Abstraction by Set-Membership

Verifying Security Protocols and Web Services with Databases

Sebastian A. Mödersheim[*]
Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kongens Lyngby
samo@imm.dtu.dk

ABSTRACT

The abstraction and over-approximation of protocols and web services by a set of Horn clauses is a very successful method in practice. It has however limitations for protocols and web services that are based on databases of keys, contracts, or even access rights, where revocation is possible, so that the set of true facts does not monotonically grow with state transitions. We extend the scope of these over-approximation methods by defining a new way of abstraction that can handle such databases, and we formally prove that the abstraction is sound. We realize a translator from a convenient specification language to standard Horn clauses and use the verifier ProVerif and the theorem prover SPASS to solve them. We show by a number of examples that this approach is practically feasible for wide variety of verification problems of security protocols and web services.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol verification; D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Verification

Keywords

Automated verification, abstract interpretation, revocation, web services, APIs

1. INTRODUCTION

Tools based on over-approximation like ProVerif have been very successful on the verification of security protocols and web services [7, 11, 10, 21, 6]. In contrast to conventional model checking approaches like [15, 4, 1], the over-approximation methods do not consider a state transition system, but just a set of derivable (state-independent) facts like intruder knowledge (and the intruder never forgets). Moreover, the creation of fresh keys and nonces is replaced by a function of the context in which they are created. For instance if agent a creates a nonce for use with agent b, this may simply be $n(a, b)$ in every run of the protocol. The main advantage is that this kind of verification works for an unbounded number of sessions, while standard model checking methods consider a bounded number of sessions. In fact, the entire interleaving problem of model checking does not occur in the over-approximation approach, and tools thus also scale better with the number of protocol steps and repeated parts of the protocol. Another advantage is that models of this kind can be represented as a set of first-order Horn clauses for which many existing methods can be used off the shelf, e.g. the SPASS theorem prover [21, 22].

A disadvantage of the abstractions are false positives, i.e. attacks that are introduced by the over-approximation. In the worst case we may thus fail to verify a correct protocol. This problem can sometimes be solved by refining the abstraction. However, if we turn to more complex systems that consist of several protocols or web services, the abstraction approaches reach a limitation. The reason is that we may consider servers that maintain some form of database, for instance a key-server maintains a set of keys, to which agents they belong and what their status is, e.g. valid or revoked/outdated. Another example is a web service for online shopping that maintains a database of orders that have been processed and their current status. Further, servers may maintain a database of access rights and access rights may be revoked. Common between these examples is that the set of true facts does not monotonically grow with the executions of the protocols. Such non-monotonic behavior simply cannot be expressed in the standard (stateless) approach of abstracting protocols by a set of Horn clauses, because deduction is monotonic, i.e. adding facts like a revocation can never lead to fewer deductions.

This work tackles this problem with a different kind of abstraction of the fresh data while maintaining the basic approach of over-approximating the protocol or web service by a set of first-order Horn clauses. As a basis, we consider a model where each participant can maintain a database in which freshly generated data like nonces, keys, or order-numbers can be stored along with their context, e.g. the owner and status of a key. To deal with such systems in an

[*]The author was partially supported by the EU-Project AVANTSSAR. This paper was written while the author was working for IBM Research Zurich. The author thanks Luca Viganò, Hanne Riis Nielson, Flemming Nielson, Graham Steel and the anonymous reviewers for helpful comments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'10, October 4–8, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0244-9/10/10 ...$10.00.

abstraction approach, we define the abstraction of all created data by their status and membership in the databases of the participants. For instance, suppose there are two agents a and b which each maintain a set of keys that are either unknown, valid, or revoked; two concrete keys k_1 and k_2 are now mapped to the same abstract key k iff they are equal in the membership of the databases, e.g. a considers both keys as revoked, and b considers both as valid.

So, as usual in these approaches, the infinite set of data is mapped to finitely many equivalence classes or representatives (if we have finitely many participants), but here the abstraction depends on the current state of the databases. Consider for example that the intruder knows a message m containing, as a subterm, the abstract key k mentioned above. Consider further a transition rule that allows one to revoke a key at agent b, so that in the abstract model, the key k should be "transformed" into a key k' representing all the keys that are known as revoked to both a and b. The idea to handle this in the abstraction is to maintain all previous facts that contain the key k in its old form and to add also all these facts with k' replaced for k. So everything the intruder knows with a valid key k (in b's eyes), he also knows with a revoked key k'. The intuitive reason why this is indeed sound is that—thanks to the over-approximation—every derivation in the abstract model corresponds to an unlimited number of executions with concrete data that fall into the same equivalence class.

The transformation of facts that arises from the state-transition of the database is expressed by a new kind of rule, so-called *term implication rules* that have the form $\phi \to k \twoheadrightarrow k'$. This expresses that, if the clauses in ϕ hold, then $f[k]$ implies $f[k']$ for every context $f[\cdot]$. We show that these rules can be encoded into standard Horn clauses.

Our contributions are both theoretical and practical. First, we define the specification language AIF, a variant of the AVISPA Intermediate Format [3] that allows for a declarative specification of the un-abstracted transition system with fresh data and databases. Second, we define a novel way to abstract this specification into a set of Horn clauses and term implication rules, a concept that naturally arises from this kind of specification. Third, we show that this abstraction is sound, i.e. without excluding attacks. Fourth we show how to encode also the term implication rules as Horn clauses without excluding or introducing attacks. Fifth, we implement this translation from AIF to Horn clauses for the syntax of the tools SPASS and ProVerif, both of which implement state of the art resolution techniques for first-order (Horn) clauses. This allows us to demonstrate with a number of non-trivial examples that the approach is practically feasible. The implementation and a library of AIF examples with detailed descriptions is available [17].

2. AIF AND THE CONCRETE MODEL

This section introduces the language AIF that we use for specifying security protocols, web services, and their goals without the abstraction. It is a variant of the AVISPA Intermediate Format[3] influenced by the needs of our methods and adding syntactic sugar for convenience.

2.1 A Running Example

Before we give the formal definition, we first introduce a simple example that we use throughout this paper. For simplicity, we limit the example to three agents: the honest user a, the honest server s, and the dishonest intruder i. The full specification considered in section 7 is parametrized and can be used with any (but fixed) number of honest and dishonest users (see also [17]).

Each agent has a database of its own that contains all the information that this agent has to maintain over a longer time (i.e., that may span several sessions). In our example, the user keeps a database of all its valid public/private key pairs that it currently has registered with the server s. We denote with $\mathsf{inv}(k)$ the private key that belongs to public key k. Thus, all entries of a's database are of the form $(k, \mathsf{inv}(k))$ and it is sufficient to represent the database entries only by the public key k (omitting $inv(k)$ in the term representation of the database). We thus write the set condition $k \in ring(a)$ for every key k in the database of a.

The server stores in its database the registered keys along with their owner and status, which is either *valid* or *revoked*. One could write for instance $(k, a, valid) \in db(s)$ for a key k that is stored in the database of s as a valid key owned by a, but we rather use a slightly different representation and write $k \in db(s, a, valid)$. This is equivalent to thinking of a server that maintains for each user two databases, namely the sets of valid and revoked keys. This representation is helpful for the abstraction below because all sets contain only data that can be abstracted (public keys in this example) rather than a mixture of different kinds of data.

