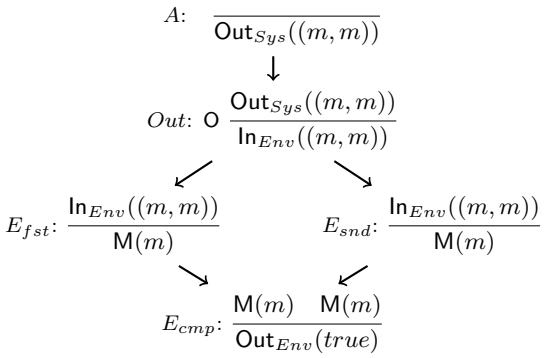

Figure 2: Dependency graph for Example 6

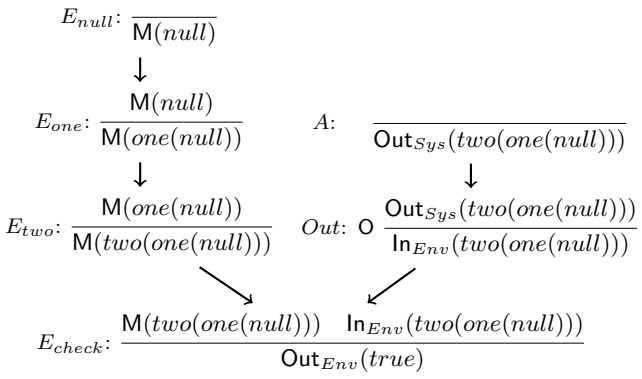

Figure 3: Dependency graph for Example 7

Finally, we define the *mirrors* of dependency graphs. Intuitively, given a dependency graph, its mirrors contain all dependency graphs on the other side of the bi-system with the same structure, notably the same edges and where the nodes are instances (potentially different due to the diff-terms) of the same rules.

Suppose that for all dependency graphs of all rules, the set of its mirrors contains all "necessary" instances. We then know that – independently of the current state of the system – if a transition is enabled by a rule on one side, the same rule also enables a transition on the other side, implying observational equivalence. This is formalized in Definition 4 below as *Dependency Graph Equivalence*.

DEFINITION 3 (MIRRORING DEPENDENCY GRAPHS). *Let S be a protocol bi-system and Env be an environment. Consider the multiset rewrite systems $L = L(S) \cup IF \cup Env$ and $R = R(S) \cup IF \cup Env$.*

Let $dg_L = (I_L, D_L) \in dgraphs(L)$ be a dependency graph. We denote by $mirrors(dg_L)$ the set of all dependency graphs $dg_R = (I_R, D_R) \in dgraphs(R)$, such that $D_R = D_L$, $|I_L| = |I_R|$, $idx(I_L) = idx(I_R)$ and for all $i \in idx(I_L)$ the ground rule instances $(I_L)_i$ and $(I_R)_i$ are ground instances of the same rules, i.e., rules with the same identifier, where new variables of rules keep their instantiation.

The set of mirrors of a dependency graph $dg_R = (I_R, D_R) \in dgraphs(R)$, denoted by $mirrors(dg_R)$, is defined analogously, replacing R by L uniformly in the above definition.

Using these notions, we now define *dependency graph equiv-*

alence. Intuitively, this definition captures that for all dependency graphs on one side of the bi-system, there is a mirroring dependency graph on the other side that respects its instantiations of new diff-variables.

Definition 4 (Dependency Graph Equivalence). *Let S be a bi-system. Consider the multiset rewrite systems $L = L(S) \cup IF \cup Env$ and $R = R(S) \cup IF \cup Env$. We say that L and R are dependency graph equivalent, written $L(S) \sim_{DG,Env} R(S)$, if for all dependency graphs dg of rules $r \in L \cup R$, the set mirrors(dg) is non-empty and contains dependency graphs for all possible instantiations of new diff-variables.*

Note that this definition requires that if, for example, there are new variables in the rules R not appearing in the rules from L used in dg, then *mirrors*(dg) must contain instances for all possible instantiations of these variables. For instance, in the case of a rule producing a conclusion $\mathsf{Out}((x, \mathit{diff}[x, y]))$, then for all possible instantiations of y, an instance must be in *mirrors*(dg).

It turns out that dependency graph equivalence is a sufficient (but not necessary) criterion for observational equivalence. Intuitively, dependency graph equivalence verifies that the left-hand side instance of a rule can always be simulated by its right-hand side, and vice versa.

