Verification Based on Set-Abstraction Using the AIF Framework

Mödersheim, Sebastian Alexander

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

Verification Based on Set-Abstraction Using the AIF Framework*

Version 1

Sebastian Mödersheim, DTU Informatics
samo@imm.dtu.dk

October 1, 2010

Abstract

The AIF framework is a novel method for analyzing advanced security protocols, web services, and APIs, based a new abstract interpretation method. It consists of the specification language AIF and a translation/abstraction processes that produces a set of first-order Horn clauses. These can then be checked with several back-ends such as SPASS. We discuss in this article how to use AIF for modeling of a variety of examples.

1 Introduction

This article is a supplement for the paper [2] which introduces the AIF framework and abstraction method. Here, we present in more detail two major case studies we have done with AIF: a fair exchange system and parts of a secure vehicle communication system. The tool, the original paper, and the AIF-source code of the examples can be obtained from the author's webpage `www.imm.dtu.dk/~samo` . We assume at this point that the reader is familiar with the paper and not give an introduction to the method here.

This document is work in progress—hence we give a version number in the title—and we plan to extend this document with more case studies in the future. Questions, feedback, and contributions from interested readers are most welcome!

2 Fair Exchange: Asokan Shoup Waidner

The largest example, and in fact one of the original motivations for the set-abstraction idea, is the contract signing protocol ASW based on optimistic fair-exchange [1].

*The author thanks Paolo Modesti and Graham Steel for helpful comments.

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. In the first round of the main protocol, the contract partners exchange a signature on the contractual text that contains a hash of a nonce from each side. These signatures alone do not make a valid contract yet. In the second round of the main protocol, the agents shall exchange the nonces that they have chosen. Knowing the signatures from the first round and both nonces gives a valid *standard contract*. The relation to abort and resolve procedures are discussed later.

Sets 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. More in detail we have the following status of nonces for the honest agents and the server:

```
% Status that a nonce can have from the server's point of view
SSts     : {aborted,valid};

% Status that a nonce can have from a user's point of view
Sts      : {hashed,revealed,aborted,valid};
```

The status `hashed` is for nonces in the first round of the main protocol, `revealed` for the second round, `aborted` for nonces from an aborted run, and `valid` for nonces in a valid standard or replacement contract.

We consider the following sets/databases of nonces:

```
 idb(D),% intruders database
 condb(HA,B,Sts),% database of an honest user:
  % - name of the user,
% - name of the other user with whom the contract is
% - status of the respective exchange
scondb(A,B,SSts);       % server's database:
% - names of the two users that want to make a contract
% - status of the respective exchange
```

Observe that there is no field for the "contract". We currently use simply $contract(A, B)$ to denote a contract between A and B. In fact, we thus use the

contractual text as an abstract constant that is the same in all sessions of A and B for simplicity.

Main Protocol We abstract here from the concrete message format with signatures and the like and instead use new, uninterpreted function symbols `msg1/4` (the parameters are used for: sender, receiver, contract, hashed-nonce) and `msg2/3` (the parameters are used for: sender, repeating message 1, hashed-nonce).

We define for these message constructors the following intruder rules:

```
\D,HB.iknows(M1).iknows(M2) => iknows(msg1(D,HB,M1,M2));
\HA,B.iknows(msg1(HA,B,M1,M2)) => iknows(M1).iknows(M2);

\D.iknows(M1).iknows(M2) => iknows(msg2(D,M1,M2));
\HA.iknows(msg2(HA,M1,M2)) => iknows(M1).iknows(M2);
```

Here and elsewhere, D ranges over dishonest, HA and HB over honest users.

