Verifying SeVeCom Using Set-based Abstraction

Sebastian Mödersheim
DTU Informatics
Lyngby, Denmark
Email: samo@imm.dtu.dk

Paolo Modesti
Università Ca' Foscari Venezia
Venice, Italy
Email: modesti@dsi.unive.it

***Abstract*—We formally analyze the Secure Vehicle Communication system developed by the EU-project SeVeCom, using the AIF framework which is based on a novel set-abstraction technique. The model involves the hardware security modules (HSMs) of a number of cars, a certification authority, and the protocols executed between them. Each participant stores a database of keys that can be added or deleted depending on the different operations. The AIF-framework allows us to model and automatically analyze such databases without bounding the number of steps that the system can make and, in contrast to previous approaches in protocol abstraction, can handle databases that do not monotonically grow (and thus allow for revocation of keys). We report on two new attacks found and verify that under some reasonable assumptions, the system is secure in a black-box cryptography model.**

***Index Terms*—Vehicular communications, security protocol verification, abstract interpretation**

I. Introduction

The EU-project SeVeCom [1], [2] proposes a modern system for secure vehicle communication that shall satisfy two seemingly conflicting goals, namely on the one hand authentication and accountability for vehicle communication and protecting the privacy on the other hand. To that end, each car contains a tamper-proof *hardware security module* (HSM) that holds all private keys of the car and that performs all encryption, decryption, and verification operations. For ordinary communication, this includes a number of short-term key-pairs that are registered with, and certified by, a trusted certification authority (CA). While the CA is thus able to link all communication back to a particular car (e.g. in case of police investigation), the other participants cannot see this relation, but only link actions that are performed using the same *pseudonym* (i.e. short-term key).

There is a growing number of automated tools for protocol verification that represent the cryptographic messages by abstract terms (and thereby ignore cryptographic attacks). Many such tools like Scyther [3] are restricted to "simple" protocols that consist only of a message exchange and therefore cannot analyze a system like SeVeCom that requires the participants to maintain databases of keys. Other tools like AVISPA [4] allow for modelling databases but require restricting the number of steps that honest agents can execute, and do not scale well with this number. There are several abstraction-based approaches, avoid this problem and allow for verification with an unbounded number of steps. These techniques however have a kind of *monotonicity* built-in: what is true at some point cannot become false later, forbidding to model revocation for instance and are thus not suitable for analyzing SeVeCom.

The AIF framework [5] is an extension of the abstraction approach that allows for modeling databases (or sets) of messages that do not necessarily monotonically grow (allowing for revocation) and that inherits the benefits from the classical abstraction approaches, namely verification without bounding the number of steps that honest and dishonest participants can make.

We show that AIF is indeed well-suited for modelling and verifying the SeVeCom system. We consider several models of the system for different intruder models: one model considers the revocation-update protocols of the CA's root keys in the presence of an intruder with direct access to the HSMs. The second, comprehensive model of all the protocols needs to consider an intruder who is either outside a particular car or at least does not have access to the signing function of the HSM (which would lead to trivial attacks). We present two novel attacks that were found by our analysis in the root key update protocol, and discuss some reasonable assumptions to prevent them. We verify the security properties of the system under these assumptions. Beyond the verification of the concrete system SeVeCom, this work demonstrates how the relevant aspects of time can be modeled in verification approaches that actually *abstract from time*.

II. AIF

The AIF framework consists of the AIF specification language and a translator from AIF to first-order Horn clauses that incorporates the set-abstraction technique; it is connected to the SPASS theorem prover [6] and the protocol verifier ProVerif [7] to check the generated Horn clauses. These automated tools do not always terminate, but when they do, this gives us either an attack or a proof of security. We give a brief introduction to the AIF language; a formal definition is found in [5].

a) State Transition Systems: An AIF specification describes a state-transition system. A *state* is a set of *facts* that are true in that state, for instance the state $\{ik(crypt(k,m)), ik(inv(k)), ik(sign(inv(k'),m))\}$ may intuitively say that in this state, the intruder knows a public-key encrypted message $crypt(k, m)$, the corresponding private key $inv(k)$ and a signature $sign(inv(k'), m)$. None of the symbols here has a predefined meaning. Their meaning is rather defined through the *transition rules* that we define on states. For instance we may define rules that reflect the ability of the intruder to encrypt and decrypt messages with known keys:

$$\begin{aligned}&ik(K).ik(M) \Rightarrow ik(crypt(K, M));\\&ik(crypt(K, M)).ik(inv(K)) \Rightarrow ik(M);\\&ik(inv(K)).ik(M) \Rightarrow ik(sign(inv(K), M));\\&ik(sign(inv(K), M)).ik(K) \Rightarrow ik(h(M));\end{aligned}$$

Observe that in specifying such rules, we use *variables*, denoted by identifiers that start with an uppercase letter, in contrast to constants which start with a lowercase letter. Such a rule can be applied to any state that contains facts that match the left-hand side; this yields a new state that is obtained from the old one by adding the facts of the right-hand side under the given match.

