POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Formal verification of a V2X Security Credential Management System

Supervisors
Prof. Riccardo Sisto
Prof. Fulvio Valenza
Dott. Simone Bussa

Candidate
Francesco Rametta

Academic Year 2022-2023

Ai miei genitori

† Ai miei nonni

Acknowledgements

Grazie ai miei relatori, i Prof. Riccardo Sisto e Fulvio Valenza, che mi hanno permesso di approfondire il mondo della verifica formale e quello delle V2X con questo lavoro di tesi.

Un grazie speciale va al Dottorando Simone Bussa che è stato sempre presente e disponibile supportandomi durante questo percorso e gli auguro il meglio per la sua carriera accademica.

Grazie

Contents

List of Figures 7

1 Introduction 9
1.1 Structure of the document 10

2 General Overview 11
2.1 V2X - Vehicle to everything 11
2.2 Formal Verification 13
2.2.1 Symbolic Modelling 15
2.3 Tamarin-Prover 16
2.4 Features 17
2.4.1 Input File - Theory 17

3 Security Credential Management System 23
3.1 SCMS Design 24
3.2 SCMS Structure 25
3.3 Certificate Provisioning Model and Certificate Types 29
3.3.1 Certificate Types 30
3.4 Butterfly Key Expansion 31
3.5 SCMS - Protocol Steps 33
3.5.1 Bootstrapping 33
3.5.2 Certificate Provisioning 34
3.5.3 Misbehavior reporting 37
3.5.4 Revocation 38

4 Objective of the thesis 41

5 SCMS - Design Model in Tamarin 43
5.1 Preliminary steps and Assumptions 43
5.2 Tamarin theory setup 44
5.2.1 Support rules 47
5.3 Bootstrapping 49
5.4 Certificate Provisioning 50
5.5 Revocation 56

6 Results 63
6.1 Sanity checks 63
6.1.1 Executability 63
6.1.2 Authentication 65
6.1.3 Bootstrapping success 67
6.1.4 Partial deconstruction 68
6.2 Secrecy 69
6.3 Privacy Properties 70
6.3.1 Anonymity 70
6.3.2 Unlinkability 72

7 Conclusion 77

Bibliography 79

List of Figures

2.1 Possible scheme PKI. 12
2.2 Formal verification procedure steps. 14

3.1 SCMS architecture. 26
3.2 SCMS Legend. 27
3.3 Butterfly key expansion steps. 33
3.4 Enrollment process. 35
3.5 Certificate provisioning process. 36
3.6 Pseudonym certificate revocation process. 39

5.1 Simplified model. 44
5.2 Certificate provisioning scheme. 51
5.3 Revocation scheme. 57

6.1 Authentication sanity check success. 66
6.2 Authentication - attacker with enrollment certificate - 1. 66
6.3 Authentication - attacker with enrollment certificate - 2. 67
6.4 Authentication - RA to PCA. 68
6.5 Authentication - RA to PCA message craft - 1. 69
6.6 Authentication - RA to PCA message craft - 2. 70
6.7 Lemma secrecy. 71
6.8 Observational equivalence: rule-equivalence. 76

Chapter 1

Introduction

Vehicles are currently being developed and sold with increasing levels of connectivity and automation. Communication technologies connect the vehicles with various on-road elements, such as pedestrians, infrastructures, roads, cloud computing platforms etc. V2X communication technology is expected to improve traffic efficiency by reducing traffic incidents and road pollution. Some examples of how this technology could be very promising and useful are: traffic jam/incident reporting, collision warning and avoidance, cooperative automated driving, infotainment services. All of this, results in a heightened risk of cyber-security attacks. In literature many different types of attack have been extensively and deeply analyzed and discussed. Here the focus is on the privacy. The main issue is related to the mechanism used to provide authenication and integrity of the exchanged messages. This is typically obtained using digital signatures and a digital certificate that certifies the public key of the vehicle. But this exposes the vehicle to a lack of anonymity and unlinkability. An attacker who intercepts all messages signed using the same certificate could link these data and trace the vehicle position. The most promising solution to solve this problem is the use of pseudonymous certificates. By using pseudonyms, each node can safely exchange messages preserving authentication and integrity and at the same time they can provide a certain level of conditional linkability, that can be used by authorities in case of disputes to resolve the initial identity of the vehicle or to profile services offered to the nodes.

Among all pseudonym related vehicular communication protocols, the state-of-the-art is represented by *Security Credential Management System*, developed by the Crash Avoidance Metrics Partners LLC.

The thesis focuses on the formal analysis of the protocol by means of the TAMARIN prover. Formal verification strives to provide a rigid and thorough method of analyzing the correctness of a security protocol, allowing even minor flaws to be discovered. Since the aims of using this communication protocol will be, to be widespread on V2X communications, even the slightest vulnerability will result in tremendous harm to all users who use it. This is why it is important to formally verify these protocols, so to be sure they meet the security properties they claim. Lemmas were implemented that describe all the security properties that must be fulfilled by the protocol.

1.1 Structure of the document

This thesis work has been divided into the following parts to explain in the most complete way all the various phases:

- **Chapter 2**: it gives a general overview and the context of this thesis. In particular outlines general information about V2X and an introduction about what is Formal Verification and the tool used to analyze this protocol, which is Tamarin-prover.
- **Chapter 3**: it provides a deep analysis of the protocol to be formally verified and all details related to why it was chosen in comparison to other protocols.
- **Chapter 4**: it describes the objective of this work thesis
- **Chapter 5**: provides a thorough analysis of the design choices and the model obtained to be able to analyze it with the Tamarin-prover.
- **Chapter 6**: it presents the results obtained by this formal verification.
- **Chapter 7**: it presents the conclusion and possible future work that could be added and improve the current thesis.

Chapter 2

General Overview

2.1 V2X - Vehicle to everything

Vehicle to everything is a new generation of communication technology that connects vehicles to everything else. "V" represents the vehicles and "X" represents everything else that interacts with it (other vehicles, people, road infrastructure and the network).

In V2V connections, vehicles communicate with each other by sending and receiving data such as speed, relative position, and brakes. They can also capture photos, audio, and video of the surrounding environment. In this way, it is possible to predict the behavior of other vehicles and improve road safety, as well as semi-automatic and automatic driving in some cases.

V2I refers to all communications between vehicles and road infrastructure, such as traffic lights and traffic cameras while V2P refers to communications between vehicles and vulnerable groups, such as pedestrians and cyclists, through devices such as smartphones and wearable devices.