Theorem 1. *Let S be a bi-system. If $L(S) \sim_{DG,Env} R(S)$ then $L(S) \approx_{Env} R(S)$.*

Proof. Consider the multiset rewrite systems $L = L(S) \cup IF \cup Env$ and $R = R(S) \cup IF \cup Env$, and the relation $\mathcal{R}$:

$$\begin{aligned}\mathcal{R} = \{(\mathcal{S}_A, \mathcal{S}_B) \mid\ & \mathcal{S}_A = \mathit{state}(dg_L), \mathcal{S}_B = \mathit{state}(dg_R),\\ & dg_R \in \mathit{mirrors}(dg_L), dg_L \in \mathit{dgraphs}(L)\}\\ \cup \{(\mathcal{S}_A, \mathcal{S}_B) \mid\ & \mathcal{S}_A = \mathit{state}(dg_L), \mathcal{S}_B = \mathit{state}(dg_R),\\ & dg_L \in \mathit{mirrors}(dg_R), dg_R \in \mathit{dgraphs}(R)\}.\end{aligned}$$

First note that $(\emptyset, \emptyset) \in \mathcal{R}$. We now show that $\mathcal{R}$ is an observational equivalence relation as defined in Definition 1. For this, we must show that for all states $(\mathcal{S}_A, \mathcal{S}_B) \in \mathcal{R}$ we have:

1. If $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$ and r is the recipe of a rule in $IF \cup Env$, then there exists actions $l' \in \mathcal{F}^\sharp$ and $\mathcal{S}'_B \in \mathcal{G}^\sharp$ such that $\mathcal{S}_B \xrightarrow[r]{l'} \mathcal{S}'_B$, and $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$.

2. If $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$ and r is the recipe of a rule in S_A, then there exist recipes $r_1, \ldots, r_n \in \rho$ of rules in S_B, actions $l_1, \ldots, l_n \in \mathcal{F}^\sharp$, $n \geq 0$, and $\mathcal{S}'_B \in \mathcal{G}^\sharp$ such that $\mathcal{S}_B \xrightarrow[r_1]{l_1} \ldots \xrightarrow[r_n]{l_n} \mathcal{S}'_B$, and $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$.

We distinguish the following cases:

1. Assume $(\mathcal{S}_A, \mathcal{S}_B) \in \mathcal{R}$, $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$ for a rule instance ri, and r is the recipe of a rule in $IF \cup Env$. Then, by the definition of $\mathcal{R}$, there is a dependency graph $dg_L \in \mathit{dgraphs}(L)$ with $\mathcal{S}_A = \mathit{state}(dg_L)$, and a dependency graph $dg_R \in \mathit{dgraphs}(R)$ with $\mathcal{S}_B = \mathit{state}(dg_R)$. Since the transition $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$ is possible in $\mathcal{S}_A$, dg_L can be extended to dg'_L with the rule instance ri corresponding to this transition, and $\mathit{state}(dg'_L) = \mathcal{S}'_A$. Then $dg'_L \in \mathit{dgraphs}(L)$, and by $L(S) \sim_{DG,Env} R(S)$ we have that for all possible instantiations of new diff variables, the corresponding dependency graph $dg'_R \in \mathit{dgraphs}(R)$. By the definition of $\mathcal{R}$, in dg_R the instantiations of the new variables (including the new diff-variables) correspond to the instantiations of some $dg'_R \in \mathit{mirrors}(dg'_L)$. Then, by the construction of $\mathit{mirrors}(dg'_L)$, dg'_R is identical to dg_R except for the last rule instance ri'. Moreover, by the construction of $\mathit{mirrors}(dg'_L)$, ri' is an instance of the rule with the same identifier. Since the dependency graph dg'_R has the same structure D as dg'_L and all rules in $IF \cup Env$ have no new diff-variables, there exists a transition $\mathcal{S}_B \xrightarrow[r]{l'} \mathcal{S}'_B$ with the same recipe as ri. Moreover, $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$ since there are mirroring dependency graphs for $\mathcal{S}'_A$ and $\mathcal{S}'_B$.

   The symmetric case is analogous.

2. Alternatively, assume $(\mathcal{S}_A, \mathcal{S}_B) \in \mathcal{R}$, $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$, and r is the recipe of a rule in $L(S)$. Then, by the definition of $\mathcal{R}$, there is a dependency graph $dg_L \in \mathit{dgraphs}(L)$ with $\mathcal{S}_A = \mathit{state}(dg_L)$. Since in this state the transition $\mathcal{S}_A \xrightarrow[r]{l} \mathcal{S}'_A$ is possible, dg_L can be extended to dg'_L with the rule instance ri corresponding to this transition, and $\mathit{state}(dg'_L) = \mathcal{S}'_A$. Then $dg'_L \in \mathit{dgraphs}(L)$, and by $L(S) \sim_{DG,Env} R(S)$ we have that for all possible instantiations of new diff variables, the corresponding dependency graph $dg'_R \in \mathit{dgraphs}(R)$. By the definition of $\mathcal{R}$, there is a dependency graph $dg_R \in \mathit{dgraphs}(R)$ with $\mathcal{S}_B = \mathit{state}(dg_R)$, where the instantiations of the new variables (including the new diff-variables) correspond to the instantiations of some $dg'_R \in \mathit{mirrors}(dg'_L)$. Then, by the construction of $\mathit{mirrors}(dg'_L)$, this graph dg'_R is identical to dg_R except for the last rule instance. By assumption, ri was an instance of a rule in $L(S)$. Therefore, by the construction of $\mathit{mirrors}(dg'_L)$, the last rule instance ri' in dg'_R is an instance of the rule with the same identifier. Hence there exists a transition $\mathcal{S}_B \xrightarrow[r']{l'} \mathcal{S}'_B$. Moreover, $(\mathcal{S}'_A, \mathcal{S}'_B) \in \mathcal{R}$ since there are mirroring dependency graphs for $\mathcal{S}'_A$ and $\mathcal{S}'_B$.

