AIF-ω: Set-Based Protocol Abstraction with Countable Families

Sebastian Mödersheim[1] and Alessandro Bruni[2]

[1] DTU Compute, Lyngby, Denmark
[2] IT University of Copenhagen, Copenhagen, Denmark

Abstract. Abstraction based approaches like ProVerif are very efficient in protocol verification, but have a limitation in dealing with stateful protocols. A number of extensions have been proposed to allow for a limited amount of state information while not destroying the advantages of the abstraction method. However, the extensions proposed so far can only deal with a *finite* amount of state information. This can in many cases make it impossible to formulate a verification problem for an unbounded number of agents (and one has to rather specify a fixed set of agents). Our work shows how to overcome this limitation by abstracting state into countable families of sets. We can then formalize a problem with unbounded agents, where each agent maintains its own set of keys. Still, our method does not loose the benefits of the abstraction approach, in particular, it translates a verification problem to a set of first-order Horn clauses that can then be efficiently verified with tools like ProVerif.

1 Introduction

A very successful idea in protocol verification, most prominently in the ProVerif tool, is an abstraction approach that over-approximates every possible protocol behavior by a set of first-order Horn clauses, rather than considering the set of reachable states [12,2]. The benefit is that one completely avoids the state-explosion problem (i.e., that the number of reachable state grows exponentially with the number of sessions) and allows one to even deal with an unbounded number of sessions. The fact that this approach "throws away" the state space does indeed not hurt the modeling and analysis for most protocols: typically, the amount of context needed to participate in a protocol is contained within a session, and all information that is shared across different sessions is immutable like agent names and long-term keys.

We run into limitations with this approach, however, when we consider protocols that use some kind of long-term information that can be changed across multiple sessions of the protocol. As an example, a web server maintains a database of ordered goods, or a key server stores valid and revoked keys. In the case of a key server, some actions can just be performed while the key is valid, but as soon as this key is revoked, the same actions are disabled. This behavior does not directly work with the Horn-clause approach, because they have the monotonicity property of classic logics: what is true cannot become false by learning

more information. This is at odds with any "non-monotonic" behavior, i.e., that something is no longer possible after a particular event has occurred.

Several works have proposed extensions of the abstraction approach by including a limited amount of state information, so as to allow the analysis of stateful protocols, without destroying the large benefits of the approach. The first was the AIF tool that allows one to declare a fixed number N of sets [10]. One can then specify a transition system with an unbounded number of constants. These constants can be added to, and removed from, each of the sets upon transitions, and transitions can be conditioned by set memberships. The main idea is here that one can abstract these constants by their set membership, i.e., partitioning the constants into 2^N equivalence classes for a system with N sets. The AIF tool generates a set of Horn clauses using this abstraction, and can use either ProVerif or the first-order theorem prover SPASS [13] to check whether a distinguished symbol *attack* can be derived from the Horn clauses. The soundness proof shows that if the specified transition system has an attack state then *attack* can be derived from the corresponding Horn clause model. There are two more approaches that similarly bring state information into ProVerif: StatVerif [1] and Set-π [4]. We discuss them in the related work.

While AIF is an infinite state approach, it has the limitation to a fixed number N of sets. For instance, when modeling a system where every user maintains its own set of keys, one needs to specify a fixed number of users, so as to arrive at a concrete number N of sets. The main contribution of AIF-ω is to overcome precisely this limitation and instead allow for specifying N *families* of sets, where each family can consist of a countably infinite number of sets. For instance, we may declare that $User$ is an infinite set and define a family $ring(User)$ of sets so that each user $a \in User$ has its own set of keys $ring(a)$. To make this feasible with the abstraction approach, we however need to make one restriction: the sets of a family must be pairwise disjoint, i.e., $ring(a) \cap ring(b) = \emptyset$ for any two users a and b. In fact, we do allow for AIF-ω specifications that could potentially violate this property, but if the disjointness is violated, it counts as an attack.

The contributions of this work are the formal development and soundness proof of this countable-family abstraction. It is in fact a generalization of the AIF approach. Besides this generalization, AIF-ω has also a direct practical advantage in the verification tool: experiments show for instance that the verification for infinitely many agents in an example is more efficient than the finite enumeration of agents in AIF. In fact, the infinite agents specification has almost the same run time as the specification with a single agent for each role in AIF.

The rest of this paper is organized as follows. In section 2 we formally define AIF-ω and introduce preliminaries along the way. In section 3 we define the abstraction and translation to Horn clauses and prove the soundness in section 4. We discuss how to encode the approach in SPASS and ProVerif as well as experimental results in section 5. We discuss related work and conclude in section 6.

2 Formal Definition of AIF-ω

We go right into the definition of the AIF-ω language, and introduce all preliminaries along the way. An AIF-ω specification consists of the following sections: declaring user-defined types, declaring families of sets, declaring function and fact symbols, and finally defining the transition rules. We explain these concepts step-by-step and for concreteness illustrate it with the running example of a keyserver adapted from our previous paper [10].

2.1 Types

An AIF-ω specification starts with a declaration of user-defined types. These types can either be given by a complete enumeration (finite sets), or using the operator "..." one can declare that the type contains a countable number of elements. Finally, we can also build the types as the union of other types. For the keyserver example, let us define the following types:

$$\begin{array}{lll} \mathit{Honest} = \{a, b, \ldots\} & \mathit{Dishon} = \{i, p, \ldots\} & \mathit{User} = \mathit{Honest} \cup \mathit{Dishon} \\ \mathit{Server} = \{s, \ldots\} & \mathit{Agent} = \mathit{User} \cup \mathit{Server} & \mathit{Status} = \{\mathit{valid}, \mathit{revoked}\} \end{array}$$

This declares the type *Honest* to be a countably infinite set that contains the constants a and b. Similarly *Dishon* and *Server* are defined. It may be intuitively clear that in this declaration, the sets *Honest* and *Dishon* should be disparate types, but to make the "..." notation formally precise, we give each type T an *extensional semantics* $[\![T]\!]$. To that end, for each "...", we introduce new constants $t_1, t_2, \ldots$ so that for the running example we have for instance:

$$[\![\mathit{Honest}]\!] = \{a, b\} \cup \{\mathit{honest}_n \mid n \in \mathbb{N}\} \quad [\![\mathit{Dishon}]\!] = \{s, p\} \cup \{\mathit{dishon}_n \mid n \in \mathbb{N}\}$$

Comparing AIF-ω with the previous language AIF, the ability to define infinite types and families of sets over these types, are the essential new features. Drastically speaking, "..." is thus what you could not do in AIF. The complexity of this paper however suggests that it is not an entirely trivial generalization.

Besides the user-defined types, we also have two built-in types: *Value* and *Untyped*. The type *Value* is the central type of the approach, because all sets of the system can only contain elements of type value, and all freshly created elements must be of type value. It is thus exactly those entities that we later want to replace by abstract equivalence classes. Let thus $\mathfrak{A} = \{\mathit{abs}_n \mid n \in \mathbb{N}\}$ be a countable set of constants (again disjoint from all others) and $[\![\mathit{Value}]\!] = \mathfrak{A}$. Second, we have also the "type" *Untyped*. Below, we define the set of ground terms $\mathcal{T}_\Sigma$ that includes all constants and composed terms that can be built using function symbols. We want the type *Untyped* to summarize arbitrary such terms, and thus define $[\![\mathit{Untyped}]\!] = \mathcal{T}_\Sigma$.

2.2 Sets

The core concept of AIF-ω is using sets of values from $\mathfrak{A}$ to model simple "databases" that can be queried and modified by the participants of the protocols. These sets can even be shared between participants, and the modeler has a

great freedom on how to use them. For our running example we want to declare sets for the key ring of every user, and for every server a database that contains for all users the currently valid and revoked keys:

$$\mathit{ring}(\mathit{User}!) \quad \mathit{db}(\mathit{Server}!, \mathit{User}!, \mathit{Status}!)$$

This declares two *families* of sets, the first family consists of one set $\mathit{ring}(c)$ for every $c \in [\![\mathit{User}]\!]$ and the second family consists of one set $\mathit{db}(c_1, c_2, c_3)$ for every $c_1 \in [\![\mathit{Server}]\!]$, $c_2 \in [\![\mathit{User}]\!]$, and $c_3 \in [\![\mathit{Status}]\!]$.