The main protocol now consists of the following rule (A and B ranging over all agents):

```
% Round 1, Alice sends signature with hashed nonce
\HA,B.
=[NA]=>
iknows(msg1(HA,B,contract(HA,B),h(NA))).
NA in condb(HA,B,hashed);

% Round 1: Bob answers with own hashed nonce
\A,HB.
iknows(msg1(A,HB,contract(A,HB),HNA)).
=[NB]=>
iknows(msg2(HB,msg1(A,HB,contract(A,HB),HNA),h(NB))).
NB in condb(HB,A,hashed);

% Round 2: Alice reveals her nonce
\HA,B.
iknows(msg2(B,msg1(HA,B,contract(HA,B),h(NA)),HNB)).
NA in condb(HA,B,hashed)
=>
iknows(NA).NA in condb(HA,B,revealed);

% Round 2: Bob reveals his nonce
\A,HB.
iknows(msg2(HB,msg1(A,HB,contract(A,HB),h(NA)),h(NB))).
iknows(NA).
NB in condb(HB,A,hashed)
=>
iknows(NB).NB in condb(HB,A,valid);
```

```
% Round 2: Alice receives Bobs answer
\HA,B.
iknows(msg2(B,msg1(HA,B,contract(HA,B),h(NA)),h(NB))).
NA in condb(HA,B,revealed).iknows(NB)
=>
NA in condb(HA,B,valid);
```

Resilient Channels 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. This is the rule for an honest Alice having not received an answer to the first round of the main protocol:

  ```
  \HA,B.
  iknows(msg1(HA,B,contract(HA,B),h(NA))).
  NA in condb(HA,B,hashed).
  forall A,B,SSts. NA notin scondb(A,B,SSts)
  =>
  NA in condb(HA,B,aborted).
  NA in scondb(HA,B,aborted);
  ```

- The next case concerns that a dishonest agent asks for an abort of a contract that the server has not yet seen. This is a variant of the previous rule where we just do not have to maintain the user-database (because the intruder knowledge is not organized in data-bases anyway).

  ```
  %% (c) Dishonest Initiator, Unknown to server

  \D,HB.
  iknows(msg1(D,HB,contract(D,HB),h(NA))).
  forall A,B,SSts. NA notin scondb(A,B,SSts)
  =>
  NA in scondb(D,HB,aborted);
  ```

- The next case is when honest Alice asks for an abort but the respective contract is already aborted or resolved at the server. Then Alice's status is appropriately updated:

```
\HA,B,SSts.
iknows(msg1(HA,B,contract(HA,B),h(NA))).
NA in condb(HA,B,hashed).
NA in scondb(HA,B,SSts)
=>
NA in condb(HA,B,SSts).
NA in scondb(HA,B,SSts);
```

- Note that we do not have any corresponding rule for a dishonest agent, as the server's status cannot be changed by this abort request.

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.

We do not list here the transitions for the resolve protocol.

Goals 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.

```
\HA,B.
iknows(msg2(B,msg1(HA,B,contract(HA,B),h(NA)),h(NB))).
iknows(NA).
iknows(NB).
NA in condb(HA,B,aborted)
=>
attack;
```

3 Secure Vehicle Communication

Our formalization is based on the description of the SEVECOM system [3] and a first analysis in [4]. We assume here that the reader is familiar with the system as described in the referred articles.

3.1 Unbounded Key Model

The SEVECOM system works with long-term and short-term keys that can be updated. Similar to the simple key-server example from [2], our AIF model of SEVECOM is based on sets of keys. Due to the construction of AIF, it is impossible to control how many elements such a set has, because cardinality constraints would not be compatible with the underlying abstraction approach.

In principle, every set can contain an arbitrary number of keys. In contrast, the real SEVECOM system has only a fixed number of keys, namely one or two keys per vehicle and key kind. Of course, allowing an unbounded number of keys is a sound over-approximation: the real system is obtained by restricting the behavior of our model. For some questions this over-approximation is too coarse and we will later refine for analyzing the root key update.

Like [4], we consider only one single vehicle/HSM in this first model with AIF.

The Sets The HSM of the car stores two root public keys K_1 and K_2 and the authority who controls the corresponding private keys can revoke either of them by issuing e.g. for K_1:

$$\mathsf{sign}(\mathsf{inv}(K_1), \mathsf{revoke}, K_1, T) \tag{1}$$

where T is a timestamp. Note that in the formal models, the tag revoke and the timestamp may be omitted (we discuss the implications later). Then, a new root key can be introduced using the other root key:

$$\mathsf{sign}(\mathsf{inv}(K_2), \mathsf{update}, K, T) \tag{2}$$

where K is a new public key.