An AIF specification describes a state transition system by a set of rules. The first rule of our example is an initialization rule that represents an out of band registration of the key with a server. (Suppose the user physically visits the organization that owns the server.)

$$=[PK]\Rightarrow PK \in ring(a) \cdot PK \in db(s, a, valid) \cdot \mathsf{iknows}(PK)$$

This rule can be taken in any state (because there are no conditions left of the arrow) and will first create a fresh value (that never occurred before) that we bind to the variable PK, intuitively a public key. In the successor state, PK is both in the databases of a and of the server as a valid key. We use $\mathsf{iknows}(m)$ to denote that the intruder knows m, so in this case he learns immediately the new public key PK. The rule can be applied any number of times to register as many keys as desired. Note that $\mathsf{iknows}(\cdot)$ does *not* have a predefined meaning in AIF, is rather characterized by intruder deduction rules reflecting the standard Dolev-Yao model, e.g. $\mathsf{iknows}(M).\mathsf{iknows}(\mathsf{inv}(K)) \Rightarrow \mathsf{sign}_{\mathsf{inv}(K)}(M)$ (which can be applied to any state that contains facts matching what is left of $\Rightarrow$).

The second rule of our example is the transmission of a new key using a registered valid key:

$$PK \in ring(a) \cdot \mathsf{iknows}(PK)$$
$$=[NPK]\Rightarrow NPK \in ring(a) \cdot \mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(PK)}(new, a, NPK))$$

We do not repeat the condition $PK \in ring(a)$ on the RHS; in AIF this means that this condition gets removed by the transition, i.e. the user a forgets the key PK (which is a bit unrealistic and only done for the sake of simplicity).

The third rule is the server receiving such a message, registering the new key and revoking the old key:

$$\mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(PK)}(new, a, NPK)) \cdot PK \in db(s, a, valid) \cdot$$
$$NPK \notin db(s, a, valid) \cdot NPK \notin db(s, a, revoked)$$
$$\Rightarrow PK \in db(s, a, revoked) \cdot NPK \in db(s, a, valid)$$
$$\cdot \mathsf{iknows}(\mathsf{inv}(PK))$$

Here, the intruder learns the private key of the revoked key.

To define a security goal, we give yet a further rule that produces the fact *attack* if the intruder finds out the private key of a valid public key of a:

$$\mathsf{iknows}(\mathsf{inv}(PK)) \cdot PK \in db(s, a, valid) \Rightarrow attack$$

2.2 Formal Definition of AIF

We use a standard term model of messages, the only specialty is the distinction of constants and variables that will be abstracted later.

Definition 1. Messages are represented as terms over a signature $\Sigma \cup \mathfrak{A}$ and a set $\mathcal{V}$ of variables, where Σ is finite, $\mathcal{V}$ is countable, and $\mathfrak{A}$ is a countable set of constant symbols (namely those that are going to be abstracted later). Σ, $\mathfrak{A}$, and $\mathcal{V}$ are non-empty and pairwise disjoint. Let $\mathcal{V}_{\mathfrak{A}} \subset \mathcal{V}$ be a set of variables that can only be substituted by constants of $\mathfrak{A}$. Let $\mathcal{T}_{\mathfrak{A}} = \mathfrak{A} \cup \mathcal{V}_{\mathfrak{A}}$ denote the set of all *abstractable* symbols. By convention, we use upper-case letters for variables and lower-case letters for constant and function symbols.

Note that this paper will assume a free algebra interpretation of terms (i.e. two terms are equal iff they are syntactically equal). We come back later to this issue when we use SPASS (which does not consider a fixed interpretation).

Definition 2. Let Σ_f be a finite signature (disjoint from all sets above) of *fact symbols*. A *fact* is a term of the form $f(t_1, \ldots, t_n)$ where f is a fact symbol of arity n and the t_i are messages. A *positive (negative) set condition* has the form $t \in M$ ($t \notin M$) where $t \in \mathcal{T}_{\mathfrak{A}}$ and M is a *set expression*, namely a ground message term in which no symbol of $\mathcal{T}_{\mathfrak{A}}$ occurs.

The syntactic form of set expressions like M in this definition enforces that a specification can only use a fixed number of sets that we denote with N. Also, in all formal arguments in this paper we will thus simply assume these sets are called $s_1, \ldots, s_N$, while in AIF specifications, one we will use more intuitive terms like $ring(a)$ for the set of keys known by agent a.

We now come to the core of the AIF specifications, namely the state transition rules.

Definition 3. A *state* is a finite set of facts and positive set conditions. A *transition rule* r has the form

$$LF \cdot S_+ \cdot S_- =\!\![F]\!\!\Rightarrow RF \cdot RS$$

where LF and RF are sets of facts, S_+ and RS are sets of positive set conditions, S_- is a set of negative set conditions, and $F \subseteq \mathcal{V}_{\mathfrak{A}}$. We require that

$$vars(RF \cdot RS \cdot S_-) \subseteq F \cup vars(LF \cdot S_+) \text{ and } vars(S_-) \cap F = \emptyset \,.$$

Moreover, we require that each $t \in \mathcal{T}_{\mathfrak{A}}$ that occurs in S_+ or S_- also occurs in LF and each $t \in \mathcal{T}_{\mathfrak{A}}$ that occurs in RS also occurs in RF.[1]

We say $S \Rightarrow_r S'$ iff there is a grounding substitution σ (for all variables of r) such that

- $(LF \cdot S_+)\sigma \subseteq S$,
- $S_-\sigma \cap S = \emptyset$, $S' = (S \setminus S_+\sigma) \cup RF\sigma \cup RS\sigma$,
- $F\sigma$ are fresh constants from $\mathfrak{A}$ (i.e. they do not occur in S or any rule r that we consider).

[1]This condition ensures that when we remove set conditions in rules and states in the abstract model below, the elements (that will carry the set conditions in their abstraction) still appear in the normal facts.

A state S is called *reachable* using the set of transition rules R, iff $\emptyset \Rightarrow_R^* S$. Here $\Rightarrow_R$ is the union of $\Rightarrow_r$ for all $r \in R$ and $\cdot^*$ is the reflexive transitive closure. (We generally use the $\emptyset$ as the initial state.)

Intuitively, the left-hand side of a rule describes to which states the rule can be applied, and the right-hand side describes the changes to the state after the transition.

There is a subtle difference to AVISPA IF and other set-rewriting/multi-set rewriting approaches. In AIF, facts are *persistent*, i.e. a fact that holds in one state also holds in all successor states. The only entities that can be removed from a state during a transitions are the positive set conditions, namely by a transition rule that has a positive condition $x \in s_i$ on the left-hand side that is not repeated on the right-hand side.

The persistence of facts is a restriction with respect to other approaches, but one that comes without loss of generality: a non-persistent fact $f(t_1, \ldots, t_n)$ of AVISPA IF can be simulated in AIF by a persistent fact $f'(t_1, \ldots, t_n, FID)$, where FID is a fresh identifier created when introducing the fact, and using a distinguished set *valid* that contains FID in exactly those states where $f(t_1, \ldots, t_n)$ holds.

Our construction to make set membership the only "revocable" entity while facts monotonously grow over transitions gives a distinction that becomes valuable in the abstraction later. To see that, consider that the AIF transition rules (or the AVISPA IF transition rules) are not monotonic (i.e. a rule that is applicable to a state S is not necessarily applicable to any superset of S). In contrast, the Horn-clauses of the abstract model are interpreted in standard–monotone–first-order logic. Our construction thus ensures that all the non-monotonic aspects, the set memberships, are part of the abstraction.