The exclamation mark behind the types in the set declaration has a crucial meaning: with this the modeler defines a *uniqueness invariant* on the state space, namely that the sets of this family will be pairwise disjoint for that parameter. In the example, $\mathit{ring}(c_1) \cap \mathit{ring}(c_2) = \emptyset$ for any $c_1 \neq c_2$, and $\mathit{db}(c_1, c_2, c_3) \cap \mathit{db}(c_1', c_2', c_3') = \emptyset$ if $(c_1, c_2, c_3) \neq (c_1', c_2', c_3')$. This invariant is part of the definition of the transition system: it is an attack, if a state is reachable in which the invariant is violated.

An important requirement of AIF-ω is that *all family parameters of infinite type must have the uniqueness invariant*. Thus, it is not allowed to declare $\mathit{ring}(\mathit{Agent})$, because $[\![\mathit{Agent}]\!]$ is infinite. However, it *is* allowed to declare $\mathit{db}(\mathit{Server}!, \mathit{Agent}!, \mathit{Status})$ since $[\![\mathit{Status}]\!]$ is finite. This declaration with non-unique *Status* could be specified using two families $\mathit{db}_{\mathit{valid}}(\mathit{Server}!, \mathit{Agent}!)$ and $\mathit{db}_{\mathit{revoked}}(\mathit{Server}!, \mathit{Agent}!)$ instead. We thus regard non-unique arguments of a finite type as syntactic sugar that is compiled away in AIF-ω.

Since non-unique arguments are syntactic sugar, let us assume for the rest of the paper an AIF-ω specification (like in the running example) where all set parameters have the uniqueness invariant (i.e., the ! symbol). Let us denote the families of sets in general as $s_1, \ldots, s_N$ where N is the number of declared families, i.e., in the example, $N = 2$ with $s_1 = \mathit{ring}$ and $s_2 = \mathit{db}$. We thus have for every $1 \leq i \leq N$ the uniqueness invariant that $s_i(a_1, \ldots, a_n) \cap s_i(b_1, \ldots, b_n) = \emptyset$ whenever $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$.

2.3 Functions, Facts, and Terms

Finally the user can declare a set of functions and facts (predicates) with their arities. For the example let us have:

$$\text{Functions: } \mathit{inv}/1,\ \mathit{sign}/2,\ \mathit{pair}/2 \qquad \text{Facts: } \mathit{iknows}/1, \mathit{attack}/0$$

Intuitively, $\mathit{inv}(pk)$ represents the private key corresponding to public key pk, $\mathit{sign}(\mathit{inv}(pk), m)$ represents a digital signature on message m with private key $\mathit{inv}(pk)$, *pair* is for building pairs of messages; $\mathit{iknows}(m)$ expresses that the intruder knows m, and *attack* represents a flag we raise as soon as an attack has occurred (and we later ask whether *attack* holds in any reachable state).

Definition 1. *Let Σ consist of all function symbols, the extension $[\![T]\!]$ of any user-defined type T, and the values $\mathfrak{A}$ (where all constants are considered as function symbols with arity 0). Let $\mathcal{V}$ be a set of variables disjoint from Σ. We*

define $\mathcal{T}_\Sigma(V)$ to be the set of terms that can be built from Σ and $V \subseteq \mathcal{V}$, i.e., the least set that contains V and such that $f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma(V)$ if $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(V)$ and $f/n \in \Sigma$. When $V = \emptyset$, we also just write $\mathcal{T}_\Sigma$, and we call this the set of ground terms. *A* fact *(over Σ and V) has the form $f(t_1, \ldots, t_n)$ where f/n is a fact symbol and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(V)$.*

2.4 Transition Rules

The core of an AIF-ω specification is the definition of its transition rules that give rise to an infinite-state transition system, where each state is a set of facts and set conditions (as defined below). The initial state is simply the empty set of facts and set conditions. We proceed as follows: we first give the formal definition of syntax and semantics of rules. We then discuss the details at hand of the rules of the running example. Finally, we give a number of restrictions on rules that we need for the abstraction approach in the following section.

In the following we often speak of *the type* of a variable (and may write $X : T$); this is because variables occur only within rules (not within states) and are then always declared as part of the rule parameters.

Definition 2. *A* positive set condition *has the form $t \in s_i(A_1, \ldots, A_n)$ where t is either a constant of $\mathfrak{A}$ or a variable of type Value, the family s_i of sets has been declared as $s_i(T_1!, \ldots, T_n!)$, and each A_i is either an element of $[\![T_i]\!]$ or a variable of type T_i' with $[\![T_i']\!] \subseteq [\![T_i]\!]$. A positive set condition is called* ground *if it contains no variables. A* negative set condition *has the form $t \notin s_i(\_)$ where t and s_i are as before.*

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

$$r(X_1\colon T_1, \ldots, X_n\colon T_n) = LF \cdot S_+ \cdot S_- \Rightarrow\!\!\!\lfloor F \rfloor\!\!\!\Rightarrow RF \cdot RS$$

where

1. *$X_1, \ldots, X_n$ are variables and $T_1, \ldots, T_n$ are their types; We often abbreviate $(X_1\colon T_1, \ldots, X_n\colon T_n)$ by $\boldsymbol{X}\colon \boldsymbol{T}$;*
2. *The X_i are exactly the variables that occur in the rule*
3. *LF and RF are sets of facts;*
4. *S_+ and RS are sets of positive set conditions;*
5. *S_- is a set of negative set conditions;*
6. *F is a set of variables that are of type Value and they do not occur in LF, S_+, or S_-.*
7. *For every untyped variable that occurs in RF, it also occurs in LF.*

Let $\mathcal{V}_\mathfrak{A}$ denote the subset of the X_i that have type Value.

A rule r gives rise to a state-transition relation $\Rightarrow_r$ where $S \Rightarrow_r^\sigma S'$ holds for states S and S' and a substitution σ iff

- *σ has domain $\{X_1, \ldots, X_n\}$ and $\sigma(X_i) \in [\![T_i]\!]$ for each $1 \leq i \leq n$;*
- *$(LF \cdot S_+)\sigma \subseteq S$,*

– *For every negative set condition* $X \notin s_i(\_)$ *of* S_-*, state* S *does not contain* $\sigma(X) \in s_i(a_1, \ldots, a_m)$ *for any* $(a_1, \ldots, a_m)$*.*
– $S' = (S \setminus \sigma(S_+)) \cup \sigma(RF) \cup \sigma(RS)$*,*
– $\sigma(F)$ *are fresh constants from* $\mathfrak{A}$ *(i.e. they do not occur in* S *or the AIF-*ω *specification).*

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* $r \in R$ *(ignoring substitution* σ*) and* $\cdot^*$ *is the reflexive transitive closure.* □

Intuitively, a rule r can be applied under *match* σ if the left-hand side facts $\sigma(LF)$ and positive set conditions $\sigma(S_+)$ are present in the current state, and none of the negative conditions $\sigma(S_-)$ holds. Upon transition we remove the matched set conditions $\sigma(S_+)$ and replace them with the right-hand side set conditions $\sigma(RS)$ and facts $\sigma(RF)$. The semantics ensures that all reachable states are ground, because σ must instantiate all variables with ground terms. The semantics defines facts to be *persistent*, i.e., when present in a state, then also in all successor states. Thus only set conditions can be "taken back".