These vehicles and infrastructure typically exchange broadcast messages with their neighbors, and they are not encrypted. Since they are sensitive data that can compromise user safety and privacy, mechanisms are needed to protect them, especially authentication to ensure that the message comes from a valid vehicle and integrity to ensure that the message is not modified by an attacker once it is sent.

These infrastructures gather information from the surrounding area and send real-time information that include dangerous intersections (blind spots), accidents, construction sites, and emergency vehicles passing through. Therefore, this information, alerts the user to potentially dangerous situations and optimizations that the driver can take.

In this way, they can become nodes in V2X communications and send and receive information and alerts, for example communicating with a traffic light, if a pedestrian or cyclist takes longer than the green signal to cross the road it can warn adjacent vehicles.

As we mentioned earlier, every message exchanged must be protected to ensure integrity and authentication. Almost all standard network protocols use digital signatures and asymmetric certificates to achieve this. The problem with doing it is that the anonymity of vehicles is not guaranteed because every certificate contains the vehicle's ID and thus its identity, allowing malicious users to intercept all messages sent by the same vehicle associated with the same certificate and therefore, in the worst case, compromise the vehicle's location.

In this thesis, we focus on privacy properties. To solve this problem, the use of pseudonymous certificates has been introduced. It is a certificate that allows the node to authenticate itself anonymously, by using a pseudonym, which is derived from the real ID. Various schemes have been proposed in the literature to manage them in a distributed manner. They are issued by entities in a hierarchy, a Public Key Infrastructure (PKI), and have a limited lifespan, meaning they must be changed regularly.

If this did not happen and the vehicles had a single pseudonym certificate, they would be traceable by means of their pseudonym. The traceability problem would shift from traceability of the real identity of the vehicle to traceability of the pseudonym. If the attacker were able to discover the real identity of the vehicle with the unique pseudonym (e.g. the vehicle and the attacker are alone in a empty street), the linkability property would be compromised.

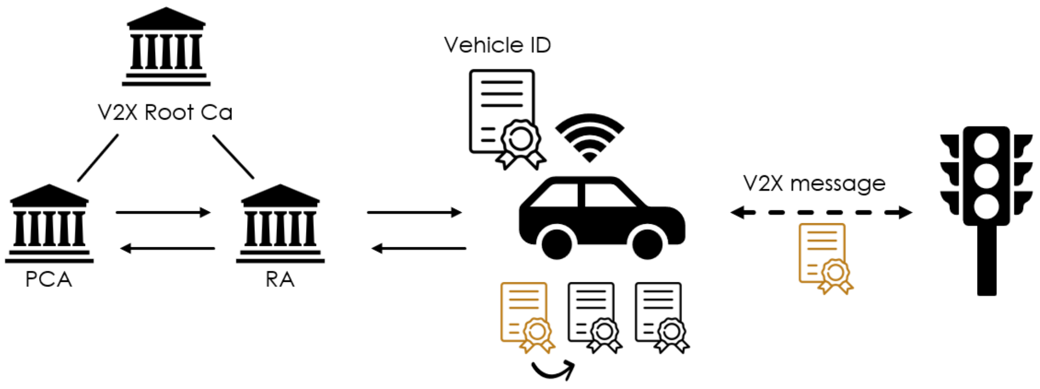


Figure 2.1. Possible scheme PKI.

As we said, there are different types of schemes (asymmetric encryption, identity-based cryptography, group signature schemes, symmetric cryptography schemes), but the one we will analyze uses asymmetric encryption (e.g. 2.1), which in the literature has been shown that this type of scheme is more advantageous than the others.

This system architecture is a top contender to assist the creation of a national Public Key Infrastructure for V2X security and is currently moving from research to proof-of-concept. For reliable communications among participating vehicles and infrastructure equipment, which is required for safety and mobility applications based on V2X communications, it issues digital certificates to them. A mechanism is needed to create and distribute these pseudonymous certificates to the various vehicles. The various pseudonymous certificates follow a cycle consisting of four phases.

1. Bootstrapping
2. Certificate provisioning
3. Misbehavior reporting
4. Revocation

Boostrapping is the phase in which the vehicle registers with the Certification Authority (CA), where the vehicle presents itself to a CA to obtain an enrollment certificate containing the vehicle's ID. This certificate is not used in communications, but rather to request pseudonymous certificates from the CA, which provides the device with all the required information to communicate with the SCMS protocol and other devices.

Certificate provisioning is the most complex part, where the pseudonymous certificates that the vehicle must use, are delivered. The vehicle contacts the CA with the enrollment certificate obtained in the boostrapping phase (which is used for authentication) and receives a series of pseudonymous certificates in response. These certificates will be used to communicate with other nodes in the protocol.

At some point, a node may be detected as malicious. This leads us to the Misbehavior Reporting phase, which is a detection tool that analyzes messages sent by other nodes to determine which ones are correct. An algorithm for misbehavior detection is executed on the device (i.e., locally on the node) to identify nodes with anomalous behavior.

Revocation is the phase in which a malicious node is reported to the competent authority and a revocation is requested. At this point, the authorities can use the pseudonym to identify the real identity of the vehicle and revoke it.

2.2 Formal Verification

The main problem when developing new communication protocols is that even if researchers try to explore all possible combinations to avoid security failures, they can arise unexpectedly and attackers are very creative in exploiting all possibilities. Therefore, it is crucial to detect possible weaknesses and vulnerabilities at the initial stages of design to secure the system. And to do this comes to the rescue Formal Verification. It uses a variety of mathematical and logical techniques to assess the accuracy of designs. By employing such techniques, it is able to check several aspects of the system from the very beginning and offer security guarantees, including the functional correctness of implementations, programming defects, side-channel analysis and the fulfillment of security properties.

Fig. 2.2 shows the formal verification process for the protocol. The process generally consists of four fundamental steps, the first of which is carefully reading the protocol's specification. The following stage involves manually creating a model from a specified specification. The defined model is then translated into the input language of the model