We close this discussion with the remark that all previous abstraction approaches in protocol verification like [7, 11, 10, 21] are entirely based on persistent facts. This (usually) means an over-approximation that leads to the following phenomenon [16]: every participant can react to a given message any number of times, even if the real system prevents that with challenge-response or timestamps. As can be seen by the success of the abstraction methods, this over-approximation usually works fine (if one does not consider replay which requires special care [8]). So in general, for what concerns this new abstraction approach where we have the choice to make things revocable, one may start with a model where all facts are persistent and perform the above encoding of non-persistent facts only when necessary, i.e. when one obtains false attacks caused by the over-approximation.

2.3 Syntactic Sugar

For readability and brevity of specifications, the AIF language supports a number of constructs to avoid finite enumerations. One can declare a number of variables that range over a given set of constants, e.g.:

$$\begin{array}{rcl} A, B & : & \{a, b, s, i\}; \\ Honest & : & \{a, b\}; \\ Status & : & \{valid, revoked\}; \end{array}$$

We call variables that have been declared in this way *enumeration variables*. An AIF specification includes the enumeration of all sets or databases that occur in the specification. Here, the enumeration variables can be used. For example:

$$\mathsf{Sets} : \mathit{ring}(\mathit{Honest}),\ \mathit{db}(s, A, \mathit{Status});$$

defines that every honest agent Honest has its own keyring $\mathit{ring}(\mathit{Honest})$, which may be for instance a set of public keys, and the server s has a database for each agent A and each Status, each of which may again be a set of public-keys. Thus, this example specification uses $N = 10$ sets.

One can further use the enumeration variables as abbreviations in rules. First, we may use universal quantification of enumeration variables in negative set conditions, e.g.

$$\forall A, \mathit{Status}. \mathit{PK} \notin \mathit{db}(s, A, \mathit{Status})$$

to mean that PK cannot occur in any of the sets covered by expanding all values of the enumeration variables, so this example expands to 8 negative set conditions.

Second, we can parametrize an entire rule over enumeration variables. We may write for instance $\lambda A. \Rightarrow \mathsf{iknows}(A)$ to denote that the intruder knows every agent name. We write λ to avoid confusion with quantification: in fact, the meaning of $\lambda X.r$ is the set of rules $\{r[X \mapsto v] \mid v \in V\}$ where V is the enumeration declared for X.

With this syntactic sugar, it is easy to generalize our example specification for any number of honest and dishonest users and servers, namely by replacing the constants by enumeration variables and enumerating the desired set of agents there [17]. The "unrolling" of this sugar is not always efficient and we plan as future work to investigate strategies for avoiding that in the translation.

2.4 Inconsistent Rules

We exclude rules that are "inconsistent" in a certain sense (although their semantics is well-defined):

Definition 4. A rule $r = LF \cdot S_+ \cdot S_- \,{=}[F]{\Rightarrow}\, RF \cdot RS$ is called *inconsistent*, if any of the following holds:

- $t \in M$ occurs in S_+ and $t \notin M$ occurs in S_-, or
- $s \in M$ occurs in $S_+ \setminus RF$ and $t \in M$ occurs in RF, and the rule allows for an instantiation σ with $s\sigma = t\sigma$.

For the rest of this paper, we consider only consistent rules.

The first kind of inconsistent rule is simply never applicable. For the second kind, we get the contradiction only under a particular instantiation, namely when $s\sigma = t\sigma$, because the rule says that the constraint $s\sigma \in M$ should be removed and $t\sigma \in M$ should be added or kept. (The semantics tells us that here the positive constraint to keep $t\sigma \in M$ wins.)

Note that all rules of our running example are consistent; for instance in the second rule, the instantiation $\mathit{PK}\sigma = \mathit{NPK}\sigma$ is not possible because NPK is fresh, and in the third rule such a substitution is also ruled out by the left-hand side constraints $\mathit{PK} \in \mathit{db}(s, a, \mathit{valid})$ and $\mathit{NPK} \notin \mathit{db}(s, a, \mathit{valid})$. In fact, the notion that a rule allows for the instantiation $s\sigma = t\sigma$ is purely syntactical (i.e. independent of the actually reachable states).

There are two reasons to exclude inconsistent rules. First, they often result from a specification mistake, i.e. they do not reflect what the user actually wanted to model. Second, the soundness proof of our abstractions below is more complex when allowing the second kind of inconsistent rules.

3. SET-BASED ABSTRACTION

The core idea of set-based abstraction is the following: we abstract the fresh data according to its membership in the used sets. For instance, if we have three sets s_1, s_2, and s_3, we may abstract all elements that are contained in s_1 but not in s_2 and s_3 into one equivalence class denoted $\mathit{val}(1, 0, 0)$.

In our running example, we have the sets $s_1 = \mathit{ring}(a)$, $s_2 = \mathit{db}(s, a, \mathit{valid})$, and $s_3 = \mathit{db}(s, a, \mathit{revoked})$. Thus let $\mathit{val}(1, 0, 0)$ represent the class of all public keys that the user a has created but that are not (yet) registered with the server s as valid or revoked. The abstract model thus does not distinguish between several different keys that have the same status in terms of set-membership.

The standard way to express the abstract model by Horn clauses in previous approaches does not work with this abstraction. In particular, when the set membership of a constant changes from the abstract value a to the abstract value a', then for every derivable fact $f[a]$ that contains a also $f[a']$ is derivable. This requires an extension with a new kind of rule that can exactly express $f[a] \implies f[a']$ for every context $f[\cdot]$ and which we formalize below. Note that this kind of rule is different from an algebraic equation like $a \approx a'$, because $f[a']$ does not necessarily imply $f[a]$; moreover, it is different from a rewrite rule, because $f[a]$ is not *replaced* by $f[a']$ but both $f[a]$ and $f[a']$ hold.

3.1 Definition of the Abstraction

Definition 5. Consider a set of rules that uses the ground terms $s_1, \ldots, s_N$ in set conditions $t \in s_i$ and $t \notin s_i$ (including the choice of a total order on the s_i). For a state S, we define the function abs_S that maps from $\mathfrak{A}$ to $\mathit{val}(\mathbb{B}^n)$ as follows: $\mathit{abs}_S(c) = \mathit{val}(b_1, \ldots, b_N)$ with b_i true iff $(c \in s_i) \in S$. This induces an equivalence relation (parametrized by a state S) on $\mathfrak{A}$: define $c \equiv_S c'$ iff $\mathit{abs}_S(c) = \mathit{abs}_S(c')$.

It is indeed unusual that an abstract interpretation depends on states and can change from state to state. This reflects exactly why the databases we want to model do not exactly fit into the standard abstraction approach of protocol verification: the abstract model does not have a notion of states any more. We will see below (in section 4) how to overcome this problem and define a state-independent abstraction function.

3.2 Term Implication Rules

We now introduce the form of rule that allows us to deal with abstractions with the changing set-membership of constants.

Definition 6. A *term implication rule* has the form

$$\frac{P_1 \quad \ldots \quad P_n}{s \twoheadrightarrow t}$$

where the P_i are predicates (i.e. facts) and $\mathit{vars}(t) \cup \mathit{vars}(s) \subseteq \bigcup_{i=1}^{n} \mathit{vars}(P_i)$. An *implication rule* is either a term implication rule or a Horn clause. We often write $A \to C$ instead of $\dfrac{A}{C}$. We may also write $A \to C_1 \cdot \ldots \cdot C_n$ as an abbreviation for the set of rules $\{\, A \to C_i \mid 1 \le i \le n \,\}$.