To illustrate the AIF-ω rules more concretely, we now discuss the rules of the key server example. We first look at the three rules that describe the behavior of honest users and servers:

$$\begin{aligned}
&\mathit{keyReg}(A\colon \mathit{User}, S\colon \mathit{Server}, \mathit{PK}\colon \mathit{Value}) = \\
&\quad =[\mathit{PK}]\Rightarrow \mathit{iknows}(\mathit{PK}) \cdot \mathit{PK} \in \mathit{ring}(A) \cdot \mathit{PK} \in \mathit{db}(S, A, \mathit{valid})
\end{aligned}$$

$$\begin{aligned}
&\mathit{userUpdateKey}(A\colon \mathit{Honest}, S\colon \mathit{Server}, \mathit{PK}\colon \mathit{Value}, \mathit{NPK}\colon \mathit{Value}) = \\
&\quad \mathit{PK} \in \mathit{ring}(A) \cdot \mathit{iknows}(\mathit{PK}) \\
&\quad =[\mathit{NPK}]\Rightarrow \mathit{NPK} \in \mathit{ring}(A) \cdot \mathit{iknows}(\mathit{sign}(\mathit{inv}(\mathit{PK}), \mathit{pair}(A, \mathit{NPK})))
\end{aligned}$$

$$\begin{aligned}
&\mathit{serverUpdateKey}(A\colon \mathit{User}, S\colon \mathit{Server}, \mathit{PK}\colon \mathit{Value}, \mathit{NPK}\colon \mathit{Value}) = \\
&\quad \mathit{iknows}(\mathit{sign}(\mathit{inv}(\mathit{PK}), \mathit{pair}(A, \mathit{NPK}))) \cdot \mathit{PK} \in \mathit{db}(S, A, \mathit{valid}) \cdot \mathit{NPK} \notin \mathit{db}(_) \\
&\quad \Rightarrow \mathit{PK} \in \mathit{db}(S, A, \mathit{revoked}) \cdot \mathit{NPK} \in \mathit{db}(S, A, \mathit{valid}) \cdot \mathit{iknows}(\mathit{inv}(\mathit{PK}))
\end{aligned}$$

Intuitively, the *keyReg* rule describes an "out-of-band" key registration, e.g. a physical visit of a user A at an authority S. Here, the left-hand side of the rule is empty: the rule can be applied in any state. Upon the arrow, we have PK, meaning that in this transition we create a *fresh* value from $\mathfrak{A}$ that did not occur previously. Intuitively this is a new public key that the user A has created. We directly give the intruder this public key, as it is public. The two set conditions formalize that the key is added to the key ring of A and that the server S stores PK as a valid key for A in its database. Of course, the user A should also know the corresponding private key $\mathit{inv}(PK)$, but we do not explicitly express this (and rather later make a special rule for dishonest agents). Note that, having no prerequisites, this rule can be applied in any state, and thus every user can register an unbounded number of keys with every server.

The *userUpdateKey* rule now describes that an honest user (for the behavior of dishonest users, see below) can update any of its current keys PK (the requirement $\mathit{iknows}(PK)$ is explained below) by creating a new key NPK and

sending an update message $sign(inv(PK), pair(A, NPK))$ to the server, signing the new key with the current key. As it is often done, this example does not explicitly model sending messages on an insecure channel and rather directly adds it to the intruder knowledge (see also the model of receiving a message in the next rule). Further, NPK is added to the key ring of A. Finally, observe that the set condition $PK \in ring(A)$ is not repeated on the right-hand side. This means that PK is actually removed from the key ring. Of course this is a simplistic example: in a real system, the update would include some kind of confirmation message from the server, and the user would not throw away the current key before receiving the confirmation.

The third rule *serverUpdateKey* formalizes how a server processes such an update message: it will check that the signing key PK is currently registered as a valid key and that NPK is not yet registered, neither as valid nor as revoked. If so, it will register NPK as a valid key for A in its database. PK is now removed from the database of valid keys for A, because $PK \in db(S, A, valid)$ is not present on the right-hand side; PK is added to the revoked keys instead. Note that the check $NPK \notin db(\_)$ on the left-hand side actually models a server that checks that *no* server of *Server* has seen this key so far.[3] As a particular "chicane", we finally give the intruder the private key to every revoked key. This is modeling that we want the protocol to be secure (as we define shortly) even when the intruder can get hold of old private keys.

Remaining rules of the example model the behavior of dishonest agents and define what constitutes an attack:

$$iknowsAgents(A\colon Agent) = \;\Rightarrow iknows(A)$$

$$sign(M1, M2\colon Untyped) = iknows(M1) \cdot iknows(M2) \Rightarrow iknows(sign(M1,M2))$$

$$open(M1, M2\colon Untyped) = iknows(sign(M1, M2)) \Rightarrow iknows(M2)$$

$$pair(M1, M2\colon Untyped) = iknows(M1) \cdot iknows(M2) \Rightarrow iknows(pair(M1,M2))$$

$$proj(M1, M2\colon Untyped) = iknows(pair(M1,M2)) \Rightarrow iknows(M1) \cdot iknows(M2)$$

$$dishonKey(A\colon Dishon, PK\colon Value) = iknows(PK) \cdot PK \in ring(A) \\ \Rightarrow iknows(inv(PK)) \cdot PK \in ring(A)$$

$$attdef(A\colon Honest, S\colon Server) = iknows(inv(PK)) \cdot PK \in db(S,A,valid) \Rightarrow attack$$

The first rules are basically a standard Dolev-Yao intruder for the operators we use (i.e., the intruder has access to all algorithms like encryption and signing, but cannot break cryptography and can thus apply the algorithms only to messages and keys he knows). The rule *dishonKey* expresses that the intruder gets the private key to all public keys registered in the name of a dishonest agent. This

[3] If one would rather like to model that servers cannot see which keys the other servers consider as valid or revoked, one runs indeed into the boundaries of AIF-ω here. This is because in this case one must accept that at least a dishonest agent can register the same key at two different servers, violating the uniqueness invariant. If one wants to model such systems, one must resort to finitely many servers.

reflects the common model that all dishonest agents work together. Finally the rule *attdef* defines security indirectly by specifying what is an attack: when the intruder finds out a private key that some server S considers currently as a valid key of an honest agent A. One may give more goals, especially directly talking about authentication—note that this secrecy goal implicitly refers to authentication, as the intruder would for instance have a successful attack if he manages to get a server S to accept as the public key of an honest agent any key to which the intruder knows the private key. For an in-depth discussion of formalizing authentication goals, see [4].

2.5 Restrictions and Syntactic Sugar

There are a few forms of rules that are problematic for the treatment in the abstraction approach later. Actually, problematic rules may also indicate that the modeler could have made a mistake (i.e. has something different in mind than what the rule formally means). Most of the problematic rules are either paradox (and thus useless) or can be compiled into non-problematic variants as syntactic sugar. We first define problematic, or inadmissible, rules, then discuss what is problematic about them and how they are handled. Afterwards, we assume to deal only with admissible rules.

Definition 3. *A rule* $r(\boldsymbol{X : Type}) = LF \cdot S_+ \cdot S_- =[F]\Rightarrow RF \cdot RS$ *is called* inadmissible*, if any of the following holds:*

1. *Either* $X \in s_i(\ldots)$ *occurs in* S_+ *or* $X \notin s_i(\_)$ *occurs in* S_-*, but* X *does not occur in* LF*, or*
2. $X \in s_i(A_1, \ldots, A_n)$ *occurs in* S_+ *and* $X \notin s_i(\_)$ *occurs in* S_-*, or*
3. *both* $X \in s_i(A_1, \ldots, A_n)$ *and* $X \in s_i(A'_1, \ldots, A'_n)$ *occur in* S_+ *for* $(A'_1, \ldots, A'_n) \neq (A_1, \ldots, A_n)$*, or*
4. *both* $X \in s_i(A_1, \ldots, A_n)$ *and* $X \in s_i(A'_1, \ldots, A'_n)$ *occur in* RS *for* $(A'_1, \ldots, A'_n) \neq (A_1, \ldots, A_n)$*, or*
5. $X \in S_i(A_1, \ldots, A_n)$ *occurs in* RS *and neither:*
   - $X \in F$*, nor*
   - $X \notin s_i(\_, \ldots, \_)$ *occurs in* S_-*, nor*
   - $X \in s_i(A'_1, \ldots, A'_n)$ *occurs in* S_+*;*

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

Also we define the distinguished semantics *as the following restriction of the* $\Rightarrow$ *relation:* $S \Rightarrow^{\sigma}_{r} S'$ *additionally requires that* $\sigma(X) \neq \sigma(Y)$ *for any distinct variables* $X, Y \in \mathcal{V}_{\mathfrak{A}}$*.* □

Condition (1) is in fact completely fine in a specification and it only causes problems in the abstraction approach later (since set conditions are removed from the rules and put into the abstraction of the data). To "rule out" such an occurrence without bothering the modeler, the AIF-ω compiler simply introduces a new fact symbol $occurs/1$ and adds $occurs(X)$ on the left-hand and right-hand side of every rule for every $X \in \mathcal{V}_{\mathfrak{A}}$ of the rule. (In the running example, the rule

userUpdateKey has $iknows(PK)$ on the left-hand side, simply because without it, it would satisfy condition (1); since in this example the intruder knows all public keys, this is the easiest way to ensure admissibility without introducing *occurs*.)