   Again, the symmetric case is analogous.

□

As shown in [24], we can use constraint solving to find (restricted) normal dependency graphs; for a detailed discussion about the constraint solving procedure used and the link to restricted normal dependency graphs, see the extended version of this paper [1]. This provides the basis for our algorithm, depicted in Figure 4, which determines whether $L(S) \sim_{DG} R(S)$ holds. For each rule r in $L(S)$, $R(S)$, and the environment, the algorithm finds all corresponding normal dependency graphs with r as a root using constraint solving. For each of these dependency graphs, it then checks whether the set of mirrors contains all instances required for normal dependency graph equivalence. If this holds, it reports that verification is successful; otherwise it returns the dependency graph that lacks a mirroring instance as a counterexample. Note that these instances are counterexamples to dependency graph equivalence, but not necessarily to observational equivalence because of the approximation requiring that each rule is simulated by itself.

1: **function** VERIFY(S)
2: $RU \leftarrow L(S) \cup R(S) \cup IF \cup Env$
3: **while** $RU \neq \emptyset$ **do**
4: choose $r \in RU$, $RU \leftarrow (RU \setminus \{r\})$
5: compute $DG \leftarrow dgraphs(r)$ by constraint solving
6: **if** $\exists dg \in DG$ s.t. $mirrors(dg)$ lacks ground instances
7: **then return** "potential attack found: ", dg
8: **return** "verification successful"

Figure 4: Pseudocode of our verification algorithm.

This is due to the undecidability of the initial problem, and all related tools [9, 11, 8, 7, 25] also have this limitation.

We now explain how we adapt and use this algorithm in TAMARIN. We provide examples of its use in Section 5.

4.3 Tamarin

The TAMARIN prover [24, 22] is a security protocol verification tool that supports both the falsification and unbounded verification of security protocols specified as multiset rewriting systems with respect to trace-based properties.

In TAMARIN, a security protocol P's executions are modeled by its set of traces, defined as the concatenation of the sets of action labels at each step. A trace is a sequence of facts denoting the sequence of actions taken during a protocol's execution. The *trace* of an execution

$$S_0, (l_1 \xrightarrow[rec_1]{a_1} r_1), S_1, \ldots, S_{k-1}, (l_k \xrightarrow[rec_k]{a_k} r_k), S_k$$

is the sequence of the multisets of its action labels $[a_1, \ldots, a_k]$.

We now briefly recall the TAMARIN prover's adversary message derivation rules MD. To define the protocol and adversary rules, we assume that Σ_{Fact} includes the persistent fact symbol K modeling messages known to the adversary, the linear fact symbol Out modeling messages sent by the protocol, and the linear fact symbol In modeling messages sent by the adversary. The adversary's message deduction capabilities are captured by the following set of rules.

$$\begin{aligned} MD = \{\ & \mathsf{Out}(t) \multimap [] \to \mathsf{K}(t),\ \mathsf{K}(t) \multimap [\ \mathsf{K}(t)\ ] \to \mathsf{In}(t), \\ & \mathsf{Fr}(x\colon fr) \multimap [] \to \mathsf{K}(x\colon fr),\ [] \multimap [] \to \mathsf{K}(x\colon pub)\ \} \\ \cup \{\ & \mathsf{K}(t_1), \ldots, \mathsf{K}(t_n) \multimap [] \to \mathsf{K}(f(t_1, \ldots, t_n)) \mid f \in \Sigma^n_{Fun}\ \} \end{aligned}$$

The adversary learns all messages that are sent and it can send any message it knows (i.e., it learns or can derive) to the protocol. It can generate fresh values and it knows all public values. Additionally, the adversary can apply all operators to terms it knows. When using an equational theory, each of the equations gives rise to a deconstruction rule that lets the intruder derive the result. For example for symmetric encryption and decryption, with the equation $sdec(senc(m,k),k) = m$, TAMARIN automatically generates the rule $\mathsf{K}(senc(m,k)), \mathsf{K}(k) \multimap [] \to \mathsf{K}(m)$, which the adversary uses to decrypt messages.

We also add one new adversary deduction rule to MD, which we call $IEquality$, which allows the adversary to compare two values for equality:

$$IEquality : \mathsf{K}^{\downarrow}(x), \mathsf{K}^{\uparrow}(x) \multimap [] \to .$$

The use of $\mathsf{K}^{\downarrow}$ and $\mathsf{K}^{\uparrow}$ in this rule restricts how the adversary can derive the terms.[1] Here, this annotation is crucial as we want to compare two terms that are derived separately. Moreover, it also prevents immediate non-termination: otherwise, once two values are successfully compared, one could compare their hashes, followed by their hashes, etc.