For implication rules, we define a function that, given a set Γ of facts, yields all facts that can be derived from Γ by one rule application:

$$\begin{aligned}\left[\!\left[\frac{\phi_1 \quad \dots \quad \phi_n}{\phi}\right]\!\right](\Gamma) &= \{\phi\sigma \mid \phi_1\sigma \in \Gamma \wedge \dots \wedge \phi_n\sigma \in \Gamma\}\\ \left[\!\left[\frac{\phi_1 \quad \dots \quad \phi_n}{s \twoheadrightarrow t}\right]\!\right](\Gamma) &= \{C[t\sigma] \mid \ C[s\sigma] \in \Gamma \wedge \phi_1\sigma \in \Gamma \\ &\qquad \wedge \dots \wedge \phi_n\sigma \in \Gamma \quad \}\end{aligned}$$

Here, $C[\cdot]$ is a *context*, i.e. a "term with a hole", and $C[t]$ means filling the hole with term t. The *least fixed-point* of a set of implication rules R, denoted $LFP(R)$ is defined as the least set Γ that is closed under $[\![r]\!]$ for each $r \in R$.

3.3 Translation to Abstract Rules

We now translate the standard transition rules (that work on the real sets) to implication rules of an abstract model (that work on the abstract encoding of set membership). We show in section 4 that this abstraction is sound.

Definition 7. Consider a transition rule

$$r = LF \cdot S_+ \cdot S_- =\!\![F]\!\!\Rightarrow RF \cdot RS$$

Let $\mathcal{T}_{\mathfrak{A}}(r)$ be the symbols from $\mathcal{T}_{\mathfrak{A}}$ that occur in r. We define for each $t \in \mathcal{T}_{\mathfrak{A}}(r)$ and for each $1 \le i \le N$:

$$L_i(t) = \begin{cases} 1 & \text{if } t \in s_i \text{ occurs in } S_+ \\ 0 & \text{if } t \notin s_i \text{ occurs in } S_- \\ X_{t,i} & \text{otherwise} \end{cases}$$

$$R_i(t) = \begin{cases} 1 & \text{if } t \in s_i \text{ occurs in } RS \\ X_{t,i} & \text{otherwise, if } L_i(t) = X_{t,i} \text{ and } t \notin F \\ 0 & \text{otherwise} \end{cases}$$

Here, let $X_{t,i} :: \mathbb{B}$ be variables that do not occur in r. Let

$$\begin{aligned} L(t) &= val(L_1(t), \dots, L_N(t)) \\ R(t) &= val(R_1(t), \dots, R_N(t)) \ . \end{aligned}$$

The *abstraction* $\overline{r}$ *of the rule* r is defined as:

$$\overline{r} = LF\lambda \to RF\rho \cdot C$$

for the following substitutions λ and ρ and term implications C:

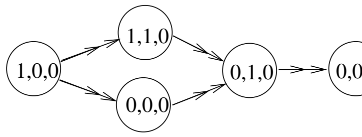
- $\lambda = [t \mapsto L(t) \mid t \in \mathcal{T}_{\mathfrak{A}}(r)]$
- $\rho = [t \mapsto R(t) \mid t \in \mathcal{T}_{\mathfrak{A}}(r)]$
- $C = \{t\lambda \twoheadrightarrow t\rho \mid t \in \mathcal{T}_{\mathfrak{A}}(r) \setminus F\}$

3.4 The Example

Figure 1 shows the translation of our running example. Thanks to the abstraction, it is straightforward to convince oneself that *attack* is unreachable, as this requires the fact $\mathsf{iknows}(\mathsf{inv}(val(X_1, 1, X_2)))$ (i.e. a valid key) whereas the only rule that gives the intruder a private key has the incompatible set membership $(X_1, 0, 1)$ (i.e. a revoked key) and there is no term implication rule that could turn a revoked key into a valid one. Let

$$\begin{aligned} SK &= \{val(0,0,0), val(0,1,0), val(0,0,1)\} \\ K &= SK \cup \{val(1,0,0), val(1,1,0)\} \end{aligned}$$

K is the set of all public keys that occur in some fact. (The other three bearable keys $val(0,1,1)$ and $val(1,1,1)$



$$\begin{tikzcd} & {1,1,0} \arrow[dr, two heads] & & \\ {1,0,0} \arrow[ur, two heads] \arrow[dr, two heads] & & {0,1,0} \arrow[r, two heads] & {0,0,1} \\ & {0,0,0} \arrow[ur, two heads] & & \end{tikzcd}$$

Figure 2: The key life-cycle as formalized by the term implications.

and $val(1,0,1)$ do never occur.) The subset SK contains those keys that can ever occur as the signing key in a signature.

The fixed-point is $\Gamma = \{\mathsf{iknows}(m) \mid m \in M\}$ where

$$\begin{aligned} M = \ & \mathcal{DY}(K \cup \{\mathsf{sign}_{\mathsf{inv}(sk)}(a, new, k) \mid sk \in SK, k \in K\} \\ & \cup\{\mathsf{inv}(val(0,0,1))\}) \end{aligned}$$

and $\mathcal{DY}(\cdot)$ denotes the closure under protocol-independent intruder deduction rules (like encryption). In particular, only the private keys of revoked, invalid keys get known to the intruder, and *attack* is not in Γ.

We note that the concrete term implications $s \twoheadrightarrow t$ which get activated in Γ, displayed in Figure 2, represent exactly the life-cycle of keys.

4. SOUNDNESS

For verification, the crucial property of our abstraction is that if the concrete model has an attack, then so has the abstract model. If this holds, then verification of the abstract model implies verification of the concrete model.We take a detour over some intermediate models which greatly simplifies the actual proof of correctness.

The labeled concrete model.

The first idea is to *label* all symbols of $\mathcal{T}_{\mathfrak{A}}$ in the concrete model with the corresponding abstract terms according to Definition 5. Being merely an annotation, this does not change the model.

Definition 8. The *labeled concrete model* is defined as the following modification rules of the concrete model: every $t \in \mathcal{T}_{\mathfrak{A}}$ on the LHS (RHS) of a rule is labeled with $L(t)$ ($R(T)$) (cf. Definition 7). We denote the labeling of term t with label l by $t@l$. Moreover, for each $t \in \mathcal{T}_{\mathfrak{A}}$ that occurs on both sides, we add the *label modification* $t@L(t) \mapsto t@R(t)$. This label modification is applied as a replacement on the successor state: let $r' = r \cdot (t@l \mapsto t@l')$ the augmentation of r with the label modification. We then define r' transitions based on r transitions as follows: if $S \Rightarrow_r S'$ under match σ then $S \Rightarrow_{r'} S'\tau$ where τ is the replacement of all occurrences of $t\sigma$ (for any label) with $t\sigma$ labeled by l'.