Condition (2) means that the rule is simply never applicable. The compiler refuses it as this is a clear specification error.

An example for condition (3) is the rule: $r(\ldots) = X \in s_1(A) \cdot X \in s_1(B) \Rightarrow \ldots$ Recall that our uniqueness invariant forbids two distinct sets of the same family (like s_1 here) to have an element in common. So this rule cannot be applicable in any state that satisfies the invariant unless $\sigma(A) = \sigma(B)$. As this may be a specification error, the compiler also refuses this with the suggestion to unify A and B. Similarly, condition (4) forbids the same situation on the right-hand side, as for $\sigma(A) \neq \sigma(B)$ the invariant would be violated. Also in this case, the compiler refuses the rule with the suggestion to unify A and B.

An example of a rule that is inadmissible by condition (5) is the following:

$$r(X\colon \mathit{Value}) = p(X) \implies X \in s_1(a)$$

The problem here is that we insert X into $s_1(a)$ without checking if possibly X is already member of another set of the s_1 family. Suppose for instance the state $S = \{p(c), c \in s_1(b)\}$ is reachable, then r is applicable and produces state $S = \{p(c), c \in s_1(a), c \in s_1(b)\}$ violating the invariant that the sets belonging to the same family are pairwise disjoint. However, note that r is only *potentially* problematic: it depends on whether we can reach a state in which both $p(c)$ and $c \in s_1(\ldots)$ holds for some constant $c \in \mathfrak{A}$, otherwise r is fine.

The AIF-ω compiler indeed allows for such inadmissible rules that potentially violate the invariant, but transforms them into the following two admissible rules:

$$r_1(X\colon \mathit{Value}) = p(X) \cdot X \notin s_1(\_) \implies X \in s_1(a)$$
$$r_2(X\colon \mathit{Value}, A\colon T) = p(X) \cdot X \in s_1(A) \implies \mathit{attack}$$

where T is the appropriate type for the parameter of s_1. Thus, we have turned this into one rule for the "safe" case (r_1) where X is not previously in any set of s_1, and one for the "unsafe" case (r_2) where X is already in s_1 and applying the original rule r would lead to a violation of the invariant (unless $\sigma(A) = a$);[4] in this case we directly raise the attack flag. Note that neither r_1 nor r_2 still have the problem of condition (5). The compiler simply performs such case splits until no rule has the problem of condition (5) anymore. We thus allow the user to specify rules that would potentially violate the invariant, but make it part of the analysis that no reachable state actually violates it.

Finally, consider the restriction to a distinguished semantics of Definition 3. Here is an example why the standard semantics of Definition 2 can make things very tricky:

$$r(\ldots) = p(X, Y) \cdot X \in s_1(a) \cdot Y \in s_1(a) \rightarrow X \in s_1(a)$$

[4] In fact, we are here over-careful as the case $\sigma(A) = a$ in the second rule would still be fine; but a precise solution in general would require inequalities—which we leave for future work.

Suppose the state $S = \{p(c, c) \cdot c \in s_1(a)\}$ is reachable, then the rule clearly is applicable in S (with $\sigma(X) = \sigma(Y) = c$), but the rule tells us that Y should be removed from $s_1(a)$ while X stays in there. (Here, the semantics tells us that the positive $X \in s_1(a)$ "wins", and the successor state is also S.) However, it would be quite difficult to handle such cases in the abstraction and it would further complicate the already complex set of conditions of Definition 3.

Therefore we like to work in the following with the distinguished semantics of Definition 3, where the instantiation $\sigma(X) = \sigma(Y)$ in the above example is simply excluded. To make this possible without imposing the restriction on the modeler, the AIF-ω compiler applies the following transformation step. We check in every rule for every pair of variables $X, Y \in \mathcal{V}_{\mathfrak{A}}$ whether $\sigma(X) = \sigma(Y)$ is possible, i.e. neither X nor Y is in the fresh variables, and left-hand side memberships of X and Y do not contradict each other. (Observe that in none of the rules of the running example, such a unification of two $\mathcal{V}_{\mathfrak{A}}$ variables is possible.) If the rule does not prevent $X = Y$, the AIF-ω compiler generates a variant of the rule where Y is replaced by X. Thus, we do not loose the case $X = Y$ even when interpreting the rules in the distinguished semantics.

As a fruit of all this restriction we can prove that admissible rules cannot produce a reachable state that violates the invariant:

Lemma 1. *Considering only admissible rules in the distinguished semantics. Then there is no reachable state S and constant $c \in \mathfrak{A}$ such that S contains both $c \in s_i(a_1, \ldots, a_n)$ and $c \in s_i(a'_1, \ldots, a'_n)$ for any $(a_1, \ldots, a_n) \neq (a'_1, \ldots, a'_n)$.*

Proof. By induction over reachability. The property trivially holds for the initial state. Suppose S is a reachable state with the property, and $S \Rightarrow_r^\sigma S'$. Suppose S' contains both $c \in s_i(a_1, \ldots, a_n)$ and $c \in s_i(a'_1, \ldots, a'_n)$. Since S enjoys the property, at least one of the two set conditions has been introduced by the transition. Thus there is a value variable X in r and $\sigma(X) = c$, and $X \in s_i(A_1, \ldots, A_n)$ is in RS and either $\sigma(A_j) = a_j$ or $\sigma(A_j) = a'_j$, so without loss of generality, assume $\sigma(A_j) = a_j$. By excluding (4) of Def. 3, RS cannot contain another set condition $X \in s_i(A'_1, \ldots, A'_n)$ (such that $\sigma(A'_j) = a'_j$), so $c \in s_i(a'_1, \ldots, a'_n)$ must have been present in S already. By excluding (5), we have however either of the following cases:

- $X \notin s_i(\_)$ is in S_-, but that clearly contradicts the fact that $\sigma(X) \in s_i(a'_1, \ldots, a'_n)$ is in S.
- $X \in s_i(B_1, \ldots, B_n)$ is in S_+, and by excluding (3) and (2) this is the only positive or negative condition for X on the s_i family. This means that only $\sigma(B_j) = a'_j$ is possible, so $c \in s_i(a'_1, \ldots, a'_n)$ actually gets removed from the state upon transition, and is no longer present in S'.
- $X \in F$, but that is also absurd since then $\sigma(X)$ cannot occur in S.

So in all cases, we get to a contradiction, so we cannot have c being a member of two sets of the s_i family. □

3 Abstraction

We now define a translation from AIF-ω rules to Horn clauses augmented with a special kind of rules, called term implication. (We show in a second step how to encode these term implication rules into Horn clauses, to keep the approach easier to grasp and to work with.) The basic idea is that we abstract the constants of $\mathfrak{A}$ into equivalence classes that are easier to work with. In fact, in the classic AIF, we had finitely many equivalence classes, but in AIF-ω we have a countable number of equivalence classes, due to the countable families of sets.

The abstraction of a constant $c \in \mathfrak{A}$ for a state S shall be $(e_1, \ldots, e_N)$ where e_i represents the set membership for the family s_i: either $e_i = 0$ if c belongs to no member of s_i or $e_i = s_i(a_1, \ldots, a_n)$ if e_i belongs to set $s_i(a_1, \ldots, a_n)$ in S. For instance in a state with set conditions

$$\{c_1 \in db(a, s, revoked), c_2 \in db(b, s, valid), c_2 \in ring(b)\}$$

the abstraction of c_1 is $(0, db(a, s, revoked))$ and similarly the abstraction of c_2 is $(ring(b), db(b, s, valid))$. Thus, we do not distinguish concrete constants in the abstraction whenever they have the same set memberships.