The $IEquality$ rule is applicable whenever one side of a bi-system can construct the same value twice (but in different ways), that is it has dependency graphs as premises for both instances of x. Note that the other bi-system side trivially has the mirroring dependency graph if an output is compared with itself (same dependency graph twice), for example. But, if on one side the adversary can decrypt a message and compare it with the content, while on the other side that is not possible, then the $IEquality$ rule will expose this. The analysis of the $IEquality$ rule presented in Examples 10 and 11 illustrates this point in more detail.

For the details of our modification to the TAMARIN prover we refer the reader to the extended version of this paper [1]. There we show how our model can be instantiated for TAMARIN, and present an implementation of the above algorithm for verifying observational equivalence with TAMARIN. Note that just like the algorithm outlined in Figure 4 (line 4), TAMARIN carries out a rule-by-rule analysis.

The main challenges in implementing our algorithm in the TAMARIN prover relate to limiting the size of the state-space, which requires fine-tuning TAMARIN's internal heuristic. To aid termination, we restrict traces to normal forms as much as possible. Moreover, compared to the original TAMARIN prover, we needed to remove some of its normal form conditions because they are sound for trace properties, but not for observational equivalence. One such example is the normal form condition prohibiting repeated adversarial derivation of a term. However, equality comparison with previous values must be possible, e.g., to test whether the output equals the input in the protocol $\mathsf{In}(x), \mathsf{Fr}(y) \multimap [] \to \mathsf{Out}(diff(x,y))$.

5. CASE STUDIES

We now present four case studies. We first apply TAMARIN to two standard examples that have been analyzed using other tools. Afterwards we present two examples: one that is outside the scope of previous work and one that verifies a practical RFID protocol that had previously received manual analysis only. Note that all proofs are constructed in our tool completely automatically, with the exception of the attack on TPM_Envelope. For this protocol, interaction was limited to human input at a few key choice points and the remainder was automated. We provide a file containing the steps necessary to derive the identified attack for TPM_Envelope. For the other protocols, we just give their specification as TAMARIN's built-in strategy finds the proofs. All example files can be loaded into our extension of TAMARIN and are available at [1], together with TAMARIN. From now on, we use TAMARIN to refer to our extension, instead of the original TAMARIN.

5.1 Motivating examples

We start with two well-known examples from [7, 10]. For each example, we explain how TAMARIN determines observational equivalence using the algorithm VERIFY, presented in Figure 4.

[1] In all other rules K is actually also either $\mathsf{K}^{\downarrow}$ or $\mathsf{K}^{\uparrow}$. However, this distinction is required only for automation (namely state-space reduction and improving termination) so we have omitted it for ease of presentation. Full details can be found in the extended version [1].

Example 10 (Probabilistic encryption). Consider the equational theory:

$$pdec(penc(m, pk(k), r), k) \simeq m\,.$$

This equation gives rise to the following *decryption rule* for probabilistic encryption for the adversary, which is automatically generated by Tamarin:

$$Dpenc : \mathsf{K}(penc(m, pk(k), r)), \mathsf{K}(k) \multimap \mathsf{K}(m)\,.$$

We now express, as a bi-system, that a probabilistic encryption cannot be distinguished from a random value:

$$\begin{array}{rll} S = \{ & GEN: & \mathsf{Fr}(k) \multimap \mathsf{Key}(k), \mathsf{Out}(pk(k)) \\ & ENC: & \mathsf{Key}(k), \mathsf{Fr}(r_1), \mathsf{Fr}(r_2), \mathsf{In}(x) \multimap \\ & & \mathsf{Out}(\mathit{diff}[r_1, penc(x, pk(k), r_2)]) \quad \}\,. \end{array}$$

We summarize below how Tamarin automatically proves this property. The algorithm Verify (line 2) first constructs the set RU of rules to be analyzed,

$$\begin{array}{rl} RU = \{ & L(GEN), R(GEN), L(ENC), R(ENC), \\ & \textsc{Fresh}_{Sys}, \textsc{Fresh}_{Env}, IEquality, Dpenc\}\,, \end{array}$$

together with the remaining rules in IF and Env. Recall that $L(name)$ represents the rule $name$ instantiated with the left side of the diff-term, and likewise for $R(name)$ with the right side. Then Verify iterates over all rules (lines 3–4) until either an attack is found (line 7) or all rules have been checked and the verification is complete (line 8), which happens in this example.

We now describe, for each rule, how Verify processes it. Verify first generates dependency graphs with the rule as the root (line 5). Afterwards, for each resulting dependency graph, it looks for a mirror (line 6) that contains all instances required by the definition of normal dependency graph equivalence. In this example, it always finds a mirror and verification therefore succeeds. Due to space and readability constraints, we present the left-diff instantiation and right-diff instantiation of each rule together, even though Tamarin analyzes them independently. Due to space constraints, we also do not explicitly present the dependency graphs; however, we do explain how they are mirrored in each case so that the verification succeeds.