As an example, the second rule of our running example looks as follows in the labeled model:

$$\begin{aligned} & \mathsf{iknows}(PK@(1,X_1,X_2)) \cdot PK@(1,X_1,X_2) \in ring(a) \\ & =\!\![NPK@(1,0,0)]\!\!\Rightarrow \\ & NPK@(1,0,0) \in ring(a) \\ & \mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(PK@(0,X_1,X_2))}(new, a, NPK@(1,0,0))) \cdot \\ & PK@(1,X_1,X_2) \mapsto PK@(0,X_1,X_2) \end{aligned}$$

Lemma 1. *In the labeled model, in every state S, every occurrence of an abstractable constant c is labeled with $l = (b_1, \dots, b_N)$ such that b_i is true iff the set condition $c \in s_i$ is contained in S.*

$=\!\![PK]\!\!\Rightarrow$ $\mathsf{iknows}(PK) \cdot PK \in \mathit{ring}(a) \cdot PK \in \mathit{db}(s,a,\mathit{valid})$	$\rightarrow$ $\mathsf{iknows}(\mathit{val}(1,1,0))$
$\mathsf{iknows}(PK) \cdot PK \in \mathit{ring}(a)$ $=\!\![NPK]\!\!\Rightarrow$ $NPK \in \mathit{ring}(a) \cdot$ $\mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(PK)}(\mathit{new},a,NPK))$	$\mathsf{iknows}(\mathit{val}(1,X_1,X_2))$ $\rightarrow$ $\mathit{val}(1,X_1,X_2) \twoheadrightarrow \mathit{val}(0,X_1,X_2) \cdot$ $\mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(\mathit{val}(0,X_1,X_2))}(\mathit{new},a,\mathit{val}(1,0,0)))$
$\mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(PK)}(\mathit{new},a,NPK)) \cdot$ $PK \in \mathit{db}(s,a,\mathit{valid}) \cdot NPK \notin \mathit{db}(s,a,\mathit{valid}) \cdot NPK \notin \mathit{db}(s,a,\mathit{revoked})$ $\Rightarrow$ $PK \in \mathit{db}(s,a,\mathit{revoked}) \cdot$ $NPK \in \mathit{db}(s,a,\mathit{valid}) \cdot$ $\mathsf{iknows}(\mathsf{inv}(PK))$	$\mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(\mathit{val}(X_1,1,X_2))}(\mathit{new},a,\mathit{val}(X_3,0,0)))$ $\rightarrow$ $\mathit{val}(X_1,1,X_2) \twoheadrightarrow \mathit{val}(X_1,0,1) \cdot$ $\mathit{val}(X_3,0,0) \twoheadrightarrow \mathit{val}(X_3,1,0) \cdot$ $\mathsf{iknows}(\mathsf{inv}(\mathit{val}(X_1,0,1)))$
$\mathsf{iknows}(\mathsf{inv}(PK)) \cdot PK \in \mathit{db}(s,a,\mathit{valid})$ $\Rightarrow$ attack	$\mathsf{iknows}(\mathsf{inv}(\mathit{val}(X_1,1,X_2)))$ $\rightarrow$ attack

Figure 1: The transition rules of the running example (LHS) and their abstraction (RHS).

Proof. We show this by induction over reachability. It trivially holds for the initial state. For transitions, suppose the property holds in state S, and $S \rightarrow_{r'} S'$ for some labeled rule r' and let σ be the rule match. Consider any $c \in \mathit{Abs}$ that occurs in S' with label $(b_1, \ldots, b_N)$. We show for every $1 \leq i \leq N$: b_i is true iff $c \in s_I$ occurs in S'. We distinguish the following cases:

- c does not occur in S, so it was freshly created by the transition to S'. Thus there is a variable X in the fresh variables of r' such that $X\sigma = c$. By definition, X (and thus every occurrence of c in S') is labeled with $b_i = R_i(X)$ which is true iff $X_i \in s_i$ is contained in the right-hand side of r', which is the case iff $c \in s_i$ is contained in S'.

- c occurs in S, and for no abstractable variable X in r' it holds that $c = X\sigma$. Then c is simply not touched by the transition and has the same label and set memberships in both states.

- c occurs in S, and for some abstractable variable X in r', we have $X\sigma = c$. (Note there may be other variables Y with $Y\sigma = c$.) We further distinguish:
  - $X \in s_i$ occurs in S_+. Then $c \in s_i$ occurs in S and there is no variable Y such that both $Y\sigma = c$ and $Y \notin s_i$ occurs in S_- (otherwise r' would not have been applicable to S under σ). If for any variable Y with $Y\sigma = c$, $Y \in s_i$ occurs in RS, then also $X \in s_i$ must occur in RS otherwise the rule is not consistent (cf. Definition 4). Thus $R_i(X)$ and b_i is true and $c \in s_i$ is contained in S'. Otherwise, if for no variable Y with $Y\sigma = c$, $Y \in s_i$ is contained in RS, then by the definition there is a label change for X in r', namely changing at least the ith position from true to false. Then $c \in s_i$ is not in S' and b_i is false.
  - $X \notin s_i$ occurs in S_-. Then $c \in s_i$ does not occur in S, and there is no variable Y with $Y\sigma = c$ and $Y \in s_i$ in S_+. Suppose for any Y with $Y\sigma = c$, $Y \in s_i$ occurs in RS, then also $X \in s_i$ in RS (otherwise the rule were again inconsistent). Thus there is a label change in the ith position from false to true and b_i is true and $c \in s_i$ is contained in S'. Otherwise, X is labeled on both sides with false for the ith component, and b_i is false, and $c \in s_i$ does not occur in S'.
  - Neither $X \in s_i$ occurs in S_+ nor $X \notin s_i$ occurs in S_-. If $X \in s_i$ occurs in the RS, then we have a label change in the ith position of the labeling of X, namely from arbitrary X_i to 1. Thus b_i is 1 and $c \in s_i$ occurs in S'. Otherwise, if $X \in s_i$ does not occur in RS, then X is not involved in any set conditions. Then either c stays with the same label and set membership in the transition from S to S', or there is another variable Y with $Y\sigma = c$ and any of the above cases can be applied with Y in the role of X.

- c occurs in S but there is no abstractable variable X in r' such that $X\sigma = c$. Then there is no change of set memberships of c and no label change and the property remains that the labeling is correct.

□

Labeled concrete model without set conditions.

The labels are thus a correct alternative representation of the set conditions, and as a second step we now "upgrade" the labels from a mere annotation to a part of term structure, i.e. considering @ as a binary (infix) function symbol. Then, upon rule matching the label does matter. In turn, we can remove the set conditions from our model completely, because we can always reconstruct the set memberships from the labels (thanks to persistence and rule form, no abstractable constants can get lost on a transition) and the set conditions on the left-hand side of a rule are correctly handled by the label matching. In this *labeled model without set conditions*, the second rule of our running example is:

$$\begin{array}{l}\mathsf{iknows}(PK@(1,X_1,X_2))\\ =\!\![NPK@(1,0,0)]\!\!\Rightarrow\\ \mathsf{iknows}(\mathsf{sign}_{\mathsf{inv}(PK@(0,X_1,X_2))}(\mathit{new},a,NPK@(1,0,0))) \cdot\\ PK@(1,X_1,X_2) \mapsto PK@(0,X_1,X_2)\end{array}$$

Note how close this rule is to the abstract model, while still being a state transition rule. It is immediate from Lemma 1 that this changes the model only in terms of representation:

LEMMA 2. *The labeled model and the labeled model without set conditions have the same set of reachable states modulo the representation of set conditions in labels.*

The abstraction.

All the previous steps were only changing the representation of the model, but besides that the models are all equivalent. Now we finally come to the actual abstraction step that transforms the model into an abstract over-approximation.