We can generalize this to a system where we have initially an arbitrary number of root keys, any valid key can be revoked, and any valid key can be used to introduce a new key in place of a revoked one.

The car's HSM thus has the following database of keys:

$$db(Perm, Removable, Updating)$$

where

$$\begin{array}{rcl} Perm & : & \{sig, verify, decrypt\} \\ Removable & : & \{lt, st\} \\ Updating & : & \{tr, fl\} \end{array}$$

Thus, key are organized by permission (for signing, verification, decryption; these permissions are used in an exclusive way by the SEVECOM system), whether they are removable or not, and by their update status, i.e. whether they are currently under update or not. The database stores only public-keys explicitly. For keys of type *sig* and *decrypt* also the corresponding private keys are present on the device, but this is implicit in our model.

Besides the standard operations for the intruder such as generating key-pairs

```
=[K]=> iknows(K).iknows(inv(K)). K in dishonest();
```

(that are then also registered as belonging to a dishonest agent for use in goals later), the main focus are the operations of the HSM.

Goals We consider three goals:

Secrecy The intruder never knows the private key of a valid public key.

```
% The intruder gets hold of a private key
\ Perm,Removable,Updating.
K in db(Perm,Removable,Updating).
iknows(inv(K))
=>
attack;
```

Authentication All valid keys have been generated by the party that should, i.e., short-term keys by the HSM itself, long-term keys by the authority. In particular, the intruder shall not be able to install self-generated keys, but this is also covered by the secrecy goal because he knows the corresponding private key in this case.

```
% The intruder has inserted one of his keys:
\ Perm,Removable,Updating.
K in db(Perm,Removable,Updating).
K in dishonest(i).
iknows(K)
=>
attack;
```

Freshness The intruder shall not be able to introduce old keys into the HSM, even if they were once created by the correct party and the intruder does not know the private key. We want this property to prevent that older messages signed by these keys could be accepted again by the HSM.

To that end, we introduce a set *revoked*() (that is not visible to any participant) and to which we add every key that was a valid root key and has been revoked. Then it is an attack, if there is a valid root key that is part of revoked, because the intruder has reintroduced it.

```
% The intruder can make the TRD accept an old key:
\ Perm,Removable,Updating.
K in revoked(i).
K in db(Perm,st,Updating).
iknows(K)
=>
attack;
```

Observe that these goals have similarity with classic goals of secure communication, but adapted to the specific problem at hand.

Initialization We model the initialization of long-term and short-term keys for verification, signing and decryption:

```
=[K]=> iknows(K).K in db(verify,lt,fl);
=[K]=> iknows(K).K in db(sig,lt,fl);
=[K]=> iknows(K).K in db(decrypt,lt,fl);
=[K]=> iknows(K).K in db(sig,st,fl);
=[K]=> iknows(K).K in db(decrypt,st,fl);
```

This gives any number of fresh keys of all kinds (verification keys only long-term) where the "under-update"-flag is never set. Note that these rules can be applied at any time; this is again unrealistic (the HSM can only be operated via its API as soon as the car is on the road) but a sound over-approximation (it does not hurt if that were possible).

Normal use of keys The HSM can now be used for decrypting and signing messages via the API:

```
% Decrypt with API
\ Removable,Updating.
iknows(crypt(K,M)).K in db(decrypt,Removable,Updating)
=>
iknows(M).          K in db(decrypt,Removable,Updating);

% Sign with API
\ Removable,Updating.
iknows(K).iknows(M).   K in db(sig,Removable,Updating)
=>
iknows(sign(inv(K),M)).K in db(sig,Removable,Updating);
```

Root Key Update From the point of view of the HSM, the update of root keys is modeled by the following operations:

```
% Revoke a root key
iknows(sign(inv(K),K)).
K in db(verify,lt,fl)
=>
K in db(verify,lt,tr);

% Update a root key
iknows(sign(inv(K1),K)).iknows(K2).
K1 in db(verify,lt,fl).
K2 in db(verify,lt,tr)
=>
K1 in db(verify,lt,fl).
K  in db(verify,lt,fl).
K2 in revoked();
```

The first rule says that, when a revoke message comes in[1], i.e. a key K signed by $inv(K)$ and K is a valid root key not under update, then K is now under update for the HSM.