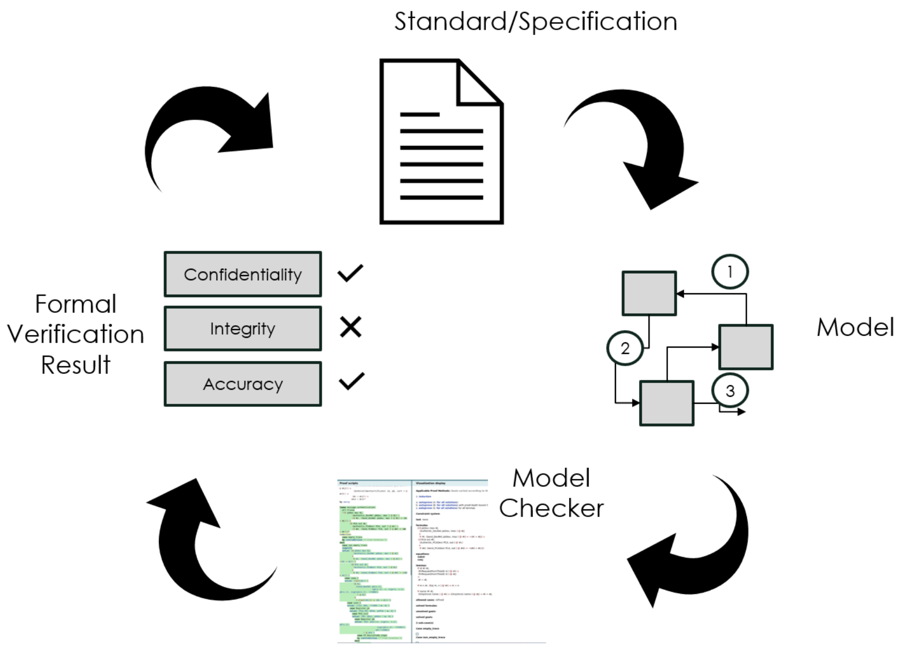


Figure 2.2. Formal verification procedure steps.

checker. The final step entails reviewing the outcomes of formal verification and, if necessary, formulating recommendations for standard amendments in light of those outcomes.

Confidentiality and Authenticity of communicating parties (Authentication) are two crucial security qualities that should be handled when it comes to communication protocols. Both of these properties are typically discussed in formal verification methods under the assumption that the adversary has complete control over the network (Dolev-Yao Model). The Dolev-Yao model is a formal model used for the verification of security protocols. The model was proposed by Danny Dolev and Andrew Yao in 1983 and has since been widely used in the formal verification of security protocols. On a public network, an attacker has complete control over the messages sent and received and has the ability to replay, edit, delete, and forward communications (even if they are part of multiple sessions). Additionally, if communications are delivered unencrypted, the attacker can read them; however, if messages are sent encrypted, it can only read them if it has access to the decryption key. Additionally, it may execute cryptographic operations and generate new messages using the information it currently has.

In this model the security properties must hold true for any state that a protocol may enter during execution based on predetermined transition rules. Additionally, it is presumptively true that messages can only be decrypted by an adversary who has access to the key according to cryptographic primitives. When a term (such as a session key) cannot become part of an adversary's knowledge in any of the protocol's reachable states, secrecy has been achieved as a security property of the communication protocol.

The following attributes can be used to describe the authentication security property of assurance of communication parties identities: aliveness, weak agreement, non-injective agreement, and (injective) agreement.

Formal verification can be divided into two types:

1. Model checkers

2. Theorem provers

Model checking involves checking the system or program against a formal model to determine whether it meets its requirements. It is often used for large, complex systems. Model checkers automatically and thoroughly check a system's model in its state space in relation to a specified specification. It involves automatically checking all possible states of a formal model of the system to ensure that the specification is satisfied. The design and specification characteristics are generally provided as algebraic constraints or theorems by the theorem provers, who frequently need human experience to direct a proof of correctness. Although model checkers are typically easier to use, more focused on a particular problem domain, and designed to verify properties in this field, the range of problems that they can handle is necessarily constrained. Some examples of tools are ProVerif, Tamarin and Scyther. Tamarin additionally provide the possibility of manual guidance and therefore act as theorem prover.

Theorem proving is used for smaller systems or critical components of larger systems. Theorem proving is a technique used to prove that a system or program satisfies a given specification using mathematical logic. Theorem proving involves formalizing the specification and the system or program in a logical language and then using deduction rules to prove that the system or program satisfies the specification. The inputs of a theorem prover are the Theory of the formal system, the Property and Human assistance. It will tell if the Property is a theorem or not. If the answer is yes it will also give a proof because it can provide a positive response only if it is able to find a proof. In case it was unable to find a proof it can say nothing because maybe the proof exists but it was unable to find it.

2.2.1 Symbolic Modelling

The symbolic model and the computational model are the two basic types of models used in formal security protocol verification. The computational model is closer to how protocols are really executed, but the proofs are more challenging to automate. The symbolic model is a more abstract model, making it simpler to construct automatic verification tools. This distinction mostly results from the fact that cryptographic primitives are viewed as perfect blackboxes in the symbolic model and are represented by function symbols in an algebra of terms that may or may not include equations. The adversary can only compute using these primitives because messages are terms on them. In contrast, a message is a bit string in the computational model, and the protocol is formalized as a probabilistic polynomial-time Turing machine. The protocol's security is then explained in terms of a game in which an adversary modelled by an arbitrary probabilistic polynomial-time Turing machine interacts with the protocol. If there is a negligible chance that any adversary will win the game, the protocol under analysis is secure. The symbolic model abstracts away numerous features in comparison to the computational model. On the one hand, this suggests that the computational model may detect protocol defects that the symbolic model could overlook. Most importantly, assessments using symbolic models omit the possibility of attacks on the protocol that take use of flaws in the particular cryptosystems employed in the protocol. On the other hand, even if the symbolic model is much simpler than the computational one, many practically important attacks on a protocol, such as man-in-the-middle (MITM) assaults, can still be detected using this type of model.

2.3 Tamarin-Prover

Tamarin-prover is an automated verification tool that was developed for the symbolic modeling and analysis of security protocols. It is based on multiset rewriting rules and property specification in a guarded fragmento of first-order logic allowing quantification over messages and timepoints. It is an open-source software tool that was developed by David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse and Benedikt Schmidt at the ETH Zurich.

It can manage equational theories like Diffie-Hellman, bilinear pairings, and user-specified subterm-convergent theories as well as complex security models for key exchange protocols. In Tamarin, there are two ways to build a proof: fully automated mode and interactive mode. The first one uses heuristics along with deduction and equational reasoning to direct the proof search. The tool might not be able to end the verification process, though, because the majority of the setting's attributes are undecidable. In the event of termination, the tool returns either a counterexample or a demonstration of correctness.

Tamarin-prover is a tool that can analyze security protocols and automatically detect potential security flaws. It uses a symbolic model-checking approach to analyze the security properties of a protocol. The symbolic model-checking approach involves representing the protocol as a set of equations, which are then solved using automated reasoning techniques. The approach is designed to detect potential security flaws by analyzing the properties of the equations that represent the protocol.

For example, it can handle complex cryptographic operations, such as hash functions and digital signatures. It can also model multiple sessions of a protocol and analyze the security properties of the protocol across all sessions.