We can specify transitions in which new values are freshly created, e.g., we can specify that at any time the intruder can generate himself a new key pair as follows:

$$=\!\![K]\!\!\Rightarrow ik(K).ik(inv(K)); \qquad (1)$$

Taking this transition, the variable K is bound to a new value (that did not occur so far), representing in this case a public key. Since the left-hand side is empty, the rule can be taken without any precondition.

b) Sets: The rules as presented up to here represent what is standard in abstraction approaches, and in particular observe the states can only monotonically grow during such transitions. This does not allow for example the modeling of a transition where a key is revoked, because the fact that the key is valid would need to be somehow "retracted". Exactly for such cases, the AIF has a way to express transitions in which the state does not monotonically grow, namely using *sets*. An AIF specification can contain an arbitrary but fixed number of sets.

For instance, our SeVeCom model will include several sets of public-keys that the HSMs of the cars maintain, e.g. $db(hsm1, ltsig, uptodate)$ denotes the set of all up-to-date long-term signing keys stored in machine $hsm1$. We can conveniently describe an enumeration of such sets using variables that range over enumeration types, for instance in this case we have a family of 28 sets $db(HSM, KeyType, Updating)$ where

$$\begin{aligned}HSM :&\quad \{hsm1, hsm2\};\\KeyType :&\quad \{root1, root2, ltsig, ltdec, stsig, stdec, ppsig\};\\Updating :&\quad \{updating, uptodate\};\end{aligned}$$

c) Transitions Using Sets: A *set-membership fact* is a fact of the form $m \in S$ where m is an element and S is a set. As an example, there are two public keys of the certification authority, called *root keys*, stored into every HSM at manufacturing time (more on their role in the system later). We can model this initialization of an HSM by a transition rule that simply creates new root keys:

$$\begin{aligned}&\lambda HSM.\ =\!\![K_1, K_2]\!\!\Rightarrow ik(K_1).ik(K_2).\\&K_1 \in db(HSM, root1, uptodate).\\&K_2 \in db(HSM, root2, uptodate);\end{aligned} \qquad (2)$$

Here, $\lambda HSM.$ says that this rule holds for any value in the domain of variable HSM ($\{hsm1, hsm2\}$ here) to avoid long enumerations. This rule is an over-approximation of reality, because it can be applied at any time and any number of times, while in the real system, there can only be one pair of root keys installed at manufacturing time. Formulating it in this way is necessary in the abstraction approaches as we explain below.

We can now model that an HSM, when receiving a correctly formed revocation message, marks the respective key as being "under update" in its database:

$$\begin{aligned}&\lambda HSM, Root.ik(sign(inv(K), [K, T])).\\&K \in db(HSM, Root, uptodate)\\&\Rightarrow K \in db(HSM, Root, updating);\end{aligned} \qquad (3)$$

Here, $Root$ ranges over $\{root1, root2\}$. Moreover, $sign(inv(K), [K, T])$ is the format of a revocation command for a root key: it needs to be signed by the private key $inv(K)$ that belongs to the root key K. For simplicity, we will often say *private root key* for the private key that belongs to a public root key. The message also includes a timestamp T that we discuss later.

We model here an HSM that is directly under the control of an intruder who can send arbitrary commands to it. This is expressed by the *ik*-fact on the left-hand side of the rule: the HSM accepts any command of the revoke-key format that the intruder can craft (as long as the respective key K is indeed in $db(HSM, Root, uptodate)$).

Similarly, for other operations where there is an answer by the HSM, we will have this answer contained in an *ik* fact on the right-hand side of the rule to model that the intruder directly obtains this answer, can analyze it, and use it for further actions like crafting another command.