- As rule GEN does not contain a diff-term, the left diff-instantiation of this rule is identical to the right diff-instantiation. The rule has only a single fresh fact as its premise and thus any dependency graph with this rule at its root contains only those two rule instances and is trivially mirrored by itself.
- The rule ENC has the same premises in the left- and right-hand side system and is therefore identical for the purpose of dependency graph computation with the ENC rule as root. (Note that outputs will be considered using the equality rule below.) The two fresh premises will result in identical dependency graphs, while the key and message reception input are independent. Hence both of them will have identical dependency graphs as premises, and the resulting dependency graphs are identical (up to the outputs) and therefore mirror each other.
- The fresh rules $\textsc{Fresh}_{Sys}$ and $\textsc{Fresh}_{Env}$ have no premises. Hence the dependency graphs with them as root are just their instances, which mirror each other in the left- and right-hand system.
- For an equality rule instance of $IEquality$ as the root of a dependency graph, the two premises are the same instance of a variable x. If both of the premises are adversary generated, then the resulting dependency graphs are the same in the left- and right-hand system, and thus will mirror themselves trivially. Alternatively, if one of the premises uses the output of an instance of either the ENC rule or the GEN rule, then there is no dependency graph with a matching second premise. This is because all system outputs, $pk(k)$ for GEN and $r1$ or $penc(x, pk(k), r2)$ for ENC, contain a fresh value, k, $r1$, respectively $r2$, that is never available to the intruder. As this will never allow a complete dependency graph to be derived, no mirroring dependency graph is needed.
- For the decryption rule generated for the probabilistic encryption, this rule is never applicable on either side as the adversary never receives the keys needed for decrypting system generated encryptions. As there is no dependency graph, no mirroring one is needed. (One might mistakenly think that this rule might apply to intruder-generated terms. However, this is not the case due to the restrictions on how the adversary may combine its knowledge ($\mathsf{K}^{\downarrow}$ vs $\mathsf{K}^{\uparrow}$) and, in any case, both sides would use the same dependency graphs as premise, so the result would be the same.)
- For all other adversary rules, it is obvious that they result in identical dependency graphs on both sides. More precisely: construction rules have adversary knowledge input and thus the same dependency graphs as premises. For the deconstruction rules, the only relevant one is the previous decryption rule, as that is the only one that can use information coming out of the system; all other rules can only be used on adversary-generated terms and thus have the same dependency graphs as premises.

This completes our summary of Tamarin's verification of observational equivalence for this example. Tamarin automatically constructs the proof in under 0.2 seconds.

Our next example is Decisional Diffie-Hellman as discussed in [7, Example 2]. Tamarin verifies the expected result that the adversary cannot distinguish a Diffie-Hellman tuple from a random tuple. Note that in contrast to [7], which uses an equational theory restricted just to the commutativity of two exponents, Tamarin supports a substantially more comprehensive model of Diffie-Hellman exponentiation.

Example 11 (Decisional Diffie-Hellman). We use the equational theory for Diffie-Hellman exponentiation with an abelian group of exponents as provided by Tamarin. Hence no additional adversary rules are needed.

We consider a single rule, which outputs the two half-keys and challenges the adversary to distinguish the actual key from an unrelated randomly generated key:

$$\begin{array}{rll} GEN: & \mathsf{Fr}(a_1), \mathsf{Fr}(a_2), \mathsf{Fr}(a_3) \multimap \\ & \mathsf{Out}(b^{a_1}, b^{a_2}, \mathit{diff}[b^{a_3}, (b^{a_1})^{a_2}])\,. \end{array}$$

Using the Verify algorithm, Tamarin collects the rules

$$\begin{array}{rl} RU = \{ & L(GEN), R(GEN), \\ & \textsc{Fresh}_{Sys}, \textsc{Fresh}_{Env}, IEquality\}\,, \end{array}$$

together with the remaining rules in IF and Env. We consider the processing of these rules below, where we again combine the treatment of left-diff instantiations and right-diff instantiations of system rules to improve readability.

- The rule GEN has only fresh facts as premise and thus any dependency graph with this rule at its root contains at most four rule instances, three of fresh rules and one of GEN itself. Thus, it is mirrored trivially by itself. The mirror is actually identical (up to the output).
- The fresh rules FRESH_{Sys} and FRESH_{Env} do not have premises. Hence the dependency graphs with them as roots are just their instances, which mirror each other on the left- and right-hand side.
- For an equality rule instance of $IEquality$ as the root of a dependency graph, the two premises are the same instance of a variable x. If both of the premises are adversary generated, then the resulting dependency graphs are the same in the left- and right-hand system, and thus will mirror themselves trivially. Alternatively, if one of the premises uses the output of an instance of the GEN rule, then there is no dependency graph with a matching second premise, except the one using the same source twice. This is because all of the system outputs cannot be related in meaningful fashion within the Diffie-Hellman exponentiation theory as it does not allow the extraction of exponents, which corresponds to computing discrete logs. As this will never allow a complete dependency graph to be derived, no mirroring dependency graph is needed. In the case of the same source being used twice, i.e., a value being compared with itself, the same premise dependency graphs work for both systems.