One of the main advantages of Tamarin-prover is that it can be supplied with detailed models of the protocol being analyzed. This means that the model accurately capture all the details of the protocol, including any non-trivial interactions between participants. On the other hand failure to do so could result in false positives or false negatives in the analysis.

A limitation of Tamarin-prover is that it is computationally expensive. The symbolic model-checking approach used by Tamarin-prover involves solving a large set of equations, which can be computationally expensive for large protocols. This can limit the size of protocols that can be effectively analyzed by Tamarin-prover.

In the following, will be presented the main components the Tool requires to asses the security properties.

2.4 Features

As previously said, Tamarin include verification and falsification of security properties in relation to the protocol's model. In the following will be presented a high-level description of Tamarin's features. More precisely the focus here is solely related to the aspects relevant for our model. See the manual for a thorough explanation of Tamarin.

2.4.1 Input File - Theory

The tool expects a .spthy file as input that contains some elements relevant to model and verify the protocol in exam. As we said, here are listed only the elements used in this particular model, following a detailed description of each one:

Functions

Tamarin supports both additional user-defined function symbols in addition to a fixed set of built-in function symbols. Only pairing and projection function symbols are available in every Tamarin file. `fst` and `snd` are two function symbols that model the projections of respectively the first and second parameter, while the binary function symbol `pair` models the pair of two messages. The following equations adequately describe the properties of projection:

```
fst(pair(x,y)) = x
snd(pair(x,y)) = y
```

The function defined for this particular protocol will be further discussed in more detail in Chapter 5.

Equations

Equational theories are used to represent the properties of functions, such as how symmetric decryption always works as the opposite of symmetric encryption when both utilize the same key. The syntax to include equations in the context is as follows:

```
equations: lhs1 = rhs1, ..., lhsn = rhsn
```

No public constants are allowed, and any variables on the right hand side must also appear on the left hand side. Both the left and right hand sides can contain variables. A particular class of user-defined equations, notably convergent equational theories with the finite variant property, are supported by Tamarin's symbolic proof search (Comon-Lundh and Delaune 2005). Because Tamarin does not verify that the submitted equations fall under this class, entering equations outside of it may result in non-termination or inaccurate results without any warning.

Built-in message theories

Tamarin supports a fixed set of Built-ins that refers to the most common functions useful to model recurrent security features e.g. encryption, hashing, etc.

```
theory exampleName
begin
    ...
builtins: signing, symmetric-encryption, asymmetric-
   encryption, diffie-hellman, hashing
end
```

In the following will be given a short description of some of the built-ins message theories. To learn more refers to the manual.
symmetric-encryption: This theory models a symmetric encryption scheme defining the function sybols secn/2 and sdec/2 which belongs to the equation sdec(senc(m,k),k) = m. Specifically, secn is used to encrypt the message `m` with the key `k`, while sdec decrypts the message with the same key `k`.
asymmetric-encryption: this theory models asymmetric encryption of messages with a public key encryption scheme. To do so function symbols are defined, binary for aenc/2, adec/2 and unary for pk/1. This symbols are related by the equation adec(aenc(m, pk(sk)), sk) = m. In this case, aenc encrypt the message `m` with the public key `pk(sk)`, adec decrypt the message with the related secret key `sk` and pk return the public key associated to the secret key.
signing: this theory models a signature scheme. It defines an equation verify(sign(m,sk),m,pk(sk)) = true for binary function sign, ternary function verify, unary function pk and the constant function true. The function sign, sign the message `m` with the secret key `sk` and verify check the signature correctness by comparing the signature with the message supplied and the corresponding public key and this must be equal to true.
xor: this theory models the exclusive-or operation. It adds the function symbols XOR/2 and zero/0. XOR satisfies the cancellation equations and is associative and commutative.

```
x ⊕ y = y ⊕ x
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)
x ⊕ zero = x
x ⊕ x = zero
```

Restrictions

The strength of restrictions allows for the creation of models that would not otherwise be conceivable. A restriction is essentially a property that must be true for each and every trace. Tamarin simply assumes the property without proving it. The following is a description of two of the most practical and widely applicable constraints. According to the restriction Equality, x and y must be equal whenever an action fact Eq(x,y) appears in the trace. Similar to this, the inequality limitation states that the two arguments must not be equal anytime the action fact Neq(x,y) appears in the trace. These sample limitations can be found in Chapter 6 of the Tamarin manual. Keep in mind that @i can be read as "at time point i" and that #represents a temporal variable. Also @i is short for @ #i.

```
restriction Equality :
 " All x y #i. Eq(x,y) @ #i ==> x = y "

restriction Inequality :
 " All x #i. Neq(x,x) @ #i ==> F "
```

The restriction syntax consists of the term restriction followed by the restriction's name, and a colon. The constraint is expressed as a guarded first-order logical formula enclosed in double quotes, "... "

Multiset rewriting rules

They are rules that model the core of the theory provided and to specify the parallel execution of the protocol and the adversary. Multiset rewriting is a formalism that is commonly used to mimic concurrent systems since it easily supports distinct transitions. All communications are described as terms in a multiset rewriting system, which specifies a transition system. The status of the system is made up of a variety of facts. A fact represents an element in the system, e.g., a public key belonging to an agent while a sequence of action facts generated by an execution is called a trace. Multiset rewriting rules define how the system transitions to a new state. In Tamarin, a rewrite rule has a name and three parts, each of which is a list of facts: the left-hand side of the rule, the transition label, or what we refer to as a "action fact," and the right-hand side of the rule.

$$[\text{LHS}]\ -[\mathit{actionFacts}] \rightarrow [\text{RHS}]$$

A rule consumes the left-hand side facts, whereas the right-hand side facts are produced by the rule. Some facts (linear facts) vanish when ingested, whereas others (persistent facts) do not and can be consumed an infinite number of times. Like built-ins functions, there exist some built-in rule. More specifically: `[ Fr(~x)]` fact which is used to generate a fresh value (random) and `In(m)` and `Out(m)` which are used to represent the network controlled by the Dolev yao attacker. The `In(m)` fact model the reception of messages while the `Out(m)` model the sending of messages over the untrust network. There exist also the possibility to model a trusted (private) network, which will be further explained later.