Also observe that the set-membership facts on the left-hand and right-hand side differ by their update-status. While for all other facts, the state monotonically grows over transitions, set-membership facts behave differently: left-hand side facts that do not appear on the right-hand side get *removed* by the transition. Thus, the matched key K in this example is moved from the *uptodate* to the *updating* set (of the respective machine and key kind). In fact, the other transitions ensure that only signatures with up-to-date keys are considered as valid. On the left-hand side of rules, we may also specify that a rule is only applicable to states in which a certain set-membership fact does *not* hold. For instance if we declare a family of sets $used(HSM)$, we can model a simple replay-prevention:

$$\begin{aligned}&\lambda HSM, Root.ik(sign(inv(K), [K, T])).\\&T \notin used(HSM).K \in db(HSM, Root, uptodate)\\&\Rightarrow T \in used(HSM).K \in db(HSM, Root, updating);\end{aligned} \qquad (4)$$

This transition can only be taken for a timestamp T that the HSM has never seen before and that is afterwards stored as used. Here we do not model any properties of time (like freshness); we come back to timestamps later.

d) Goals and Reachability: AIF has only one built-in fact symbol: attack. We use rules that have this fact on their right-hand side to specify *attack states.* For instance we can specify that it is an attack if the intruder finds out the private key of a valid root key:

$$\lambda HSM, Root, Updating. K \in db(HSM, Root, Updating).$$
$$ik(inv(K)) \Rightarrow \mathsf{attack};$$

The *initial state* is the empty set of facts. We say that an AIF specification is *secure*, if we can reach no attack state from the initial state by using the transition rules.

e) Abstraction: Ideally, when writing a model, one does not need to think about the techniques that are used to analyze it, but unfortunately the complexity of the problems, and the side conditions that several techniques have, usually require a certain level of technical knowledge. The AIF framework is based on abstraction techniques and the AIF language is designed so that all requirements for the *soundness* of the abstraction are satisfied by construction. Soundness here means: if the abstraction of an AIF specification is secure, then so is the concrete AIF specification. The other direction does not hold in general, because the abstraction over-approximates the behavior of the concrete system and can thereby introduce attacks that have no counter-part in the concrete model. We call such attacks *false positives.*

The main point of set-based abstraction is that we consider the equivalence classes induced by the set-membership of values; in our example, the abstract model identifies all public-keys that belong to the same subset of all the databases. For this to be sound, it is crucial that the specification cannot distinguish several values that have the same membership status, so we cannot write conditions like $X \neq Y$, not even indirectly. That implies that we cannot control the cardinality of sets (that they contain a particular number of elements). More generally, we can say that what is true for one value is also true for any number of values in some reachable state.

There are two main consequences for our model of $SeVeCom$. First, we need to allow that all sets of keys can hold any number of keys; this is of course sound in the sense that it over-approximates the real behavior where number is fixed. It turns out that this over-approximation does not introduce any false attacks. Second, we cannot directly talk about time (and timestamps) because the abstract model eliminates every notion of transitions and the timely order of events. We must therefore use sets to model crucial properties of time when they are needed. For instance in the example rule (4), we have only modeled the aspect of replay checking, but not recentness; we do the latter not before § IV where it becomes unavoidable.

III. Root Key Update

We first consider a model that focuses on the root keys and their update (and ignores all other kinds of keys and protocols). We use from the previous section the initialization rule (2) and the revocation rule (3) (without replay check). The next corresponding operation that we model is the update operation which is triggered by a command of the form $sign(inv(K), [K', T])$ where K is a root key in uptodate status, and a new root key K' that is to replace the other root key (which must be in updating status). The rationale behind the format of the update and revoke command is that, if one of the root private keys is compromised, it can only be used to revoke itself, but cannot be used to update either key (which requires the knowledge of both root private keys). The intention is that the system should be secure as long as at most one of the two keys is compromised. The update for key *root1* is formalized as follows (*root2* analogously):

$$\lambda HSM. ik(sign(inv(K_2), [K, T])).$$
$$K_2 \in db(HSM, root2, uptodate).$$
$$K_1 \in db(HSM, root1, updating)$$
$$\Rightarrow K_2 \in db(HSM, root2, uptodate).$$
$$K \in db(HSM, root1, uptodate). K_1 \in revoked(HSM);$$

Here, we again ignore the timestamp T at first. We also use a new family of sets here: $revoked(HSM)$. They contain all the public root keys that have ever been discarded from an HSM and are used later to formulate the goals.