The second main idea (as in other Horn-clause based approaches) is to formulate Horn clauses that entail all facts (under the abstraction) that hold in any reachable state. This is like merging all states together into a single big state.

3.1 Translation of the Rules

We first define how admissible AIF-ω rules are translated into Horn clauses and then show in the next section that this is a sound over-approximation (in the distinguished semantics).

Definition 4. *For the translation, we use the same symbols as declared by the user in AIF-ω plus the following:*

- *new untyped variables $E_{i,X}$ for $X \in \mathcal{V}_{\mathfrak{A}}$ and $1 \leq i \leq N$.*
- *a new function symbol val/N (where N is the number of families of sets)*
- *new fact symbols $isT_i/1$ for every user-defined type T_i,*
- *and finally the infix fact symbol $\twoheadrightarrow /2$.*

For an admissible AIF-ω rule

$$r(X_1\colon T_1, \ldots, X_m\colon T_m) = LF \cdot S_+ \cdot S_- \;\text{=}[F]\Rightarrow\; RF \cdot RS$$

define its translation into a Horn clause $[\![r]\!]$ as follows.

$$L_i(X) = \begin{cases} s_i(A_1, \ldots, A_n) & \text{if } X \in s_i(A_1, \ldots, A_n) \text{ occurs in } S_+ \\ 0 & \text{if } X \notin s_i(\_) \text{ occurs in } S_- \\ E_{i,X} & \text{otherwise} \end{cases}$$

$$R_i(X) = \begin{cases} s_i(A_1, \ldots, A_n) & \text{if } X \in s_i(A_1, \ldots, A_n) \text{ occurs in } RS \\ E_{i,X} & \text{otherwise, if } L_i(X) = E_{i,X} \text{ and } t \notin F \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} L(X) &= (L_1(X), \ldots, L_N(X)) \\ R(X) &= (R_1(X), \ldots, R_N(X)) \\ \lambda &= [X \mapsto val(L(X)) \mid X \in \mathcal{V}_{\mathfrak{A}}] \\ \rho &= [X \mapsto val(R(X)) \mid X \in \mathcal{V}_{\mathfrak{A}}] \\ C &= \{\lambda(X) \twoheadrightarrow \rho(X) \mid X \in \mathcal{V}_{\mathfrak{A}} \setminus F, \text{ and } \lambda(X) \neq \rho(X)\} \\ Types &= \{isT_i(X_i) \mid T_i \text{ is a user defined type}\} \\ [\![r]\!] &= Types \cdot \lambda(LF) \to \rho(RF) \cdot C \end{aligned}$$

where $\to$ is the "normal implication" in Horn clauses. We keep the set operator $\cdot$ from AIF-ω in our notation, denoting in Horn clauses simply conjunction. Finally, note that our Horn clauses have in general more than one fact as conclusion, but this is of course also just syntactic sugar.

We give the translation for the behavior of the honest agents in the running example (other rules are similar and shown in the appendix for completeness).

$$[\![keyReg]\!] = isUser(A) \cdot isServer(S) \to iknows(val(ring(A), db(S, A, valid)))$$

$$\begin{aligned} [\![userUpdateKey]\!] = {} & isHonest(A) \cdot isServer(S) \cdot iknows(val(ring(A), E_{db,PK})) \\ & \to iknows(sign(inv(val(0, E_{db,PK})), pair(A, val(ring(A), 0)))) \cdot \\ & (val(ring(A), E_{db,PK}) \twoheadrightarrow val(0, E_{db,PK})) \end{aligned}$$

$$\begin{aligned} [\![serverUpdateKey]\!] = {} & isUser(A) \cdot isServer(S) \cdot \\ & iknows(sign(inv(val(E_{ring,PK}, db(S, A, valid))), pair(A, val(E_{ring,NPK}, 0)))) \\ & \to iknows(inv(val(E_{ring,PK}, db(S, A, revoked)))) \cdot \\ & (val(E_{ring,PK}, db(S, A, valid)) \twoheadrightarrow val(E_{ring,PK}, db(S, A, revoked))) \cdot \\ & (val(E_{ring,NPK}, 0) \twoheadrightarrow val(E_{ring,NPK}, db(S, A, valid))) \end{aligned}$$

First note that all right-hand side variables of the Horn clauses also occur on the left-hand side; this is in fact the reason to introduce the typing facts like *isUser*. In fact, the variables of each Horn clause are implicitly universally quantified (e.g. in ProVerif) and we explicitly add these quantifiers when translating to SPASS. Thus, $[\![keyReg]\!]$ expresses that the intruder knows all those values (public keys) that are in the key ring of a user A and registered as valid for A at server S.

For the $[\![userUpdateKey]\!]$ rule, let us first look at the abstraction of the involved keys PK and NPK. We have $L(PK) = (ring(A), E_{db,PK})$ (we write the family name *db* rather than its index for readability) and $R(PK) = (0, E_{db,PK})$. This reflects that the rule operates on any key in the key ring of an honest agent A, where the variable $E_{db,PK}$ then is a placeholder for what status the key has in the database. The fact that in the original transition system, the key PK gets removed from the key ring when applying this rule, is reflected by the 0 component in the right-hand side abstraction: this is any key that is not in the key-ring but has the same status for *db* as on the left-hand side. Actually, in the

key update message that the agent produces for the signing key $inv(PK)$ it holds that PK is no longer in the key ring. The Horn clause reflects that: for every value in the ring of an honest user, the intruder gets the key update message with the same key removed from the key ring (but with the same membership in db). Finally, the $\twoheadrightarrow$ fact here intuitively expresses that everything that is true about an abstract value $val(ring(A), E_{db,PK})$ is also true about $val(0, E_{db,PK})$. We formally define this special meaning of $\twoheadrightarrow$ below.

3.2 Fixedpoint Definition

We define the fixedpoint for the Horn clauses in a standard way, where we give a special meaning to the $s \twoheadrightarrow t$ facts: for every fact $C[s]$ that the fixedpoint contains, also $C[t]$ must be contained. We see later how to encode this (and the typing facts) for existing tools like ProVerif and SPASS.

Definition 5. *Let*

- $Types = \{isT_i(c) \mid c \in [\![T_i]\!]$ *for every user-defined type* $T_i\}$.
- *For a set of ground facts* Γ, *let* $Timplies(\Gamma) = \{C[t] \mid s \twoheadrightarrow t \in \Gamma \wedge C[s] \in \Gamma\}$ *where* $C[\cdot]$ *is a context, i.e. a "term with a hole", and* $C[t]$ *means filling the hole with term* t.
- *For any Horn clause* $r = A_1 \dots A_n \to C_1 \dots C_m$, *define* $Apply(r)(\Gamma) = \{\sigma(C_i) \mid \sigma(A_1) \in \Gamma, \dots, \sigma(A_n) \in \Gamma, 1 \leq i \leq m\}$.

For a set of Horn clauses R, *we define the least fixed-point* $LFP(R)$ *as the least closed set* Γ *that contains Types and is closed under* $Timplies$ *and* $Apply(r)$ *for each* $r \in R$.