Raw Sources and Partial Deconstructions

Precomputation produces what are known as the "raw sources," which simply means that the sources for each fact have already been calculated. However, there are situations when Tamarin is unable to determine the source of a fact. This can result in so-called partial deconstruction due to Tamarin's untyped system and other factors. Let's think about a straightforward example to illustrate the idea. The first rule below simply constructs a fact `Simple_Fact(~a)` holding the fresh value `~a`. The second rule then uses this information, sending the term in it over the untrusted network. Be aware that the term `mex` in the consumed fact is a fresh value, but the second rule is completely unaware of this. Assume Tamarin cannot resolve (i.e., determine the origin of) the fact `Simple_Fact (mex)`. The conveyed word `mex` in this situation might represent any number of things, such as private keys, user secrets, session identifiers, and more. Tamarin is unable to rule out the likelihood that the network attacker has just acquired all the necessary knowledge in this situation. Typically, the lemmas are not terminated as a result of this.

```
rule Create_Fresh_Value :
    [ Fr (~a) ]
    --[ ]->
    [ Simple_Fact (~a) ]

rule Send_Value :
    [ Simple_Fact (mex) ]
    --[ ]->
    [ Out(mex) ]
```

There are two ways to tackle the problem in the aforementioned scenario. The first one should only be utilized if you are certain that doing so won't change the model. By simply adding a tilde `~` to the term `mex` in the second rule, as shown below, you may inform the prover that it represents a new value. This might change your model since, if the term `mex` wasn't fresh in some traces, you simply eliminated those traces. This can lead to missing attacks.

```
rule Send_Value :
    [ Simple_Fact (~mex) ]
    --[ ]->
    [ Out (~mex) ]
```

Another option is to create a sources lemma. A sources lemma has the same appearance as a regular lemma but has the `[sources]` annotation. A sources lemma, as contrast to a standard lemma, is evaluated using the unrefined sources. The objective is to produce a genuine and verifiable sources lemma that provides Tamarin with sufficient knowledge to fully deconstruct all unresolved facts, the so called partial deconstructions. In this example, we want to make it clear to Tamarin that if the second rule is invoked with the term `n`, the first rule must have been called previously with the same term. We mark the rules in the lemma with action facts so that they can be referenced. We mark the second rule with `Problem(mex)` because it is the one that partially deconstructs the sent term. We annotate the first rule with `Source(~a)` because it is the term's source. The lemma should state that whenever a `Source(n)` precedes a `Problem(n)`, either the attacker already knew the term n, or a `Source(n)` preceded it. Tamarin learns in the first scenario that the term is a fresh value and cannot be arbitrary. In the second case, the attacker already knows the term and learn nothing when it is sent out. KU() stands for knowledge of the attacker. The full example is provided below. Keep in mind that the second approach is fully secure. You cannot unintentionally exclude traces since Tamarin would discover a counterexample to the sources lemma in such a situation.

```
rule Create_Fresh_Value :
    [ Fr (~a) ]
    --[ Source (~a) ]->
    [ Simple_Fact (~a) ]

rule Send_Value :
    [ Simple_Fact (mex) ]
    --[ Problem (mex) ]->
    [ Out(mex) ]

lemma typing [sources] :
    " All n #i.
    ( Problem (n) @i
    ==>
    ( (Ex #j. Source (n) @j) | (Ex #j. KU(n) @j & j < i)
      )) "
```

Let bindings

They are used to make the definition of theory more readable and modular. Let bindings enable the definition of local macros within the boundaries of their related rules. They are especially helpful if a term appears more than once in a rule or if we need to access the components of a term.

```
rule MyRuleName:
    let foo1 = h(bar)
        foo2 = <'bars', foo1>
        ...
        var5 = pk(~x)
    in
    [ ... ] --[ ... ]-> [ ... ]
```

In the context of a rule, these let-binding expressions can be used to specify local term macros. Each macro, which defines a substitution, should appear on a separate line: the right-hand side of the = sign may be any arbitrary term, and the left-hand side of the sign must be a variable. After replacing all variables that appear in the let by their right-hand sides, the rule will be interpreted. The aforementioned example shows how macros can make use of the left-hand sides of previously defined macros.

Lemmas

They describe the properties that we want the model satisfies. Lemmas are extremely similar to restrictions, with the important distinction being that, unlike restrictions, lemmas must either be proven or disproven. Exists-trace lemmas and all-traces lemmas are the two basic categories of lemmas. Existential lemmas include those with exists-trace. If and only if there is at least one trace that satisfies the specified property, it is true. On the other hand, all-traces lemmas are true if and only if the property is true for all traces. Lemmas are expressed as logical formulas. Below are some examples that were modified from the Tamarin manual. Keep in mind that the all-traces annotation is typically left out. The first lemma technically states that there is a trail between an agent A sending a message and an agent B receiving it, who may or may not be the same agent A. According to the second lemma, B must have actually transmitted that message first whenever someone asserts to have gotten an authentic message m from B.

```
lemma executable :
 exists-trace
    " Ex A B m #i #j. Send (A,m)@i & Recv (B,m) @j "

lemma message_authentication :
 all-traces
    " All B m #i. Authentic (B,m) @i
        ==> (Ex #j. Send (B,m ) @j & j<i) "
```

Oracle

It is frequently more convenient to write a "oracle" than to manually pick each proof goal in interactive mode. An oracle is a piece of code that executes apart from the Tamarin-prover. It takes a numbered list of proof goals as input, and produces an ordered list of numbers indicating which proof goals should be resolved first.

An oracle can directly change the original sequence of facts in any way you choose. An oracle can be executed by Tamarin using `tamarin-prover interactive file_theory .spty --heuristic=O --oraclename=oraclefile.py` . Tamarin does not always terminate, nevertheless, because of the nature of the particular problem.

Chapter 3

Security Credential Management System

The primary design objective is to offer security and privacy to the greatest extent that is acceptable and practical. Vehicles are issued pseudonym certificates, and the development and provisioning of such certificates are distributed among several entities in order to achieve a decent amount of privacy in this environment. One of the biggest issues is to enable effective revocation of misbehaving or defective vehicles while protecting privacy from insider assaults given the high number of pseudonym certificates per vehicle.