A. Modeling the Authority

The revocation and update messages should be (at least in normal protocol runs) be generated by the certificate authority (CA), the (supposed) owner of the root keys. This will happen whenever a root key is suspected to be compromised, or maybe even on a regular basis. In a first model, we consider a CA that can at any point revoke either root key. Let us first consider a model where the authority can generate a pair of revoke and update commands at any time non-deterministically. Because we cannot be sure a priori that the update is correctly communicated, we must model the CAs database of root keys independent of the HSMs databases. To that end we use the two sets $dbr(Root)$ (recall $Root : \{root1, root2\}$). The revoke and update request is produced in one transition; we give the one for revoking and updating *root1*:

$$K_1 \in dbr(root1). K_2 \in dbr(root2).$$
$$=[K]\Rightarrow K \in dbr(root1). K_2 \in dbr(root2).$$
$$ik(sign(inv(K_1), [K_1, T])). ik(sign(inv(K_2), [K, T]));$$

B. Goals

We consider three goals:

Secrecy: The intruder never knows the private key of a valid (or under update) public root key:

$$\lambda HSM, Root, Updating.$$
$$K \in db(HSM, Root, Updating). ik(inv(K));$$
$$\Rightarrow \mathsf{attack};$$

Authentication: The intruder shall not be able to produce a confusion among the parties about who generated which keys and for which purpose. For the root key update, the only potential confusion is that the intruder manages to make the HSM accept an intruder-generated key as a root key. This would mean a violation of the secrecy goal already (because the intruder knows the private key of a self-generated public key pair). For the comprehensive model in § IV, however, there are more interesting authentication properties.

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 using these keys could be accepted again by anybody. To that end, we use the set $\mathit{revoked}(\mathit{HSM})$ that we introduced before to hold all revoked keys:

$$\lambda \mathit{HSM}, \mathit{Root}, \mathit{Updating}. K \in \mathit{revoked}(\mathit{HSM}).$$
$$K \in \mathit{db}(\mathit{HSM}, \mathit{Root}, \mathit{Updating}) \Rightarrow \mathsf{attack};$$

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

C. An Attack

We first get a violation of the freshness goal and one that even works if the timestamps are checked for recentness. The attack uses the fact that the CA can generate revoke-update commands arbitrarily for the existing keys and the intruder does not even need to know any private root keys for the attack to work. Suppose we initially have two root keys installed on the HSM, called k_0 and k_1. Suppose further, the authority revoke-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 $\mathit{sign}(\mathit{inv}(k_0), [k_n, T])$ (for some timestamp T). At any such update, the intruder can block the current update message and replay an old message in order to install an older key k_i.

This attack is limited (but not prevented) by the use of timestamps: this works only if several updates are performed within a too short time, i.e. with overlapping validity periods of the timestamps. Since it is reasonable to assume that root-key updates are performed rather infrequently, this attack is not of such a practical relevance, but it suggests actually several additional security measures that are at least not explicitly mentioned in [1].

First, revoke-updates should be performed only with non-overlapping validity periods of the timestamps. Second, the HSMs should store all timestamped messages for the time they are valid and compare every further incoming message with them to prevent replay. Third, it may be a good idea to include into the update message also the public key that is supposed to be revoked and updated. Each of these suggestions can prevent the attack and each seems reasonable and good practice. An easy way to model the replay-prevention in the HSM is shown by rule (4), requiring a novel timestamp in every message. We can do the same for the update command of the HSM, but need to ensure that the revoke and update command are generated with different timestamps. This rule does not really model recentness of timestamps (so the intruder may arbitrarily delay the delivery of a message to the HSM in this model); we consider another time model in § IV.

D. Revoking the wrong key

We now check what happens if the intruder is given one of the two root keys. The following rule gives the intruder all the private root keys stored as *root1* on the HSM.

$$\lambda \mathit{HSM}. K \in \mathit{db}(\mathit{HSM}, \mathit{root1}, \mathit{uptodate}).$$
$$\Rightarrow K \in \mathit{db}(\mathit{HSM}, \mathit{root1}, \mathit{uptodate}).\mathit{ik}(\mathit{inv}(K));$$

Consequently, we restrict the secrecy goal to private keys of *root2*. But also this has an attack namely if the CA happens to revoke the wrong key, i.e., if the CA wrongly thinks that *root2* is compromised, and issues a revoke-command for a *root2* key, i.e. $\mathit{sign}(\mathit{inv}(k_2), [k_2, T])$ (for some *root2*-key k_2 and timestamp T). Since we have given the intruder a *root1*-key k_1, he can now issue the update command for k_2, namely $\mathit{sign}(\mathit{inv}(k_1), [k_3, T])$ for some intruder-generated key k_3. It is indeed unclear, even assuming that the intruder can only know one of the root keys, how the CA can be sure which one it is.