For our running example we can describe the "essential" fixedpoint as follows, for every $A \in [\![Honest]\!]$, $D \in [\![Dishon]\!]$ and $S \in [\![Server]\!]$:

$$
\begin{array}{l}
val(ring(A), 0) \twoheadrightarrow val(ring(A), db(S, A, valid)) \\
val(ring(A), 0) \twoheadrightarrow val(0, 0) \\
val(ring(A), db(S, A, valid)) \twoheadrightarrow val(0, db(S, A, valid)) \\
val(0, 0) \twoheadrightarrow val(0, db(S, A, valid)) \\
val(0, db(S, A, valid)) \twoheadrightarrow val(0, db(S, A, revoked)) \\
val(ring(D), 0) \twoheadrightarrow val(ring(D), db(S, D, valid)) \\
val(ring(D), db(S, D, valid)) \twoheadrightarrow val(ring(D), db(S, D, revoked)) \\
iknows(val(ring(A), 0)) \\
iknows(sign(inv(val(0, 0)), pair(A, val(ring(A), 0)))) \\
iknows(inv(0, db(S, A, revoked))) \\
iknows(val(ring(D), 0))iknows(inv(val(ring(D), 0)))
\end{array}
$$

Here, we have omitted the type facts, "boring" intruder deductions, and consequences of $\twoheadrightarrow$ (i.e., when $C[s]$ and $s \twoheadrightarrow t$ omit $C[t]$). Note that the $\twoheadrightarrow$ facts reflect the "life cycle" of the keys.

4 Soundness

We now show that the fixedpoint of Definition 5 represents a sound over-approximation of the transition system defined by an AIF-ω specification: if an attack state is reachable in the transition system, then the fixedpoint will contain the fact *attack*. The inverse is in general not true, i.e., we may have *attack* in the fixedpoint while the transition system has no attack state. However, soundness thus gives us the guarantee that the system is correct, if the fixedpoint does not contain *attack*. To show soundness we take several steps:

- We first annotate in the transition system in every state all occurring constants $c \in \mathfrak{A}$ with the equivalence class that they shall be abstracted to.
- We then give a variant of the rules that correctly handles these labels.
- We can then eliminate all set conditions $s \in \ldots$ and $s \notin \ldots$ from the transition rules and states, since this information is also present in the labels.
- Finally, we show for any fact that occurs in a reachable state, the fixedpoint contains its abstraction (i.e., replacing any labeled concrete constant with just its label).

Note that the first three steps are isomorphic transformations of the state transition system, i.e., we maintain the same set of reachable states only in a different representation.

4.1 The Labeled Concrete Model

The basic idea of our abstraction is that every constant $c \in \mathfrak{A}$ shall be abstracted by what sets it belongs to, i.e., two constants that belong to exactly the same sets will be identified in the abstraction. The first step is that in every reachable state S, we shall label every occurring constant $c \in \mathfrak{A}$ with this equivalence class. Note that upon state transitions, the equivalence class of a constant can change, since its set memberships can.

Definition 6. *Given a state S and a constant $c \in \mathfrak{A}$ that occurs in S. Then the N-tuple $(e_1, \ldots, e_N)$ is called the* correct label of c in S *if for every $1 \leq i \leq N$ either*

- *$e_i = 0$ and $c \in s_i(a_1, \ldots, a_n)$ does not occur in S for any $a_1, \ldots, a_n$, or*
- *$e_i = s_i(a_1, \ldots, a_n)$ and $c \in s_i(a_1, \ldots, a_n)$ occurs in S and $c \in s_i(a'_1, \ldots, a'_n)$ does not occur in S for any $(a'_1, \ldots, a'_n) \neq (a_1, \ldots, a_n)$.*

We write $c@l$ for constant c annotated with label l.

Note that, at this point, the label is merely an annotation and it can be applied correctly to every constant $c \in \mathfrak{A}$ in every reachable state S, because by Lemma 1, c can never be in more than one set of the same family, i.e., $c \in s_i(a_1, \ldots, a_n)$ and $c \in s_i(a'_1, \ldots, a'_N)$ cannot occur in the same state S.

4.2 Labeled Transition Rules

While, in the previous definition, the labels are just an annotation that decorate each state, we now show that we can actually modify the transition rules so that they "generate" the labels on the right-hand side, and "pattern match" existing labels on the left-hand side.

Definition 7. *Given an AIF-ω rule r we define the* corresponding labeled rule *r' as the following modification of r:*

- *Every variable $X \in \mathcal{V}_{\mathfrak{A}}$ on the left-hand side is labeled with $L(X)$ and every variable $X \in \mathcal{V}_{\mathfrak{A}}$ on the right-hand side (including the fresh variables) is labeled with $R(X)$.*
- *All variables $E_{i,X}$ that occur in $L(X)$ and $R(X)$ are added to the rule parameters of r'.*
- *For each variable $X \in \mathcal{V}_{\mathfrak{A}}$ that occurs both on the left-hand side and the right-hand side and where $L(X) \neq R(X)$, we augment r' with the* label modification *$X@L(X) \mapsto X@R(X)$.*

The semantics of r' is defined as follows. First, the labeling symbol @ is not treated as a mere annotation anymore, but as a binary function symbol (so labels are treated as a regular part of terms, including variable matching on the left-hand side). To define the semantics of the label modifications, consider a rule

$$r' = r'_0 \cdot (X_1@l_1 \mapsto X_1@r_1) \cdot \ldots \cdot (X_n@l_n \mapsto X_n@r_n)$$

where r'_0 is the basis of r' that does not contain label modifications. We define $S \Rightarrow^{\sigma}_{r'} S'$ iff $S \Rightarrow^{\sigma}_{r'_0} S'_0$ and S' is obtained from S'_0 by replacing every occurrence of $\sigma(X_i@l_i)$ with $\sigma(X_i@r_i)$ for $i = 1, 2, \ldots, n$ (in this order).

Note that the order $i = 1, 2, \ldots, n$ does not matter: the distinguished semantics requires that all distinct variables $X, X' \in \mathcal{V}_{\mathfrak{A}}$ have $\sigma(X) \neq \sigma(X')$ and therefore the label replacements are on disjoint value-label pairs.

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

$$\begin{aligned}
&\mathit{userUpdateKey}'(A\colon \mathit{Honest}, S\colon \mathit{Server}, \mathit{PK}\colon \mathit{Value}, \mathit{NPK}\colon \mathit{Value}) = \\
&\quad \mathit{PK}@(\mathit{ring}(A), E_{db,X}) \in \mathit{ring}(A) \cdot \mathit{iknows}(\mathit{PK}@(\mathit{ring}(A), E_{db,X})) \\
&\quad =[\mathit{NPK}@(\mathit{ring}(A), 0)]\Rightarrow \mathit{NPK}@(\mathit{ring}(A), 0) \in \mathit{ring}(A) \cdot \\
&\quad \mathit{iknows}(\mathit{sign}(\mathit{inv}(\mathit{PK}@(0, E_{db,X})), \mathit{pair}(A, \mathit{NPK}@(\mathit{ring}(A), 0)))) \cdot \\
&\quad (\mathit{PK}@(\mathit{ring}(A), E_{db,X}) \mapsto \mathit{PK}@(0, E_{db,X}));
\end{aligned}$$

Lemma 2. *Given a set R of AIF-ω rules, and let R' be the corresponding labeled rules. Then R' induces the same state space as R except that all states are correctly labeled.*

Proof. This requires two induction proofs, one showing that every R-reachable state has its R'-reachable correspondent. The other direction, that every R'-reachable state has an R-reachable correspondent is similar and actually not necessary for the overall soundness, so we omit it here.

For the initial state $\emptyset$, the statement is immediate. Suppose now S_1 is an R-reachable state, S_1' is an R'-reachable state where S_1' is like S_1 but correctly labeled. Suppose further $S_1 \Rightarrow_r^\sigma S_2$ for some $r \in R$, some substitution σ, and some successor state S_2. We show that the corresponding rule $r' \in R'$ allows for a transition $S_1 \Rightarrow_{r'}^{\sigma'\tau} S_2'$ where S_2' is the correctly labeled version of S_2 and some substitutions σ' and τ.

The substitution σ' here is an adaption of σ, because untyped variables are substituted for terms that can contain constants from $\mathfrak{A}$ that are labeled in S_1' but unlabeled in S_1. In fact, this label may even change upon transition, in this case, σ' contains the label of S_1'. Thus, σ and σ' only differ on untyped variables.

The substitution τ is for all variables $E_{i,X}$ that occur in the label variables of r'. We show the statement for the following choice of τ: for each label variable $E_{i,X}$ that occurs in r' (where by construction $X \in \mathcal{V}_\mathfrak{A}$ is a variable that occurs in r' and $1 \leq i \leq N$), we set $\tau(E_{i,X}) = e_i(X)$ if $e(X) = (e_1(X), \ldots, e_N(X))$ is the correct label of $\sigma(X)$ in S_1. Note that τ is a grounding substitution and does not interfere with σ or σ'.