In the field of Vehicle-to-vehicle (V2V) communications, has been proved that communications between nearby vehicles in the form of continuous broadcast of Basic Safety Messages (BSMs) has the potential to reduce unimpaired vehicle crashes by 80% through active safety applications. BSMs are digitally signed and contain the sender's time, position, speed, path history, and other pertinent data. The receiver assesses each message, confirms the signature, and then determines whether or not the driver needs to see a warning. The success of safety applications based on BSMs is directly impacted by their correctness and reliability, which are of utmost importance. Each BSM is digitally signed by the sending cars to thwart attackers from introducing bogus messages, and the receiving vehicles confirm the signature before acting on it. Building confidence among participants and ensuring the system operates properly both require a Public-Key Infrastructure (PKI) that facilitates and manages digital certificates. The PKI with special characteristics is implemented via the Security Credential Management System (SCMS) suggested in this work. This SCMS differs from a conventional PKI in a number of ways. The size (i.e., the number of devices it supports) and the harmony of security, privacy, and efficiency are the two factors that matter most. It will be able to produce 300 billion certificates for 300 million automobiles annually when operating at full capacity. The SCMS design also provides efficient methods for requesting certificates and handling revocation and moreover it is essential for the SCMS to also be able to cover V2I applications as well as service announcement and provisioning like Internet access. In this paper are introduced novel concepts:

- distributed provisioning of certificates for privacy protection against insider attacks

- butterfly keys for communication-efficient request of an arbitrarily large number of certificates by a device
- linkage values for efficient revocation of seemingly unrelated pseudonym certificates of a device
- elector ballots for management of root certificate authority and electors

3.1 SCMS Design

For V2V safety applications, a risk analysis was conducted; however, it was not published. According to the risk assessment, both insiders and outsiders who assault SCMS's security could pose a threat to users' privacy. Thus, it was determined that the SCMS must prevent or lessen the following forms of attacks:

- Attacks on end-users' privacy from SCMS insiders
- Attacks on end-users' privacy from outside the SCMS
- Authenticated bogus messages leading to false warnings

The first two points on the list are addressed via a process called "Privacy by Design. "Misbehavior detection" addresses the third point. We make the assumption that the cryptographic systems in use today are sufficiently secure for the SCMS design. This presumption would be false if sufficiently powerful quantum computers were introduced, as they would be able to defeat the Elliptic Curve Digital Signature Algorithm (ECDSA) for all feasible curve sizes. Nevertheless, this design is flexible and modular, so even if this were to occur, it would still permit an upgrade to post-quantum cryptography methods.

Talking about "Privacy by Design" security system's main objective is to safeguard end users' privacy, particularly that of individuals driving private vehicles. Since the majority of privately owned vehicles have a single registered owner, the system is created to make it challenging to follow that vehicle based on its data transfers. This is because the capacity to track the vehicle may be used to associate vehicle operation with the registered owner. There are two ways to do this: 1) Future apps that send unicast or multicast messages (as opposed to broadcast messages) should utilize encryption and other techniques to prevent the communication's parties' identities from being revealed. 2) Applications that involve sending broadcast messages from end-user vehicles use the privacy-preserving features of the SCMS to make it challenging for eavesdroppers in two physically distinct locations to determine whether BSMs transmitted at the two locations originated from the same vehicle. Inside attackers and outside attackers are the two categories of attackers that are considered. BSMs are accessible to outside attackers, but they are unable to obtain any other data, including certificates that have not yet been broadcast. BSMs and other data, such as data produced during the certificate issuance process, are accessible to an inside attacker. We suggest that end-entity devices be issued with a large number of certificates and that they frequently alter the certificates accompanied by BSMs in order to maintain privacy against external attackers. The SCMS operations are split among its components, and those components are required to have organizational separation between them in order to provide defense from inside attackers. The SCMS is made to require cooperation from at least two of its parts in order to gather useful data for device tracking. We define "unlinkability" as the idea that it becomes more difficult to identify that two broadcasts from the same device actually originated from that device if they are more distant in time and space. The requirement states that if a vehicle's broadcast messages contain data that is specific to the vehicle and may be connected to a location, the data should change often to make it very difficult for an eavesdropper to track that vehicle.

Instead, talking about "Misbehavior Detection & Revocation", this is addressed by periodically distributing Certificate Revocation Lists (CRLs). The CRL is used by devices to recognize and reject messages from revoked devices. In order to reject any further requests for certificates from revoked devices, the SCMS also keeps internal blacklists of those devices. Each device receives a number of certificates for V2V safety applications, so standard CRLs would not be suitable in our case because they would grow too large. This is make efficient by introducing a novel concept of *linkage values*.

3.2 SCMS Structure

In the following are explained all the components of the SCMS protocol. Figure 3.1 shows them and their connections. V2V and V2I functionality is provided separately by components labelled V/I, whereas general V2X functionality is provided by components marked 'X'. The three pairs of RSEs and OBEs are used to highlith different use cases. The first pair displays the connections necessary for bootstrapping, the second pair demonstrates the connections necessary for certificate provisioning and misbehavior reporting, and the third pair demonstrates the connections necessary for retrieving the CRL from the CRL Store.