While the protocol suggests a complete symmetry between the keys, one may think of using the keys in distinct ways. Suppose the *root1* is used for the daily business, while the private *root2* is kept reserved for emergencies, maybe under additional physical protection. Then it makes sense to assume that the intruder can only find out *root1* and if there is any suspicion of compromise (or also at regular time intervals), we use *root2* to update it, and we never update *root2* itself. Restricting our model to only updates of *root1* keys, we have verified the secrecy goal.

IV. Comprehensive Model

We now extend our previous model considering also the long-term and the short-term keys and the related update protocols. We first show how to integrate the notion of time into our model and discuss the modelling of the intruder.

A. A Timed Model

Time plays an important role in the SeVeCom protocols, namely for the validity of key certificates and timestamps to prevent replay. Each HSM (and the CA) has its own clock. These clocks may differ by a certain margin δ; as long as δ is much smaller than the validity period of long-term keys, this may in the worst case disrupt communication for time δ at the end of a key's validity period, but not endanger any security properties. We therefore simply assume synchronized clocks.

The abstractions of the AIF framework do not provide any notion of time and so we cannot directly talk about the order in which events occur. Despite this fact, we can model some properties of time using the sets. The idea is to divide the timeline into several epochs. For SeVeCom, we find a split into three epochs suitable: *old*, *expsoon* and *new*. We want that the abstraction of keys depends (besides the databases that we already have) also on the epoch that they belong to. We thus define a family of sets $\mathit{timer}(\mathit{Time})$ where Time ranges over $\{\mathit{old}, \mathit{expsoon}, \mathit{new}\}$.

These epochs are used to model the life-cycle of the key as follows. A key is first freshly created by an HSM and is in epoch *new*; the key cannot be used yet, the HSM first needs to run a key registration protocol with the CA. After registration, the key becomes valid and moves from status *new* to status *expsoon*. This means that the HSM can now use the key for encryption or signing and a process to generate and register a new key can be started. After some time, the key finally moves from *expsoon* to *old* and can no longer be used; the HSM will delete the old key (but an intruder can still try to use/re-introduce it).

Let us look at these steps of the key life cycle in more detail. When the HSM has a key K that is currently in use, i.e., in epoch *expsoon*, it can generate a new key NK which is initially in epoch *new*. This rule has the form:

$$\begin{array}{l} K \in \mathit{timer}(\mathit{expsoon}).K \in \mathit{db}(\cdot)\ldots \\ =\![NK]\!\Rightarrow K \in \mathit{timer}(\mathit{expsoon}).NK \in \mathit{timer}(\mathit{new}). \\ K \in \mathit{db}(\cdot).\mathit{ik}(\cdot); \end{array}$$

where $\mathit{ik}(\cdot)$ abbreviates an outgoing certificate request message for the new key NK (to the CA) and $\ldots$ represents some other facts.

The second step models the actual progress of time:

$$K \in \mathit{timer}(\mathit{expsoon}) \Rightarrow K \in \mathit{timer}(\mathit{old});$$

i.e. a key K can change its status from *expsoon* to *old*—and this can happen "at any time" so to speak: the "world" that we model here can just choose to do such a transition for any *expsoon* key. Observe that this progress of time is independent of what the parties are doing at this time.

The third step is now expressing that, if an HSM has key K currently in use and a fresh key NK has successfully been registered with the CA, then as soon as K has turned *old* (with the previous rule), we can discard K and start using NK as the current key:

$$\begin{array}{l} K \in \mathit{timer}(\mathit{old}).NK \in \mathit{timer}(\mathit{new}).K \in \mathit{db}(\cdot).NK \in \mathit{db}(\cdot) \\ \Rightarrow K \in \mathit{timer}(\mathit{old}).NK \in \mathit{timer}(\mathit{expsoon}).NK \in \mathit{db}(\cdot). \\ K \in \mathit{revoked}(\mathit{HSM})\ldots; \end{array}$$

Thus NK's transition from *new* to *expsoon* happens exactly when the HSM starts using it.