  Additionally, note that multiple instances of the GEN rule are entirely unrelated and do not provide any advantage for the adversary. TAMARIN analyzes this and computes all possible variants, determining that no combination is useful.
- For all other adversary rules, it is obvious that they result in identical dependency graphs on both sides. Namely, the construction rules have adversary knowledge input and thus the same dependency graphs as premises.

The VERIFY algorithm therefore returns that verification is successful. TAMARIN verifies this, completely automatically, in 15.2 seconds.

This concludes our two motivating examples. They were small enough that we could give relatively detailed descriptions of VERIFY's workings. For subsequent examples, we will be more concise. Readers interested in the full gory details may generate them themselves by using TAMARIN and running the files for each case study.

5.2 Feldhofer's RFID protocol

The RFID protocol due to Feldhofer et al. [17] is of practical interest as it can be implemented with relatively few logic gates using AES encryption and hence it fits well with the requirements of current RFID chips. We use the description from [26] as the basis of our model, which we present in

$$\begin{array}{ll} R \to T: & nr \\ T \to R: & \{|nr, nt|\}_{k(R,T)} \\ R \to T: & \{|nt, nr|\}_{k(R,T)} \end{array}$$

Figure 5: RFID protocol

Figure 5 using Alice&Bob notation. The protocol is between a reader R and a tag T that share a key $k(R,T)$. Note that $\{|\dots|\}_k$ denotes symmetric encryption in this example.

In the first message, the reader sends a random nonce to the tag. In the second message, the tag sends back that nonce and one of its own choosing, encrypted with the shared key. In the third message, the reader responds with the same nonces in reverse order, also encrypted.

The desired security property for this protocol is privacy for the tags. Namely, if at least two tags share a key with a reader (and in practice, there will be many such tags), then the adversary cannot determine which tag is actually communicating with the reader. TAMARIN verifies this in under 1.6 seconds.

5.3 TPM_Envelope protocol

We first briefly explain the key part of this protocol [13] and afterwards present the TAMARIN rules along with further details. A stateful Trusted Platform Module (TPM) creates a one-use public/private key pair, and publishes the public key. A participant Alice then encrypts a nonce (the *secret*) with the public key (creating an *envelope*), which she sends to a participant Bob. Bob then either requests from the TPM the envelope's content, learning the secret, or requests a TPM-signed certificate stating that he did not ask for the content. In both cases, the TPM complies with the request and changes its state in such a way that it can afterwards only comply with repetitions of the first request, but never with the other request. This is where mutable state is crucial, i.e., the original capability of issuing either the certificate or the secret is revoked once the choice is made. The trace-based secrecy property verified in [13] states that the adversary may learn either the certificate or the secret, but not both.

We investigated whether this protocol additionally satisfies the real–or–random property for the secret. We therefore added a real–or–random challenge to the end of Alice's protocol execution, which sends out either the real secret or a random value. TAMARIN fails to prove this property and instead returns a simple attack, provided the encryption used is deterministic. The attack is as follows: the adversary (impersonating Bob) asks for the proof of never having received the secret from the TPM, and thus he must be unable to learn the secret. But, he can still distinguish whether the real–or–random challenge emits the real secret or a random value. He does this using the previously published public key, and encrypting the emitted value with it. Afterwards, he compares the resulting encryption to the envelope, and he learns that it is the real secret if it matches the envelope and a random value otherwise.

Inspecting this attack it is easy to see that it fails when probabilistic encryption is used instead. The adversary can still encrypt the emitted value with the public key, but the equality check against the envelope will always fail because the added randomness is different in the envelope and the adversary-generated comparison encryption.

In Figure 6 we present the rules used to model the pro-

tocol TPM_Envelope in TAMARIN. Note that we omit here some details, which can be found in the model file included at [1]. First we explain the rules concerning the TPM's *platform configuration registers* (PCR). The *Init* rule initializes the PCR to the initial string $'pcr0'$, and generates the fresh authentication identification key *aik* stored in the persistent AIK fact and sends out the public key $pk(aik)$. This models a long-term key for the TPM. The extension rule *Extend* allows any PCR to be extended to the hash of the concatenation of its previous value and an input. This is used when a client later either extends the PCR with $'deny'$ or $'obtain'$. This changes the TPM's state to allow creation of a certificate that the envelope was not opened ($'deny'$), or respectively opens the envelope ($'obtain'$). Rule *CertK* certifies a public key for which the TPM has stored the associated private key in the persistent key table fact KT with a particular lock. Locks are PCR values and the TPM will only extract the private key when the PCR value matches this lock. In the rule *Quote*, the current PCR value is sent out authentically, signed with the TPM's long-term key. The TPM's last rule is *Unbind*. It takes an envelope as input, and if the public key used to encrypt the envelope matches the private key in the key table, where additionally the lock in the key table matches the current PCR value, then the message in the envelope is decrypted and sent out.