We define a representation function η that maps terms and facts of the concrete model to ones of the abstract model:

Definition 9.

$$\begin{array}{rcl} \eta(t@(b_1,\ldots,b_N)) & = & val(b_1,\ldots,b_N) \text{ for } t \in \mathcal{T}_{\mathfrak{A}} \\ \eta(f(t_1,\ldots,t_n)) & = & f(\eta(t_1),\ldots,\eta(t_n)) \\ & & \text{for any function or fact symbol } f \text{ of arity } n \end{array}$$

We show that the abstract rules allow for the derivation of the abstract representation of every reachable fact f of the concrete model:

LEMMA 3. *Let R be a rule set in AIF, R' be the corresponding rule set in the labeled model without set conditions of R, f be a fact in a reachable state of R' (i.e. $\emptyset \to^*_{R'} S$ and $f \in S$ for some S). Let $\overline{R}$ be the translation into Horn clauses of the rules R according to Definition 7, and $\Gamma = LFP(\overline{R})$. Then $\eta(f) \in \Gamma$.*

PROOF. Again we show this by induction over reachability. The initial state $\emptyset$ is clear. Let now S be any reachable state and $\eta(f) \in \Gamma$ for every $f \in S$. We show that for every S' that is reached by one rule application and every $f \in S'$ also $\eta(f) \in \Gamma$.

Let the considered rule be

$$r = LF \ {=}[F]{\Rightarrow}\ RF \cdot LM$$

where LM are the label modifications (see Definition 8)—being part of the labeled model without set conditions there are no set conditions in the rule. By our constructions, the Horn clauses $\overline{R}$ contain a similar rule, namely

$$\overline{r} = \eta(LF) \to \eta(RF) \cdot \eta(LM)$$

where we extend η to sets of facts as expected. The extension of η to label modifications (and sets thereof in $\eta(LM)$) is also straightforward:

$$\eta(t@l \mapsto t@l') = val(l) \twoheadrightarrow val(l')$$

Let now σ be the corresponding substitution for $S \to_r S'$. Then $LF\sigma \subseteq S$ and thus $\eta(LF\sigma) \subseteq \eta(S)$. Thus the Horn clause $\overline{r}$ is applicable and therefore $\eta(RF\sigma) \subseteq \Gamma$. It remains only to show that all the modifications of facts by the label modification rule are also contained in Γ.

To that end, consider any fact $f[c@l] \in (S \cup RF)\sigma$ that has exactly one occurrence of $c@l$ and LM contains the rule $t@l \mapsto t@l'$ for some t with $t\sigma = c$. Since $l \twoheadrightarrow l'$ is part of the term implication of $\overline{r}$ and since we have $\eta(f[c@l]) \in \Gamma$, we also have $\eta(f[c@l']) \in \Gamma$. If there is more than one occurrence of an abstractable constant in a fact that is affected by a label modification, then we can repeatedly apply this argument. Note that the term implication of the (generalized) Horn clauses only replace one occurrence at a time. The reason is that from the label l we cannot be sure that all its occurrences correspond to the same constant $c@l$ in the concrete model, so replacement of only part of the labels is included.

We have thus shown that all the facts in S' are also contained in Γ, modulo the representation function η. $\square$

From Lemmata 2 and 3 immediately follows that the over-approximation is sound:

THEOREM 4. *Given an AIF specification with rules R. If an attack state is reachable with R, then attack $\in LFP(\overline{R})$.*

5. ENCODING TERM IMPLICATION

We show how the term implication rules that we have introduced can be encoded into Horn clauses. Intuitively, the problem is that the rule $s \twoheadrightarrow t$ expresses $C[s] \implies C[t]$ for any context C, and thus summarizes an infinite number of Horn clauses. However, this infinite enumeration can be avoided by limiting ourselves to ones that can be instantiated to a derivable fact. This can be done using a new constant symbol ϵ and two new binary fact symbols *occurs* and *implies* (i.e. these symbols do not occur in the given specification). $occurs(p,t)$ expresses that t is a subterm of some fact that holds, and either

- p is ϵ, then t is a direct subterm of a fact that holds, or
- p is also a subterm of a fact that holds, and t is a direct subterm of p.

Further, $implies(s,t)$ represents a rule of the form $s \twoheadrightarrow t$. For every n-ary fact symbol f (not including *occurs* and *implies*), every m-ary operator g, every $1 \le i \le n$ and every $1 \le j \le m$, we have the following Horn clauses:

$$\begin{array}{l} f(x_1,\ldots,x_n) \to occurs(\epsilon, x_i) \\ occurs(x, g(y_1,\ldots,y_m)) \to occurs(g(y_1,\ldots,y_m), y_j) \\ occurs(g(x_1,\ldots,x_m), x_j) \cdot implies(x_j, y) \\ \quad \to implies(g(x_1,\ldots,x_m), g(x_1,\ldots,x_{j-1},y,x_{j+1},\ldots,x_m)) \\ f(x_1,\ldots,x_n) \cdot implies(x_i, y) \\ \quad \to f(x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_n) \end{array}$$

Let us call these Horn clauses R_0. Consider an arbitrary set of Horn clauses R_h and term implication rules R_t. Call R'_t the Horn clauses that are obtained from R_t by replacing the consequence $s \twoheadrightarrow t$ by the fact $implies(s,t)$.

THEOREM 5. $LFP(R_h \cup R_t) = LFP(R_0 \cup R_h \cup R'_t) \setminus \{implies(\cdot,\cdot), occurs(\cdot,\cdot)\}$

PROOF. Let $\Gamma = LFP(R_h \cup R_t)$ and $\Gamma' = LFP(R_0 \cup R_h \cup R'_t)$ and $\Gamma'' = \Gamma' \setminus \{implies(\cdot,\cdot), occurs(\cdot,\cdot)\}$.

Soundness, i.e. $\Gamma'' \subseteq \Gamma$: $occurs(\cdot,t) \in \Gamma'$ only holds for subterms t of facts in Γ and $implies(t_1,t_2)$ only holds if for any fact $C[t_1] \in \Gamma$ also $C[t_2] \in \Gamma$ holds. As a consequence, the last rule schema of R_0 can only give facts that are in Γ.

Completeness, i.e. $\Gamma \subseteq \Gamma'$: Suppose $f \in \Gamma \setminus \Gamma'$, and suppose f is the "shortest" counter-example, i.e. it can be derived with one rule application of R_t from Γ' (it cannot be a rule from R_h since Γ' is closed under R_h). Let $\phi_1,\ldots,\phi_n \to s \twoheadrightarrow t$ be that rule, σ the substitution under which it is applied and thus $f = C[t\sigma]$ for some context $C[\cdot]$. By the assumption of shortest counter-example, $\phi_i\sigma \in \Gamma'$ and $C[s\sigma] \in \Gamma'$. Thus we also have $implies(s\sigma, t\sigma) \in \Gamma'$.

Moreover, $occurs(\cdot, u) \in \Gamma'$ for all subterms of $C[s\sigma]$ and by that we have the $implies(\cdot, \cdot)$ over corresponding subterms of $C[s\sigma]$ and $C[t\sigma]$. Thus, finally, $C[t\sigma] \in \Gamma'$. $\square$

6. DECIDABILITY

It is straightforward to adapt, to our AIF formalism, the classical proof of [13] that protocol verification is undecidable. This is because this proof relies only on intruder deduction rules that can be applied without any bounds. Moreover, since it does not even use fresh constants, the proof also applies to the abstracted model of an AIF specification. Thus, the security of AIF specification is undecidable both in the concrete and abstract model.