B. Modelling the intruder and the API

The API of the HSM offers the interface through which applications running on the car can invoke the functionality provided by the HSM. In particular, besides keys and device management, decryption and digital signing are the two main functions offered by the API. They can be employed with the long-term and short-term keys. Recall that the corresponding private keys are stored in the HSM memory and never released outside. As an example, for long-term decryption keys (that have the attribute *ltdec*), we model the decrypt function as follows in AIF:

$$\begin{array}{l} \lambda \mathit{HSM}, \mathit{Updating}.\mathit{ik}(\mathit{crypt}(K, M)). \\ K \in \mathit{db}(\mathit{HSM}, \mathit{ltdec}, \mathit{Updating}) \\ \Rightarrow \mathit{ik}(M).K \in \mathit{db}(\mathit{HSM}, \mathit{ltdec}, \mathit{Updating}); \end{array}$$

Note that *Updating* is a variable in order to model the ability to employ the keys regardless their status. Similarly we have a rule for signing:

$$\begin{array}{l} \lambda \mathit{HSM}, \mathit{Updating}.\mathit{ik}(M).K \in \mathit{db}(\mathit{HSM}, \mathit{ltsig}, \mathit{Updating}) \\ \Rightarrow \mathit{ik}(\mathit{sign}(\mathit{inv}(K), M)).K \in \mathit{db}(\mathit{HSM}, \mathit{ltsig}, \mathit{Updating}); \end{array}$$

Observe that this allows the intruder to get a signature with any valid signing key in the HSM on any message he can construct (and similarly he can decrypt any message that is encrypted with an HSM-stored decryption key). To put it another way: if the intruder has direct access to an HSM, then there is not much difference from the intruder knowing the respective private keys himself—only he cannot get self-generated keys *into* the HSM. With this, of course, the intruder can trivially break several goals of the key update protocols; the only exception is the root-key update which is secure (under the given assumptions) even for an intruder with direct access to HSMs.

It is reasonable to assume that the intruder has only direct access to the HSM(s) in his own car(s). For other cars, he can only observe the communication with their environment. In the following we thus consider an attack model where the intruder has limited access to the HSMs; experimenting with different settings we can verify all our goals if the intruder cannot access the signature function for long-term keys, while we may still give him access to all other functions without breaking the security.

C. Long-Term Key Update Protocol

The long-term signature generation key of the HSM is used to authenticate messages with the real identity of the HSM. The long-term decryption key of the HSM is used to decrypt encrypted messages that are intended for the vehicle. These keys are generated by the HSM typically at manufacturing time, and the CA creates the certificates for such keys ($\mathit{LongTerm} : \{\mathit{ltsig}, \mathit{ltdec}\}$):

$$\begin{array}{l} \lambda \mathit{HSM}, \mathit{Root}, \mathit{LongTerm}.RK \in \mathit{dbr}(\mathit{Root}) \\ =\![K]\!\Rightarrow K \in \mathit{db}(\mathit{HSM}, \mathit{LongTerm}, \mathit{uptodate}). \\ K \in \mathit{dbca}(\mathit{HSM}, \mathit{LongTerm}).RK \in \mathit{dbr}(\mathit{Root}).\mathit{ik}(K). \\ \mathit{ik}(\mathit{cert}[\mathit{HSM}, K, RK]).K \in \mathit{timer}(\mathit{expsoon}); \end{array}$$

Here, $\mathit{cert}[\mathit{HSM}, K, RK]$ denotes a certificate by the CA that K is a key of HSM (signed with the private root key $\mathit{inv}(RK)$). $K \in \mathit{timer}(\mathit{expsoon})$ ensures that K is directly usable (as the certification is already finished) and at any time we can start the update for a new key.

As explained before, the key update happens in several phases. First, the HSM generates new long-term key pairs and produces a certificate request message for the CA and waits for receiving a corresponding certificate. The second phase begins when the current keys expires, and only now the HSM starts to use the newly generated key.