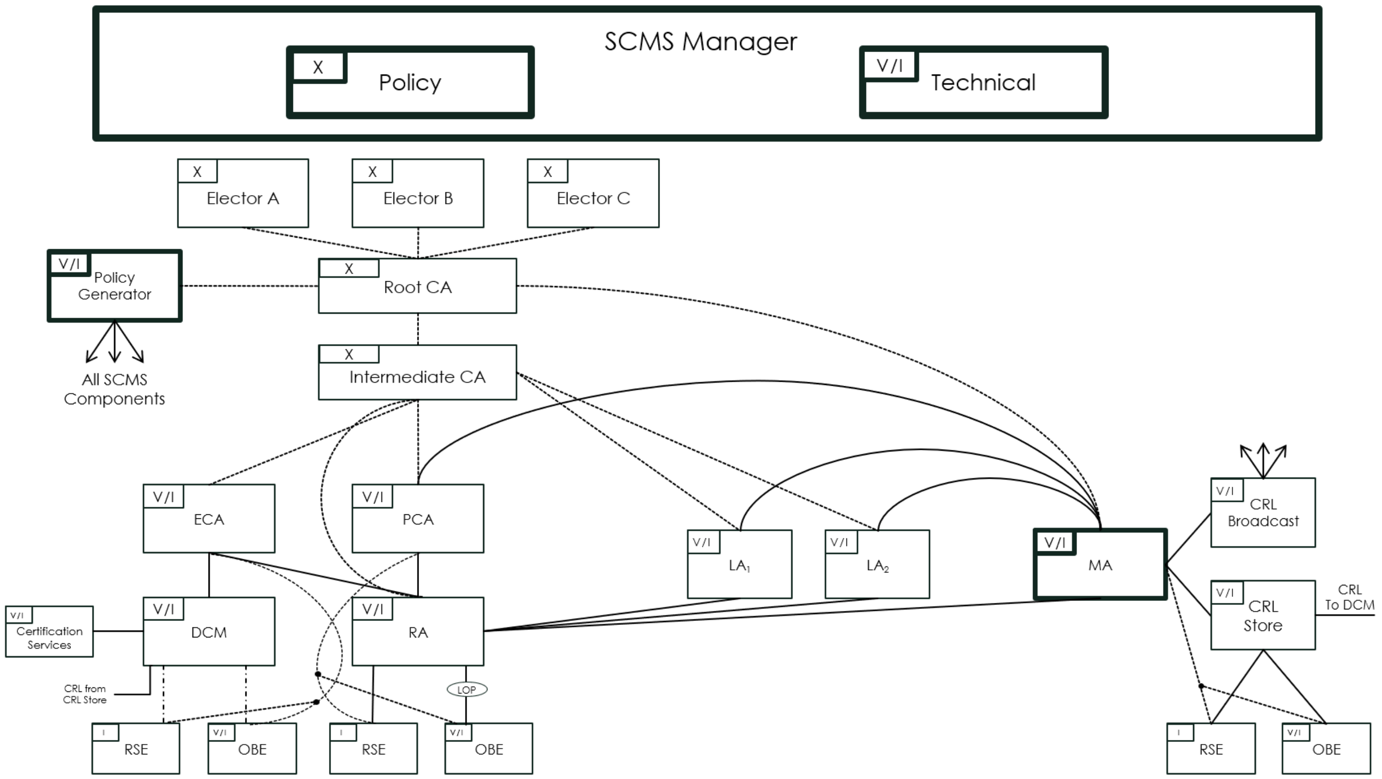


Figure 3.1. SCMS architecture.

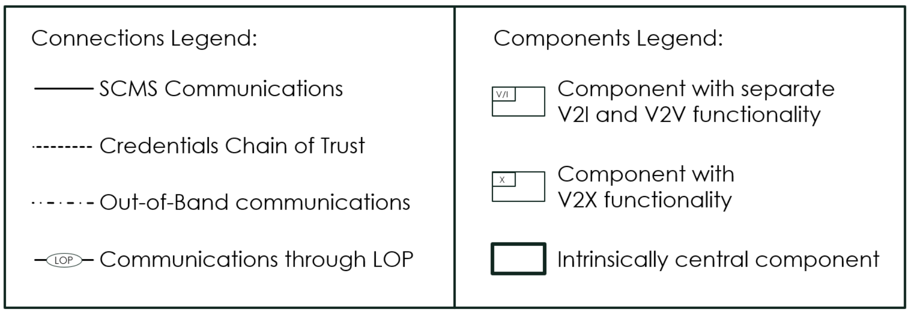

Connections Legend:
SCMS Communications
Credentials Chain of Trust
Out-of-Band communications
LOP
Communications through LOP
Components Legend:
V/I
Component with separate V2I and V2V functionality
X
Component with V2X functionality
Intrinsically central component

Figure 3.2. SCMS Legend.

In the SCMS, there are four different kinds of connections:

- Solid lines represent regular, secure communications, including certificate bundles
- Dashed lines represent the credential chain of trust. The ECA certificate is used to verify enrollment certificates, the PCA certificate to verify pseudonym, application, and identity certificates, and the CRL Generator certificate, which is a component of the MA certificate, to verify certificate revocation lists. This line does not imply that information is transferred between the components it connects.
- Dash-Dotted lines represent Out-of-Band communications, e.g., the line between the RSE and the DCM
- The Location Obscurer Proxy (LOP) is used for connections designated with the LOP symbol. The Location Obscurer Proxy is an anonymizer proxy that removes all requests' geographic information.

All online components connect with one another through a secure and dependable communication channel using protocols like the Transport Layer Security (TLS). If data is transmitted through an SCMS component that is not intended to read that data, it is encrypted and authenticated at the application layer. Here a brief description of the components presents in figure 3.1:

- SCMS Manager: Ensures that the SCMS operates effectively and fairly, establishes organizational and technological regulations, and establishes standards for evaluating misconduct and revocation petitions to make sure they are legitimate and consistent with processes.
- Certification Services: Describes the certification procedure and offers details on the categories of devices that are approved to accept digital certificates.
- CRL Store: Simple pass-through component that stores and delivers CRLs.
- CRL Broadcast: A pass-through component that broadcasts the most recent CRL via RSEs or satellite radio systems.

- Device: An end-entity (EE) device that sends or receives messages
- Device Configuration Manager (DCM): Provides all necessary configuration options and certificates during bootstrapping, and certifies to the ECA that a device is qualified to receive enrollment certificates.
- Electors: represent the center of trust of the SCMS. They certify ballots that support or disapprove an RCA or another elector. To build trust between RCAs and voters, the SCMS Manager distributes those ballots to all SCMS components, including devices. Each elector possesses a self-signed certificate, and the initial group of electors will be implicitly trusted by all system entities. Therefore, after they have implemented the basic set, all entities must safeguard electors against illegal modification.
- Enrollment CA (ECA): A device can use an enrollment certificate as a passport to authenticate against the RA when obtaining certificates, for example. Enrollment certificates may be issued by various ECAs for various manufacturers, locations, or device categories.
- Intermediate CA (ICA): In order to protect the root CA from traffic and attacks, the intermediate CA (ICA) acts as a secondary certificate authority. The Intermediate CA certificate is issued by the Root CA.
- Linkage Authority (LA): creates pre-linkage values, which are then combined with linkage values to create certificates that can be revoked effectively. The SCMS has two LAs, designated as LA1 and LA2. The splitting makes it impossible for a LA operator to link certificates associated with a specific device.
- Location Obscurer Proxy (LOP): by modifying the source addresses, hides the location of the requesting device and prevents network addresses from being associated with specific locations.
- Misbehavior Authority (MA): Processes misbehavior reports to find probable misbehavior or device malfunctions, and if necessary, revokes or adds the reported devices to the CRL. Additionally, it starts the process of putting the matching enrollment certificates to the RA's internal blacklist and connecting a certificate identifier to them. The MA consists of two subcomponents: CRL Generator (CRLG), which generates, digitally signs, and makes the CRL available to the public; and Global Misbehavior Detection, which identifies which devices are misbehaving.
- Policy Generator (PG): The Global Policy File (GPF), which contains information on global configuration, and the Global Certificate Chain File (GCCF), which contains all trust chains for the SCMS, are both maintained and updated by the Policy Generator (PG), which also signs updates.
- Short-term pseudonym, identification, and application certificates are issued to devices by the pseudonym CA (PCA). Individual PCAs can, for instance, be restricted to a specific geographic area, manufacturer, or device type.

- Pseudonym CA (PCA): provides devices with temporary pseudonym, identification, and application certifications. Individual PCAs can, for instance, be restricted to a specific geographic area, manufacturer, or device type.