The participant Alice requests an envelope key in her first rule *A*1 by extending the PCR with a nonce n of her choice. In rule *A*2, she creates the secret to be put in the envelope encryption and checks that the TPM certifies that the key can only be obtained if the PCR state is extended with $'obtain'$ and then she uses the certified public key to encrypt her secret. Alice then publishes the envelope while keeping state in A2 for her next rule and in A2ror for the real–or–random challenge. Rule *A*3 uses the state in A2 to check that the TPM's PCR was extended with $'deny'$ (which means it has not yet, and can now no longer, decrypt the envelope) and then notes the action Denied. We are only interested in traces where the adversary can show this certificate. The rule *CLKey* is used with $'obtain'$ as the *lock* input to add a new private key to the TPM's key table that is used in rule *A*2. It can of course be used with other inputs, but the resulting keys are not interesting to us. Now the key can only be extracted with a PCR extended with $'obtain'$, and thus the certificate with $'deny'$ is unavailable. The last rule is the *ROR* rule; this either outputs the real secret or a random value.

The TAMARIN prover finds the attack described above for the observational equivalence of the TPM_Envelope protocol. Note that this is a stronger property than trace-based secrecy, which had been verified by [13], so the two results are compatible. We therefore conclude that this protocol should only be used with probabilistic encryption, and not with deterministic encryption.

This example illustrates TAMARIN's handling of mutable state, which other tools cannot handle. It also illustrates the difference between trace-based properties and observational equivalence-based properties.

$$
\begin{aligned}
\mathit{Init}: &\ \mathsf{Fr}(aik) \multimap \mathsf{PCR}('pcr0'), \mathsf{AIK}(aik), \mathsf{Out}(pk(aik))\\
\mathit{Ext}: &\ \mathsf{PCR}(x), \mathsf{In}(y) \multimap \mathsf{PCR}(h(x,y))\\
\mathit{CertK}: &\ \mathsf{AIK}(aik), \mathsf{KT}(lock, sk) \multimap\\
&\ \mathsf{Out}(sign(\langle 'certk', lock, pk(sk)\rangle, aik))\\
\mathit{Quote}: &\ \mathsf{PCR}(x), \mathsf{AIK}(aik) \multimap\\
&\ \mathsf{PCR}(x), \mathsf{Out}(sign(\langle 'certcpr', x\rangle, aik))\\
\mathit{Unbind}: &\ \mathsf{PCR}(x), \mathsf{KT}(x, sk), \mathsf{In}(aenc(m, pk(sk))) \multimap\\
&\ \mathsf{PCR}(x), \mathsf{Out}(m)\\
\mathit{A1}: &\ \mathsf{Fr}(n), \mathsf{PCR}(x) \multimap \mathsf{PCR}(h(x,n)), \mathsf{A1}(n)\\
\mathit{A2}: &\ \mathsf{Fr}(s), \mathsf{A1}(n), \mathsf{AIK}(aik),\\
&\ \mathsf{In}(sign(\langle 'certk', h(h('pcr0', n), 'obtain'), pk\rangle, aik))\\
&\ \multimap \mathsf{Out}(aenc(s, pk)), \mathsf{A2}(n, s), \mathsf{A2ror}(s)\\
\mathit{A3}: &\ \mathsf{In}(sign(\langle 'certpcr', h(h('pcr0', n), 'deny')\rangle, aik)),\\
&\ \mathsf{A2}(n, s), \mathsf{AIK}(aik) \multimap[\ \mathsf{Denied}(s)\ ]\to\\
\mathit{CLKey}: &\ \mathsf{Fr}(sk), \mathsf{PCR}(x), \mathsf{In}(lock) \multimap\\
&\ \mathsf{PCR}(x), \mathsf{KT}(h(x, lock), sk), \mathsf{Out}(pk(sk))\\
\mathit{ROR}: &\ \mathsf{A2ror}(s), \mathsf{Fr}(f) \multimap \mathsf{Out}(\mathit{diff}[s, f])
\end{aligned}
$$

Figure 6: Rule set modeling the TPM_envelope

6. RELATED WORK AND CONCLUSION

We have shown how to take the well-established modeling formalism of multiset rewriting and extend it with a novel definition of observational equivalence. The result is well-suited for the verification of cryptographic protocols, as well as other applications. Based on this, we have implemented an algorithm to prove observational equivalence for protocols specified in multiset rewriting and demonstrated its effectiveness on a number of case studies. Combining TAMARIN's constraint solving with the bi-system notion results in our approach's high degree of automation.

Our equivalence notion has similarities with other notions of observational equivalence considered in the literature, including trace equivalence [11], bisimulation [2], and notions based on contexts [2, 11, 7]. However, multiset rewriting and our observational equivalence definition are more flexible than the previous approaches as we can choose the environment as well as the underlying equational theory. As illustrated in Example 5 in Section 3, this can, for example, be used to model different types of attackers. In process algebras used for protocol verification, like the applied π-calculus, the environment is implicitly defined and cannot be changed. Moreover, we support mutable state and a larger set of equational theories than other approaches as detailed below.