To prove that $S_1 \Rightarrow_{r'}^{\sigma'\tau} S_2'$, we first consider the matching of r on S_1 and r' on S_1'. We have to show that despite the additional labels, essentially the same match is still possible. Consider thus any variable $X \in \mathcal{V}_\mathfrak{A}$ that occurs on the left-hand side of r and thus $X@L(X)$ occurs correspondingly on the left-hand side of r'. We have to show that the correct label for $\sigma(X)$ in S_1 is indeed $\sigma(\tau(L(X)))$. For $1 \leq i \leq N$, we distinguish three cases:

- $L_i(X) = s_i(A_1, \ldots, A_n)$, then S_+ of r contains the positive set condition $X \in s_i(A_1, \ldots, A_n)$ and thus $\sigma(X \in s_i(A_1, \ldots, A_n))$ occurs in S_1. Thus $\sigma(\tau(L_i(X))) = \sigma(s_i(A_1, \ldots, A_n))$ is the i-th part of the correct label of $\sigma(X)$.
- $L_i(X) = 0$, then S_- of r contains the negative set condition $X \notin s_i(\_)$ and thus $\sigma(X) \in s_i(a_1, \ldots, a_n)$ does not occur in S_1 for any a_i. Thus $\sigma(\tau(L_i(X))) = 0$ is the i-th part of the correct label of $\sigma(X)$.
- $L_i(X) = E_{i,X}$. In this case the rule neither requires nor forbids X to be member of some set of family s_i. Since $\tau(E_{i,X}) = e_i(X)$ where $e_i(X)$ is i-th component of the correct label for $\sigma(X)$, we have that $\sigma(\tau(L_i(X))) = e_i(X)$ is the i-th component of the correct label for $\sigma(X)$.

Thus in all cases, $\sigma(\tau(L(X))$ is the correct label for $\sigma(X)$ in S_1. Since S_1' is correctly labeled, all occurrences of $\sigma(X)$ in S_1' are labeled $\sigma(\tau(L(X))$, and thus the rule r' is applicable to S_1' under $\sigma'\tau$ (where σ' adapts to labels in the substitution of untyped variables). It remains to show that under this match we obtain the desired successor state S_2'.

To that end, we first show that for any variable $X \in \mathfrak{A}$ that occurs in the right-hand side of r, $\sigma(\tau(R(X))$ is the correct label for $\sigma(X)$ in S_2. For $1 \leq i \leq N$, we distinguish three cases:

- $R_i(X) = s_1(A_1, \ldots, A_n)$. Then $X \in s_i(A_1, \ldots, A_n)$ occurs in RS of r and thus $\sigma(X \in s_i(A_1, \ldots, A_n))$ is in S_2. Thus $\sigma(\tau(R_i(X))) = \sigma(s_i(A_1, \ldots, A_n))$ is the i-th component of the correct label for $\sigma(X)$ in S_2.

- $R_i(X) = 0$. Then either $X \in s_i(\ldots)$ occurs in S_+ or $X \in s_i(\_)$ occurs in S_-, or X is a fresh variable, and but $X \in s_i(\ldots)$ does not occur in RS, so $\sigma(X \in s_i(a_1, \ldots, a_n))$ is not contained in S_2 for any a_j, and therefore $\sigma(\tau(R_i(X))) = 0$ is the i-th component of the correct label for $\sigma(X)$ in S_2.
- $R_i(X) = E_{i,X}$. Then the set membership of X with respect to family s_i does not change on the transition, and $\sigma(\tau(R_i(X))) = e_i(X)$ is the correct label for $\sigma(X)$ also in S_2.

Thus in all cases, $\sigma(\tau(R(X)))$ is the correct label for $\sigma(X)$ in S_2. Finally, the label replacements of r' ensure that for all $c@l$ that occur in S_1' and where the label of c has changed upon transition to S_2' to label l' will be updated. Thus S_2' is the correctly labeled version of S_2. □

4.3 Labeled Concrete Model without Set Conditions

Since every label correctly represents the set memberships of the involved constants, we can just do without set membership facts, i.e., remove from the labeled rules the S_+, S_- and RS part. We obtain states that do not contain any $s \in s_i(\cdot)$ conditions anymore, but only handle this information in the labels of the constants. It is immediate from Lemma 2 that this changes the model only in terms of representation:

Lemma 3. *The labeled model without set conditions has the same reachable states as the labeled model, except that states have no more explicit set conditions.*

4.4 Reachable Abstract Facts

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 8.

$$\eta(t@(e_1, \ldots, e_N)) = val(e_1, \ldots, e_N) \text{ for } t \in \mathfrak{A} \cup \mathcal{V}_{\mathfrak{A}}$$
$$\eta(f(t_1, \ldots, t_n)) = f(\eta(t_1), \ldots, \eta(t_n))$$
$$\text{for any function or fact symbol } f \text{ of arity } n$$

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

Lemma 4. *For an AIF-ω rule set R, let R' be the corresponding rule set in the labeled model without, f be a fact in a reachable state of R' (i.e. $\emptyset \rightarrow_{R'}^* S$ and $f \in S$ for some S). Let $[\![R]\!]$ be the translation into Horn clauses of the rules R according to Definition 4, and $\Gamma = LFP([\![R]\!])$. Then $\eta(f) \in \Gamma$.*

This lemma is simply adapting the corresponding result for AIF [10], which we omit due to the lack of space. From Lemmata 2, 3 and 4 immediately follows that the over-approximation is sound:

Theorem 1. *Given an AIF-ω specification with rules R. If an attack state is reachable with R, then attack $\in LFP(\llbracket R \rrbracket)$.*

5 Encoding in SPASS and ProVerif

We now want to use SPASS and ProVerif for checking the property $attack \in LFP(\llbracket R \rrbracket)$. Three aspects in our definition of LFP need special considerations.

First, SPASS is a theorem prover for standard first-order logic FOL (and the Horn clause resolution in ProVerif is very similar, but more geared towards protocol verification). The problem here is that the Horn clauses $\llbracket R \rrbracket$ always have the trivial model where the interpretation of all facts is set simply to true, and in this model, *attack* holds. We are interested in the "least" model and terms to be interpreted in the Herbrand universe, i.e., the free term algebra $\mathcal{T}_\Sigma$. The common "Herbrand trick" is to try to prove the FOL formula $\llbracket R \rrbracket \implies attack$, i.e., that in *every* model of the Horn clauses, *attack* is true. If that is valid, then also in the least Herbrand model, *attack* is true. Vice-versa, if the formula is not valid, then there are some models in which *attack* does not hold, and then also in the least Herbrand model. This trick is also part of the setup of ProVerif.

The second difficulty is the encoding of the user-defined types. For instance, the declaration $A = \{\ldots\}$ leads to the extension $\llbracket A \rrbracket = \{a_n \mid n \in \mathbb{N}\}$ for some new constant symbols a_n, and then by definition, $LFP(\llbracket R \rrbracket)$ contains the infinite set $\{isA(a_n) \mid n \in \mathbb{N}\}$. We could encode this by Horn clauses

$$isA(mkA(0)) \quad \wedge \quad \forall X.isA(mkA(X)) \rightarrow isA(mkA(s(X)))$$

for new function symbols mkA and s. Note that this encoding only makes sense in the least Herbrand model (standard FOL allows to interpret s as the identity). However, it easily leads to non-termination in tools. A version that works however, is simply saying $\forall X.isA(mkA(X))$. Interpreting this in the least Herbrand model, X can be instantiated with any term from $\mathcal{T}_\Sigma$ (which is countable).