Let us consider the restriction that all rule variables can be instantiated only with variables of a given depth. Such a bounding of substitutions is without loss of attacks in a typed model that can be justified for a large class of protocols by tagging [14]. For the abstract model, decidability is now obvious, because this makes the set of derivable terms finite. For the concrete model, however, we now show that verification is undecidable even when bounding the message depth. [12] shows this for verification in a standard multi-set rewriting approach, but their proof cannot be carried over to AIF immediately because AIF only supports persistent facts and membership conditions for a fixed number of sets. We show that it is expressive enough, however, to simulate Turing machines and thus obtain the following decidability results:

Theorem 6. *Reachability of the attack fact is undecidable both in the concrete and in the abstract model (even when using no sets and fresh data). With a depth restriction on substitutions, the abstract model is decidable, while the concrete model remains undecidable.*

Proof. The idea for encoding Turing machines into message-bounded AIF is that every position of the tape is modeled by a fresh constant, and the symbol is carried by set containment. In an initialization phase, we generate an arbitrary long but finite tape—the length is chosen non-deterministically:[2]

$$\Rightarrow westend(c_0)$$

$$\begin{aligned}&westend(c_0) \cdot c_0 \notin initializing\\ &=\!\![X]\!\!\Rightarrow c_0 \in initializing \cdot succ(c_0, X) \cdot X \in current\end{aligned}$$

$$\begin{aligned}&c_0 \in initializing \cdot X \in current\\ &=\!\![Y]\!\!\Rightarrow succ(X, Y) \cdot Y \in current \cdot c_0 \in initializing\end{aligned}$$

$$\begin{aligned}&c_0 \in initializing \cdot X \in current =\!\![Y]\!\!\Rightarrow\\ &succ(X, Y) \cdot eastend(Y) \cdot c_0 \in current\cdot\\ &c_0 \in q_0 \cdot c_0 \in computing\end{aligned}$$

where q_0 is the initial state of the machine. For every machine transition $(q, s) \to (q', s', L)$ the rule

$$\begin{aligned}&c_0 \in computing \cdot X \in current \cdot X \in q \cdot X \in s \cdot succ(Y, X)\\ &\Rightarrow c_0 \in computing \cdot Y \in current \cdot Y \in q' \cdot X \in s'\end{aligned}$$

The rules for moving right and neutral are similar. Additionally, when the machine reaches the eastend of the tape (which only exists in our model), we go to a sink state of the model from which no further progress can be made:

$$\begin{aligned}&c_0 \in computing \cdot X \in current \cdot X \in q \cdot X \in s \cdot eastend(X)\\ &\Rightarrow c_0 \in stuck\end{aligned}$$

Note that one can easily also encode an initial value of the tape. The Turing machine can reach a certain state q, if the concrete model has a reachable state that contains $c \in q$ for some value c. This can, of course, also be formalized by an attack rule. For this model, a depth bound for variables of 1 (i.e. variables can only be substituted by constants) is no restriction. As reachability of states for a Turing machine is undecidable, so is the reachability of an attack state in the depth bounded concrete model. $\square$

[2]The finiteness is not a restriction as we use a special sink state when reaching the eastend of the tape.

7. EXPERIMENTAL RESULTS

We have implemented the translation from AIF to a set of Horn clauses as described in the previous sections both for the syntax of the theorem prover SPASS and for the syntax of the protocol verifier ProVerif. This implementation along with a library of AIF specifications is available, including more detailed descriptions of the examples presented here [17].

Recall that above we explicitly said that we want to interpret terms and Horn clauses in the free algebra: terms are interpreted as equal iff they are syntactically equal. For instance, for different constants a and b, $a = b$ is false. The same is not necessarily true in first-order logic: it rather depends the structure (i.e. universe and interpretation of all function and relation symbols) in which a formula is interpreted. Thus, there are interpretations in which the formula $a = b$ holds. A formula is *valid*, if it holds in all interpretations (e.g. $a = b \to b = a$).

The SPASS theorem prover allows us to declare a list of axioms $\phi_1, \ldots, \phi_n$ and a conjecture ϕ. It will then try to prove or disprove that $\phi_1 \wedge \ldots \wedge \phi_n \implies \phi$ is valid. We use as the axioms ϕ_i the Horn clauses that result from the translation of AIF and as the conjecture ϕ we use simply *attack*. When SPASS returns "proof found", we know that there is indeed an attack (against the abstract model), as that can be derived from the Horn clauses in any interpretation of the symbols, including the free algebra interpretation. When SPASS however returns "completion found", then for at least one interpretation, *attack* cannot be derived. Of course this means, that the attack cannot be derived in the free algebra interpretation (because if it can be derived in the free algebra interpretation, then it can be derived in any interpretation). Thus if SPASS finds a completion, we know the given protocol is secure in the abstract model with the free algebra interpretation [21] and by the soundness also in the concrete model.

The translation to ProVerif is similar, where we may exploit domain-specific optimizations, such as treatment of the intruder-knowledge fact. In general it turns out that ProVerif is faster than SPASS in finding results, see Table 1, which is not surprising as ProVerif is a dedicated, specialized tool. (The exception where ProVerif times out is discussed below.) We have noted the number of agents that were used in each example, and it can be seen that this has a major influence on the run-time. This is of course due to the fact that with the number of agents, also the number N of sets in our model increases and the number of equivalence classes underlying the abstraction is 2^N.

Problem	Agents	Result	SPASS Time	ProVerif Time
Key-server example	a, i, s	safe	1s	0s
	a, b, c, i, s	safe	37s	0s
SEVECOM (one key)	$hsm, auth, i$	safe	12s	0s
(both keys)	$hsm, auth, i$	unsafe	0s	timeout
ASW	a, i, s	safe	3hrs	6min
TLS (simplified)	a, i	safe	1s	0s
	a, b, i	safe	75s	13s
NSL (w. conf. ch.)	a, b, i	safe	17s	0s
NSPK (w. conf. ch.)	a, b, i	unsafe	0s	0s

Table 1: Experimental results using SPASS and ProVerif

As the first concrete example, we have considered our key-server example, albeit with several honest and dishonest participants. The second example analyzes part of a system for secure vehicle communication from the SEVECOM project [18]. Here, each car has a hardware security module HSM that, amongst others, stores two public root keys of an authority (for verifying messages sent by the authority). The reason for using two root key pairs is that even if one private key is leaked, the authority can still safely update it using the other. We have found some new attacks that were missed in the analysis of [19], because that model does not include the authority (and thus no legal update messages). The attacks are practically limited as they require either several updates within a short period of time or that there is a confusion about which key has been leaked (i.e. the intruder knows one key and the authority updates the other). We have verified the system under the following simple restriction (see [17] for other suggestions to avoid the attacks): we assume that one of the two private keys is never leaked and thus never needs an update, while the other key may be leaked and updated any number of times. Under this restriction, we can verify the following goals: the intruder never finds out private keys (except for ones we give him deliberately), he cannot insert into the HSM any keys he generated himself, and he cannot re-insert old keys. Finally, if we give the intruder both keys, the resulting trivial attack is found by SPASS immediately while ProVerif times out. The reason seems to be that ProVerif dives into the more complicated derivations enabled by the additional intruder knowledge before finding the attack. We will investigate this behavior further as it occurred several times during our experimentation with this example set.