Various other tools exist for verifying notions of observational equivalence. APTE [9, 11] and AKISS [8] both verify trace equivalence, but are limited to a bounded number of sessions. Moreover, AKISS does not support non-trivial else branches or private channels. PROVERIF [7] verifies observational equivalence in the applied π-calculus for an unbounded number of sessions, but it cannot handle mutable state [3], for example, a protocol that switches between the states a and b. Extensions for PROVERIF that can deal with Diffie-Hellman equational theories [20] do not support observational equivalence. Note that our approach's restriction to bi-systems is similar to PROVERIF's restriction to bi-processes. SPEC [25] verifies open bisimulation in the spi-calculus, but unlike our approach it only supports a fixed number of cryptographic primitives and is limited to a bounded number of sessions.

In contrast to the above, there are tools like STATVERIF [3] and SAPIC [19] that support mutable state. However, they cannot verify observational equivalence. Similarly, TAMARIN, which is used as SAPIC's back-end, supports mutable state, an unbounded number of sessions, and also Diffie-Hellmann equational theories. However, prior to our extension, it

could not prove any notion of observational equivalence.

Another multiset rewriting-based approach that supports observational equivalence is Maude-NPA [23]. It creates the synchronous product of two very similar protocols, similar to our use of bi-systems. Their approach suffers from termination problems [23] and thus presents only attacks.

As future work, we plan to extend our approach so that the verification of observational equivalence is also possible when one rule must be matched by a different rule, or even by multiple rules. We will also tackle protocols with loops, where proofs will likely require induction. Moreover, we intend to look at larger protocols, such as authenticated key exchange protocols with perfect forward secrecy, such as NAXOS and its variants.

7. REFERENCES

[1] Tamarin – tool and extended papers. `http://www.infsec.ethz.ch/research/software/tamarin.html`.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, New York, 2001. ACM.

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

[4] David Basin, Cas Cremers, and Catherine Meadows. Model checking security protocols. In *Handbook of Model Checking*, chapter 24. Springer, 2015. To appear.

[5] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In *CRYPTO*, volume 1462 of *LNCS*, pages 26–45. Springer, 1998.

[6] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO*, volume 773 of *LNCS*, pages 232–249. Springer, 1993.

[7] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, February–March 2008.

[8] Rohit Chadha, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. In Helmut Seidl, editor, *ESOP*, volume 7211 of *LNCS*, pages 108–127. Springer, 2012.

[9] Vincent Cheval. APTE: An algorithm for proving trace equivalence. In *TACAS*, volume 8413 of *LNCS*, pages 587–592. Springer, 2014.

[10] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *Principles of Security and Trust (POST)*, volume 7796 of *LNCS*, pages 226–246. Springer, 2013.

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

[12] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17:435–487, December 2009.

[13] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. Formal analysis of protocols based on TPM state registers. In *CSF*, pages 66–80. IEEE, 2011.

[14] Jannik Dreier, Pascal Lafourcade, and Yassine Lakhnech. Defining privacy for weighted votes, single and multi-voter coercion. In *ESORICS*, volume 7459 of *LNCS*, pages 451–468. Springer, 2012.

[15] Jannik Dreier, Pascal Lafourcade, and Yassine Lakhnech. A formal taxonomy of privacy in voting protocols. In *Proceedings of IEEE International Conference on Communications (ICC'12)*, pages 6710–6715, Ottawa, ON, Canada, 2012. IEEE.

[16] Jannik Dreier, Pascal Lafourcade, and Yassine Lakhnech. Formal verification of e-auction protocols. In *Proceedings of the 2nd Conference on Principles of Security and Trust (POST'13)*, volume 7796 of *LNCS*, pages 247–266, Rome, Italy, 2013. Springer Verlag.

[17] Martin Feldhofer, Sandra Dominikus, and Johannes Wolkerstorfer. Strong authentication for RFID systems using the aes algorithm. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 357–370. Springer, 2004.

[18] Michèle Feltz and Cas Cremers. On the limits of authenticated key exchange security with an application to bad randomness. Cryptology ePrint Archive, Report 2014/369, 2014.

[19] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 163–178. IEEE Computer Society, 2014.

[20] Ralf Küsters and Tomasz Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *Computer Security Foundations Symposium (CSF)*, pages 157–171. IEEE, 2009.

[21] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In *Provable Security*, pages 1–16. Springer, 2007.

[22] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *CAV*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.

[23] Sonia Santiago, Santiago Escobar, Catherine Meadows, and José Meseguer. A formal definition of protocol indistinguishability and its verification using Maude-NPA. In *Security and Trust Management (STM) 2014*, pages 162–177. Springer, 2014.

[24] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF)*, pages 78–94. IEEE, 2012.

[25] Alwen Tiu and Jeremy E. Dawson. Automating open bisimulation checking for the spi calculus. In *CSF*, pages 307–321. IEEE Computer Society, 2010.

[26] Ton Van Deursen, Sjouke Mauw, and Saša Radomirović. Untraceability of RFID protocols. In *Information Security Theory and Practices. Smart Devices, Convergence and Next Generation Networks*, pages 1–15. Springer, 2008.