The third and final difficulty are the $\twoheadrightarrow$ facts that have a special meaning in $LFP(\llbracket R \rrbracket)$: whenever both $C[s]$ and $s \twoheadrightarrow t$ in $LFP(\llbracket R \rrbracket)$ then also $C[t]$ (for any context $C[\cdot]$). We can encode this into Horn clauses because we can soundly limit the set of contexts $C[\cdot]$ that need to be considered: it is sufficient to consider right-hand side facts of rules in R in which a variable $X \in \mathcal{V}_{\mathfrak{A}}$ occurs (note that a fact may have more than one such occurrence). Define thus the finite set $Con = \{C[\cdot] \mid C[X] \text{ is a RHS fact in } R, X \in \mathcal{V}_{\mathfrak{A}}\}$. We generate the additional Horn clauses: $\{\forall X, Y.C[X] \wedge (X \twoheadrightarrow Y) \rightarrow C[Y] \mid C[\cdot] \in Con\}$.

Lemma 5. *The encoding of $\twoheadrightarrow$ into Horn clauses is correct.*

Proof. Suppose $C[s]$ and $s \twoheadrightarrow t$ are in the fixedpoint. Then $C[s]$ is the consequence of some Horn clause $A \rightarrow B$, i.e., $C[s] = \sigma(B)$ such that $\sigma(A)$ is part of the fixedpoint. We distinguish two cases. First $B = C'[X]$ for some $X \in \mathcal{V}_{\mathfrak{A}}$, some context $C'[\cdot]$ and $\sigma(C'[\cdot]) = C[\cdot]$, i.e., $s = \sigma(X)$ is directly the instance of a $\mathcal{V}_{\mathfrak{A}}$ variable, and thus our Horn clause encoding covers $C[\cdot]$. Second, the only

	Number of Agents			Backend	
	Honest	Dishon	Server	ProVerif	SPASS
AIF	1	1	1	0.025s	0.891s
	2	1	1	0.135s	324.696s
	2	2	1	0.418s	Timeout
	3	3	1	2.057s	Timeout
AIF-ω	ω	ω	ω	0.034s	0.941s

Table 1. AIF vs. AIF-ω on the key-server example.

other possibility for $\sigma(B) = C[s]$ is that B contains an untyped variable that matches s or a super-term of it in $C[s]$. By the rule shape, this untyped variable is also part of the assumptions A. Since $\sigma(A)$ is already in the fixedpoint, we can by an inductive argument conclude that for every $C_0[s]$ of $\sigma(A)$ also $C_0[t]$ is in the fixedpoint. In both cases, we conclude that $C[t]$ is derivable. □

5.1 Experimental Results

Table 1 compares the run times of our key-server example for AIF and AIF-ω, taken on a 2.66 GHz Core 2 Duo, 8 GB of RAM. In AIF we have to specify a fixed number of honest and dishonest users and servers, while in AIF-ω we can have an unbounded set for each of them (denoted ω in the Figure). Observe that in AIF the run times "explode" when increasing the number of agents. It is clear why this happens when looking at an example: when we specify the sets $ring(User)$ in AIF, $User$ needs to be a finite set (the honest and dishonest users) and this gets first translated into n different sets when we specify n users. Since these sets are by construction all disjoint, we can specify in AIF-ω instead $ring(User!)$ turning the n sets of the AIF abstraction into a single family of sets in the AIF-ω abstraction—and then allowing even for a countably infinite set $User$. Observe that the run times for AIF-ω for infinitely many agents are indeed very close to the ones of AIF with one agent. Thus, even when dealing with finitely many agents, AIF-ω allows for a substantial improvement of performance whenever we can exploit the uniqueness, i.e., can specify $set(Type!)$ instead of $set(Type)$.

The key server is in fact our simplest example, a kind of "NSPK" of stateful protocols. We updated our suite of case studies for AIF to benefit in the same way from the AIF-ω extension [3]. These include the ASW protocol (one of the original motivations for developing AIF and also for extending it to AIF-ω), an in-depth analysis of the Secure Vehicle Communication protocols SEVECOM [11], and a model of and analysis of PKCS#11 tokens [9] that replicates the attacks reported in [6] and verifies the proposed fixes.

6 Conclusions

In this paper we introduced the language AIF-ω and showed how it can be used to model cryptographic systems with unbounded numbers of agents and

databases pertaining to these agents. AIF-ω extends our previous language AIF by introducing types with countably infinite constants and allowing families of sets to range over such types. The only requirement to this extension is that the sets of all infinite families are kept pairwise disjoint.

We defined the semantics of this extension and proposed an analysis technique that translates AIF-ω models into Horn clauses, which are then solved by standard off-the-shelf resolution-based theorem provers. We proved that our analysis is sound w.r.t. the transition system defined by the semantics: if an attack is reachable, then it is also derivable in the Horn clause model.

Finally, the experimental results show that the clauses produced by AIF-ω, for the protocol with unbounded agents, can be solved in running times similar to their AIF counterparts for just one agent of each type, both in ProVerif and SPASS. In contrast, adding agents to the bounded AIF model produces an exponential increase in running times.

To our knowledge, this is the first work that proposes a fully automated technique for analyzing stateful cryptographic protocols with unbounded agents. This work is a direct extension of our previous work on AIF [10], which allows to model infinite transition systems with bounded agents. Its relation with AIF-ω has been extensively described throughout this paper. Another work that uses the set-abstraction is our work on Set-π [4], which extends the Applied π-calculus by similarly introducing a notion of a fixed number of sets. Set-π presents a modeling interface that is more familiar to the user, and the process-calculus specification exposes a great deal of details (e.g. locking, replications) that are abstracted away by the AIF rules. This reduces the gap to the system implementation, but as a modeling language Set-π has essentially the same expressive power of AIF. We believe that a similar extension can be devised for our process algebraic interface, possibly using AIF-ω as an intermediate representation.

Another related work is StatVerif [1], which extends the Applied π-calculus with global cells that can be accessed and modified by the processes. As the number of cells is finite and fixed for a model, the amount of state that one can finitely represent is limited. However, the particular encoding of StatVerif is more precise in capturing state transitions synchronized over multiple cells. We claim that the two approaches are orthogonal, and we have not succeeded so far in combining the advantages of both with an encoding into Horn clauses.

The Tamarin prover [8] and its process calculus interface SAPIC [7] use multiset rewriting rules to describe cryptographic protocols, and a semi-automated search procedure to find a solution for the models. This formalism is very expressive and allows to prove security properties in stateful protocols with unbounded agents, but expressiveness comes at the price of usability, as the search procedure needs to be guided by introducing lemmas in the models.

Finally, we believe that bounded-agents results like [5] can be also derived for AIF-ω, since the resolution will never run into distinguishing single agents. The experimental results, however, suggest that for our verification it is more efficient to avoid the enumeration of concrete agents where possible.

The authors would like to thank Luca Viganò for the helpful comments.

References

1. Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark Dermot Ryan. Statverif: Verification of stateful processes. *Journal of Computer Security*, 22(5):743–821, 2014.
2. Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Computer Security Foundations Workshop*, 2001.
3. Alessandro Bruni and Sebastian Mödersheim. The AIF-ω Compiler and Examples. `http://www.imm.dtu.dk/~samo/aifom.html`.
4. Alessandro Bruni, Sebastian Mödersheim, Flemming Nielson, and Hanne Riis Nielson. Set-pi: Set membership p-calculus. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015*, 2015.
5. Hubert Comon-Lundh and Véronique Cortier. Security properties: two agents are sufficient. *Sci. Comput. Program.*, 50(1-3):51–71, 2004.
6. Sibylle B. Fröschle and Graham Steel. Analysing pkcs#11 key management apis with unbounded fresh data. In *ARSPA-WITS 2009*, 2009.
7. Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *Security and Privacy*, 2014.
8. Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification*, 2013.
9. Ming Ye. Design and Analysis of PKCS#11 key management With AIF. Master's thesis, DTU Compute, 2014. Available at `www2.compute.dtu.dk/~samo`.
10. Sebastian Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *Computer and Communications Security*, 2010.
11. Sebastian Mödersheim and Paolo Modesti. Verifying sevecom using set-based abstraction. In *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference, IWCMC 2011, Istanbul, Turkey, 4-8 July, 2011*, 2011.
12. Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Conference on Automated Deduction*, 1999.
13. Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *22nd International Conference on Automated Deduction, CADE 2009*, 2009.