The second rule says that if $K1$ and $K2$ are root keys, $K1$ is valid, and $K2$ is under update, and a new key signed by $inv(K1)$ comes in, then K replaces $K2$ as a new valid root key.

As a first shot, we can easily verify with ProVerif and SPASS within a second, that the intruder cannot find out any private keys that belong to valid public keys. While this is trivial to see for keys that are generated with the initialization rules, the question is more interesting for the key-update, because potentially an intruder may be able to insert keys he created himself.

We can also check that there is an attack when the intruder is given the private key of a valid root key, because that means in the associated abstract model that the intruder knows all such private keys. To differentiate here, we need to consider indeed the more fine-grained model below.

Modeling the Authority So far, our model has only the HSM and the intruder, but no model of the authority that actually produces the revoke or update commands for the API we have just covered. Adding this model reveals several problems of the root key update which were missed by [4] because this work did not model the authority at all. As a consequence, in the model of [4], the revoke/update commands can only originate from the intruder and only if he is explicitly given one or both of the root keys.

Let us first consider a model where the authority that can generate a pair of revoke-update commands at any time non-deterministically:

```
K1 in db(verify,lt,fl).
K2 in db(verify,lt,fl).
=[K]=>
iknows(sign(inv(K1),K1)).
iknows(sign(inv(K2),K)).
K1 in db(verify,lt,fl).
K2 in db(verify,lt,fl);
```

Here, the authority takes any two valid keys $K1$ and $K2$, generates a new key K and issues a revoke for $K1$ and an update to K.

Replay This very "prolific" way to produce updates immediately leads to an attack for which the intruder does not even need to know any private root keys. Suppose we initially have two root keys installed on the HSM, called K_0 and K_1. Suppose further, the authority revokes-updates the key K_1 to K_2 and then from K_2 to K_3 and so on. This will produce update messages of the form $\mathsf{sign}(\mathsf{inv}(K_0), K_n)$. At each such update, the intruder can block the current update message and insert an old one, re-installing an older key K_i.

[1] We assume here the worst case that the intruder completely controls what the HSM sees and what not.

Such an attack is limited in practice by the use of timestamps (which are indeed part of the real messages and have been omitted in the formal models), but it shows that we need to limit the frequency of updates so that for messages of different updates the time window where the timestamps are acceptable for the HSMs do not overlap. Note that this time-window may have to be be quite large if one wants to "reach" all cars.

Reflecting the assumption in the formal model requires the integration of time into the specification. We want to leave this for future work and simply do not consider freshness goals for now.

Revoking the wrong key Consider now the case that the intruder has indeed learned one private key of a root key K_1. If the authority now happens to revoke the other key K_2, we have the following attack: the intruder passes the revocation of K_2 to the HSM, but instead of the update of K_2 to some other key, he uses his knowledge of $inv(K_1)$ to insert a new root key of his choice.

The attack is founded on the assumption that one key may leak to the intruder (but not both) and that the authority may falsely assume that the other key has been leaked. This could be prevented if for instance only one of the two keys is used for the day-to-day business (with a non-negligible risk of leakage) and the other one is kept elsewhere, with higher physical protection. Then the second key would only be used for a revoke-update of the first one when there is the suspicion that the first one has been leaked.Such a hierarchy between root keys is not indicated in the original SEVECOM description, but would indeed make sense also for the verification below.

Another way to prevent to mitigate this attack is to require that revocation and update messages must be signed by *both* root keys and the relevant public keys being part of the signed message.

Experiments This subsection refers to the following three files in the library:

sevecom1a.aif The described model with the replay goal (leading to an attack).

sevecom1b.aif Like (1a) without the replay goal (safe).

sevecom1c.aif Like (1b) giving the intruder the private root keys (in this model we cannot selectively give only one to the intruder). This has a trivial attack, but only SPASS can find it while ProVerif times out.[2]

3.2 A Finer Model

While the AIF specification describes a model without abstraction, the verification method does rely on abstraction. In the above model, all valid, updated

[2]We assume that ProVerif runs into a loop due to all the messages the intruder can now construct and misses the attack. In contrast, SPASS does terminate (after 4 minutes) if we disable all goals (so that attack is unreachable).

root keys are abstracted into the same equivalence class (because there is no difference in set membership) so we cannot selectively give the intruder access to just one of the root keys. We now give a finer, albeit slightly more complicated model where we can distinguish different root keys and give the intruder access to only one. Note that each "root key" is actually still an equivalence class that may contain an arbitrary number of elements).