The largest example, and in fact one of the original motivations for this work, is the contract signing protocol ASW based on optimistic fair-exchange [2]. Again, we restrict our discussion to a short summary of ASW and highlighting some key issues of the formalization in AIF, more details are found in [17]. The idea is that two parties can sign a contract in a fair way, i.e. such that finally either both parties or no party has a valid contract. This requires in general a trusted third party TTP, which for ASW is only needed for resolving disputes. The TTP maintains a database of contracts that it has processed so far, which are either aborted or resolved. A resolve means that the TTP issues a valid contract. Whenever an agent asks the TTP for an abort or resolve, the TTP checks whether the contract in question is already registered as aborted or resolved. If this is not the case, then the request to abort or resolve is granted, otherwise the agent gets the abort token or replacement contract stored in the database.

The protocol is based on nonces to which the exchange is bound. Therefore each agent including the TTP maintains a database of nonces. The database stores for each nonce to which parties it relates, to which contractual text, and the status of the respective transaction. For the TTP, the status is just aborted or revoked, for honest agents the status is the stage in the protocol execution (there are several rounds and exceptions). One of the major difficulties of this case study is that the fair exchange relies on the assumption of resilient channels between agents and the TTP, i.e. the intruder (which may be a dishonest contractual partner) cannot block the communication forever. For this, we use a model where the request from the user and the answer from the TTP happen in a single transition. Roughly speaking, we have three cases for each party asking for an abort (and three similar for resolve requests):

- The party is in a stage of the protocol execution where it can ask for an abort, and the TTP has not previously seen the nonce contained in the abort request, i.e. it was not involved in a resolve or an abort. Then we can go to a state where both the party and the TTP have noted the nonce as aborted.
- The other two rules are similar but for the case that the TTP has already noted the nonce as aborted or as resolved and this result is communicated to the agent.

While in general, the handling of resilient channels cannot be done by such a contraction of several steps into a single one, the model in this case covers all real executions if we assume that no honest party sends several requests at a time and that the TTP processes requests sequentially.

Another challenge are the goals of fair exchange itself, namely when one party has a valid contract, then the other one can eventually obtain one. This is in fact a liveness property and cannot directly be expressed. We use here the fact that every agent who does not obtain a contract will eventually contact the trusted third party and get either an abort or resolve. Thus, it is sufficient to check that we never come to a state where one party has a valid contract and the other one has an abort for that contract; this is a safety property.

Finally, we have also considered some "normal" protocols that do not rely on databases, namely a simplified version of TLS, the famous flawed NSPK and the fixed variant by Lowe (NSL). The reason is that these protocols are standard examples. Also this demonstrates that we can use databases of nonces or keys as an alternative way to describe the relevant state-information of agents. For NSPK and NSL we use confidential channels instead of public-key encryption.

The experimental results demonstrate that our abstraction approach is feasible for a variety of verification problems of security protocols and web services.

8. CONCLUSIONS

The abstraction and over-approximation of protocols and web services by a set of Horn clauses is a very successful method in practice [7, 11, 10, 21, 6]. In contrast to classical model-checking approaches, this kind of over-approximation does not suffer from the usual interleaving problems and can verify protocols for an unbounded number of sessions. The technique has however limitations for protocols and web services that are based on databases of keys, contracts, or even access rights, where revocation is possible, so that the set of true facts does not monotonically grow with the transitions.

We present a new way of abstraction in the spirit of the Horn clauses approach that can handle such databases and thus broadens the scope of this abstraction method. The abstraction of data we propose is based on the membership of the data in the databases. The updating of the databases requires also an update of the abstraction of the data which we can declaratively express with a new form of rule we have introduced, the term implication rule. We show how to encode this rule into standard Horn clauses. As a consequence we can use with ProVerif an existing tool from the abstraction community, and even the general purpose first-order theorem prover SPASS. The SEVECOM and ASW examples show that our method is feasible for modeling complex real-world systems with databases and APIs that, for reasons of their non-monotonic behavior, were previously out of the scope of the standard abstraction-based methods. While the AIF-library is still small, this suggest that our method is practically feasible to tackle exactly what is missing for the verification of more complex cryptographic systems.

[20] considers an abstraction of keys in an API by attributes; this has some similarity with our set-membership abstraction. However, the attributes in [20] are static (i.e. set memberships cannot change).

Our new language AIF gives a convenient way of writing specifications in an un-abstracted form. Still, AIF is too low-level to be used by a protocol or web service designer. We thus plan as part of future work to connect more high-level languages. Also we plan to build a tool with native support for the term-implication rules and for other improvements specific to our approach. Further, the approach is currently limited to a fixed number N of sets; we plan to investigate how we can avoid this limitation. Another interesting question we want to consider is the relation of our approach to two quite different approaches, namely static analysis [9] and type-based analysis [5], which, besides all differences, show some similarities with our approach.

9. REFERENCES

[1] A. Armando, L. Compagna. SAT-based Model-Checking for Security Protocols Analysis. *Int. J. of Information Security*, 6(1):3–32, 2007.

[2] N. Asokan, V. Shoup, M. Waidner. Asynchronous protocols for optimistic fair exchange. In *IEEE Symposium on Research in Security and Privacy*, 86–99. 1998.

[3] AVISPA. Deliverable 2.3: The Intermediate Format, 2003. Available at `www.avispa-project.org/publications.html`.

[4] D. Basin, S. Mödersheim, L. Viganò. OFMC: A symbolic model checker for security protocols. *Int. J. of Information Security*, 4(3):181–208, 2005.

[5] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, S. Maffeis. Refinement types for secure implementations. In *CSF*, 17–32. 2008.

[6] K. Bhargavan, C. Fournet, A. D. Gordon, R. Pucella. Tulafale: A security tool for web services. In *FMCO*, 197–222. 2003.

[7] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW'01*, 82–96. IEEE Computer Society Press, 2001.

[8] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[9] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[10] Y. Boichut, P.-C. Héam, O. Kouchnarenko, F. Oehl. Improvements on the Genet and Klay technique to automatically verify security protocols. In *AVIS'04*, 1–11. 2004.

[11] L. Bozga, Y. Lakhnech, M. Perin. Hermes: An automatic tool for the verification of secrecy in security protocols. In *CAV'03*, LNCS 2725, 219–222. Springer-Verlag, 2003.

[12] N. Durgin, P. Lincoln, J. Mitchell, A. Scedrov. Undecidability of bounded security protocols. In *Formal methods and security Protocols*. 1999.

[13] S. Even, O. Goldreich. On the security of multi-party ping-pong protocols. In *FOCS*, 34–39. 1983.

[14] J. Heather, G. Lowe, S. Schneider. How to prevent type flaw attacks on security protocols. In *CSFW'00*. IEEE Computer Society Press, 2000.

[15] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[16] S. Mödersheim. On the Relationships between Models in Protocol Verification. *J. of Information and Computation*, 206(2–4):291–311, 2008.

[17] S. Mödersheim. Verification based on set-abstraction using the AIF framework. Tech. Rep. IMM-Technical report-2010-09, DTU/IMM, 2010. Available at `www.imm.dtu.dk/~samo`.

[18] SEVECOM. Deliverable 2.1-App.A: Baseline Security Specifications, 2009. Available at `www.sevecom.org`.

[19] G. Steel. Towards a formal security analysis of the Sevecom API. In *ESCAR*. 2009.

[20] G. Steel. Abstractions for Verifying Key Management APIs. In *SecReT*. 2010.

[21] C. Weidenbach. Towards an automatic analysis of security protocols. In *CADE*, LNCS 1632, 378–382. Berlin, 1999.

[22] C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, D. Topic. System description: Spass version 3.0. In *CADE*, 514–520. 2007.