The goals for the long-term keys are similar to those for the root keys, only here we have a relevant authentication goal. We formalize that whenever the CA has recorded the key K as a valid long-term key (for signing or decryption) of a particular HSM, then this machine also has K stored in the database (either in status *uptodate* or *updating*). It would thus count as an attack, if the intruder manages to confuse the CA about the long-term keys of an HSM. This is formulated as follows in AIF:

$$\begin{aligned}&\lambda HSM, LongTerm.K \in dbca(HSM, LongTerm).\\ &K \notin db(HSM, LongTerm, updating).\\ &K \notin db(HSM, LongTerm, uptodate) \Rightarrow \mathsf{attack};\end{aligned}$$

This goal is particularly important for accountability, because it ensures that the CA does never attribute keys to a wrong HSM. We verified that our model is safe under the assumption that the intruder does not have full access to the long-term signature function.

D. Short-Term Key Update Protocol

The short-term keys are used to sign or decrypt the periodic beacon messages broadcasted or received by the vehicle. For privacy reasons, the public keys that correspond to these short-term private keys may be certified in an anonymous manner by a trusted third party, called the pseudonym provider (PP), which is a particular instance of CAs within the PKI. An anonymous certificate contains only the public key, the validity period of the certificate, the identifier of the issuer, and the digital signature of the issuer. In particular, it does not contain the identifier of the vehicle to which it has been issued.

However the HSM does not store the certificates but only supports the pseudonym management by generating short-term key pairs, storing the private keys, and computing signatures. The HSM can be instructed (through its API) to generate a new short-term signature key pair. When the HSM generates a new key pair, it creates a new entry in the internal key database and it stores the private key together with the corresponding context information and outputs the public key. It is the responsibility of applications running on the car to obtain a certificate for it. The certificate request is passed back to the HSM for being signed with the long-term signature key of the HSM.

Also for short-term keys we check the above mentioned goals of secrecy, authentication and freshness. The SeVeCom specification assumes an authentic and confidential communication channel between the HSM and the PP. We made experiments both with realizing the channels with the long-term keys of the HSM, and by using an ideal channel model that abstracts from the implementation [8]. In both cases, we can verify that the system is safe, even allowing the intruder to have access to the short-term signature function.

V. Conclusions

Our analysis of the SeVeCom root key update protocol has revealed two potential weaknesses. Under reasonable assumptions though, we can exclude the attacks and verify the root key update. The detection of the attacks and the verification of the fixed system take a few minutes each. We have also considered a comprehensive model of the system including all communication protocols and have verified all goals under the assumption that the intruder does not have access to the signing functionality of all HSMs. The verification of this more complex task takes less than 2 hours. The specifications are available at [5].

Our work was inspired by a similar work of Steel [9], who modeled parts of the SeVeCom-system using SATMC. Here, the number of steps had to be bounded. Since he did not model the certification authority, he could not find the problems in root key update.

Our work shows that even complex systems (that require features like the revocation of keys) can be efficiently verified without bounding the number of steps that agents can perform. More generally, it shows that even the relevant aspects of time can be integrated into a model that abstracts from the traces and transitions (and thereby any notion of time): using sets for different time periods, we can integrate the time information into the abstraction.

Acknowledgments

Paolo Modesti was partially supported by MIUR Project *IPODS: Interacting Processes in Open-ended Distributed Systems* and by a scholarship for foreign nationals issued by the Danish Government under the Cultural Agreements.

References

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

[2] P. Papadimitratos, L. Buttyan, T. Holczer, E. Schoch, J. Freudiger, M. Raya, Z. Ma, F. Kargl, A. Kung, and J. Hubaux, "Secure vehicular communication systems: design and architecture," *Communications Magazine, IEEE*, vol. 46, no. 11, pp. 100–109, 2008.

[3] C. Cremers, "The Scyther Tool: Verification, falsification, and analysis of security protocols," in *Computer Aided Verification.* Springer, 2008, pp. 414–418.

[4] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. Drielsma, P. Héam, O. Kouchnarenko, J. Mantovani *et al.*, "The AVISPA tool for the automated validation of internet security protocols and applications," in *Computer Aided Verification.* Springer, 2005, pp. 281–285.

[5] S. Mödersheim, "Abstraction by Set-Membership—Verifying Security Protocols and Web Services with Databases," in *CCS 2010.* ACM, 2010, implementation and examples, including SeVeCom specifications, available on www.imm.dtu.dk/˜samo.

[6] C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic, "System description: Spass version 3.0," in *CADE*, 2007, pp. 514–520.

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

[8] S. Mödersheim and L. Viganò, "Secure pseudonymous channels," in *ESORICS*, 2009, pp. 337–354.

[9] G. Steel, "Towards a formal security analysis of the Sevecom API." in *ESCAR*, 2009.