The database is here for simplicity reduced to the root keys:

```
db(Root,Updating)
```

where $Root : \{root1, root2\}$ (and $Updating : \{updating, uptodate\}$. Besides that, we also have the set $dishonest()$ of intruder keys and $revoked()$ of revoked keys.

The operations of the HSM are as follows:

```
% Initialization
=[K1,K2]=>
K1 in db(root1,uptodate).
K2 in db(root2,uptodate).
iknows(K1).iknows(K2);

% Revoke root key i
\ Root.
iknows(sign(inv(K),K)).
K in db(Root,uptodate)
=>
iknows(K).
K in db(Root,updating);

% Update a root key 1
iknows(sign(inv(K2),K)).iknows(K1).
K2 in db(root2,uptodate).
K1 in db(root1,updating).
=>
iknows(K).iknows(K2).iknows(K).
K2 in db(root2,uptodate).
K  in db(root1,uptodate).
K1 in revoked();

% Update a root key 2
iknows(sign(inv(K1),K)).
iknows(K2).
K1 in db(root1,uptodate).
K2 in db(root2,updating).
=>
K1 in db(root1,uptodate).
K  in db(root2,uptodate).
K2 in revoked();
```

Again, the initialization can be performed at any time, registering an unbounded number of (pairs of) root keys. However, using the arguments `root1` and `root2`, we split the universe of keys into two partitions. This allows us to give the intruder access to just one partition. (The is again an over-approximation of the real system where each partition can only contain one key.)

Now we model that the authority updates root key number 1 and the intruder can getting hold of the corresponding private key.

```
% Authority revokes K1 and updates to new key K
K1 in db(root1,uptodate).
K2 in db(root2,uptodate).
iknows(K1).iknows(K2)
=[K]=>
K1 in db(root1,uptodate).
K2 in db(root2,uptodate).
iknows(sign(inv(K1),K1)).
iknows(sign(inv(K2),K));

% Intruder gets hold of inv of root key 1
K in db(root1,uptodate).
iknows(K)
=>
K in db(root1,uptodate).
iknows(inv(K));
```

Goals We again take the same goals as before, however restricting secrecy and freshness to the second root key.

```
% The intruder gets hold of a valid private key
\ Updating.
K in db(root2,Updating).
iknows(inv(K))
=>
attack;

% The intruder has inserted one of his keys:
\ Root,Updating.
K in db(Root,Updating).
K in dishonest().
iknows(K)
=>
attack;
```

```
% The intruder can make the HSM accept an old key:
\ Root,Updating.
K in revoked().
K in db(root2,Updating).
iknows(K)
=>
attack;
```

For the second goal, note that the intruder cannot even insert a key of his choice as root key 1, even though he knows a private root key. This is indeed a very positive result.

Experiments In the library the following files are relevant:

sevecom2a.aif The above formalization with two root keys, the intruder getting access only to key1. Safe.

sevecom2b.aif Giving the intruder both keys trivially leads to an attack. Again ProVerif times out.

In summary, the model of one root key being in day-to-day with the potential of leakage and revocation while the other one being absolutely safe is indeed free of attacks.

We have also verified that the above goals hold also for root key 1 if we do not give the intruder any private root keys and even if we allow also the revocation and update of root key 2 without timely distance.

The results however indicate that the situation is tricky—when we cannot be sure which of two root keys has been leaked, or when we perform several updates within a short period of time.

References

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

[2] S. Mödersheim. Abstraction by Set-Membership—Verifying Security Protocols and Web Services with Databases. In *17th ACM Conference on Computer and Communications Security (CCS 2010)*, 2010. To Appear. Available with implementation and examples on `www.imm.dtu.dk/~samo`.

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

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