- Registration Authority (RA): evaluates and verifies device requests. It then generates unique requests for certificates to the PCA based on those. The RA puts in place safeguards to make sure that devices with certificates that have been revoked are not given new ones and that they are not given more than one set of certificates at a time. Additionally, the RA relays policy decisions made by the SCMS Manager as well as authenticated information regarding changes to devices' SCMS configuration, such as a component altering its network address or certificate. Additionally, the RA shuffles the requests/reports while sending pseudonym certificate signing requests to the PCA or passing information to the MA to prevent the PCA from interpreting the sequence of requests as a clue as to which certificates may be in the same batch.

- Root Certificate Authority (RCA): In the SCMS, an RCA is the root certificate authority at the top of a certificate chain, serving as a trust anchor in the sense of a conventional PKI. It issues certificates for SCMS subsystems like PG and MA as well as ICAs. An RCA has a self-signed certificate, and trust in an RCA is established by a ballot with a majority vote of the electors. RCA certificates must be kept in a trust store, which is a secure location for storing digital certificates. Any certificate can be verified by an entity by checking every certificate in the chain leading from the one being used to the trusted RCA. The cornerstone of any PKI is a notion known as chain-validation of certificates. The system may be compromised if the RCA and its private key are not secure. An RCA is normally off-line when not in use because of how important it is.

3.3 Certificate Provisioning Model and Certificate Types

The focus in this section is to explain the provisioning of pseudonym certificates to OBEs because the provisioning of other types of certificates represent subsets of this case. A provisioning model for pseudonym certificates has been created that strikes a balance between multiple conflicting requirements:

- Privacy vs. Size vs. Connectivity: For privacy concerns, certificates should only be used for brief periods of time. Due to the devices' limited memory capacity and high price in a moving context, they cannot keep a lot of certificates. On the other hand, the majority of cars are unable to regularly connect to the SCMS and download fresh certificates as needed.

- CRL Size and Retrospective Unlinkability: Devices that behave improperly or malfunction should be able to have their certificates revoked by the SCMS, but doing so would make the CRL quite big. It has been created a way to efficiently revoke a large number of certificates without disclosing the certificates that the device used before to misbehaving.

- Certificate Waste vs. Sybil Attack: For privacy considerations, certificates must be frequently changed. One choice is to issue numerous certificates, each of which is valid for a brief window of time following the other. There would be a lot of unused certificates as a result of this. Another choice is to issue certificates that are all valid at once and for longer periods of time. This would make it possible to compromise one device and pretend to be numerous devices (the so-called Sybil attack).

To address all this requirements the best trade-off is given adopting the CAR 2 CAR Communication Consortium (C2C-CC) model where multiple certificates are valid at once, the certificate validity period is days rather than minutes, and the certificate usage pattern can vary from device to device. For instance, a device might use one certificate for 5 minutes after startup, switch to another, use that for a longer time period before changing certificates again, or even use that certificate until the end of the trip. In particular here are proposed the following parameter values:

- Certificate validity time period: 1 week
- Number of certificates valid simultaneously (batch size): minimum 20
- Overall covered time-span: 1 – 3 years

Due to a substantially higher certificate use than the Safety Pilot model, this model offers a respectable level of anonymity against tracking while maintaining modest storage requirements. The certificates of a device that doesn't use all of its available certificates in a particular week (for example, a device that only uses 13 of the available 20 certificates) cannot be linked. Additionally, if a device reuses a certificate, it can only be linked after one week. Moreover, the methodology enables a simple procedure for topping-off pseudonym certificates. Devices don't have to specifically ask for new certificates; instead, the SCMS will continuously issue new certificates throughout the device's lifetime, up until the point at which the device stops accepting certificates for a significant amount of time. The certificates will be made available by the RA in one-week batches (for example, as a zip file), which devices will download over a TCP/IP connection. The batches will be put into files and given names based on device information and time. The device will have complete control over what and how much they download, when they download it, and how often. The RA will advise users when to expect new certificate batches (for example, once per month).

3.3.1 Certificate Types

Various application types may have various requirements for certificate management within the overall Connected Vehicle system. Therefore, establishing how many various certificate management process flows need to be supported should be a part of a complete SCMS specification. Five end-entity certificate types were found to satisfy all use cases:

- OBE enrollment certificates: enrollment certificates are given out as part of the bootstrap process and are used subsequently to request message signing and/or encryption certificates.

- RSE enrollment certificates: For RSEs, these documents serve as a substitute for OBE enrollment certificates.

- OBE pseudonym certificates: Pseudonym certificates offer pseudonymity, unlinkability, and quick and effective certificate revocation. In order to defend the RA against an insider attack, security mechanisms including shuffling, linkage values, butterfly key expansion, and certificate encryption via PCA to the OBE are used. OBEs utilize this kind of certificate to certify broadcasts of basic safety messages (BSMs). Additionally, when unlinkability is necessary for permission, this kind of certificate is employed. To improve a receiver's capacity to verify each BSM they get while using BSM broadcast, pseudonym certificates may be added to each signed BSM, or they may be attached to a select number signed BSMs only.

- OBE identification certificates: are employed for situations where a device has to identify itself. Pseudonymity and unlinkability are not features offered by this type of certificate. The SCMS applies privacy-preserving methods to the OBE during the production of OBE pseudonym certificates, such as shuffling and PCA encryption. When creating OBE identifying certificates, these procedures are not always employed. However, to enable continuous certificate generation, butterfly key expansion is utilized. Identification certificates are used for authorization purposes.

- RSE application certificates: RSEs use these certificates to sign service announcement and broadcast messages as well as, to give an OBE access to an encryption key so that they can communicate encrypted data. It should be noted that this is the only type of EE certificate that contains an encryption key.

3.4 Butterfly Key Expansion

This section explain the novel cryptographic construction introduced to manage certificate issuing. Here we emphasise once again the fact that this mechanism has been designed and deeply analyzed in [1] . In a typical process a device would typically create a private/public key pair before requesting certificates from a PKI. The device generates a certificate signing request (CSR) that contains the public key and sends it over a secure channel to the PKI. The certificate will subsequently be signed by the PKI's CA and given to the requester. Such a strategy has drawbacks for OBE pseudonym certificates since thousands of public keys would need to be produced in the device and submitted to the SCMS. This drawback is solved by the butterfly keys, that enables an OBE to request any number of certificates, each with a unique signing key and encryption key. This is accomplished by sending a single request that only has two expansion functions, one encryption public key seed, and one public key seed for signing. To prevent the RA from connecting the content of certificates with a specific OBE, the PCA encrypts them to the OBE. Without butterfly keys, each certificate would require the OBE to transmit a separate signing key and encryption key. In addition to reducing the amount of work required by the requester to generate the keys, butterfly keys enable requests to be made even when connectivity is not good. Below is a description of the butterfly expansion for signing keys for elliptic curve cryptography, but it may be readily modified for any discrete-logarithm-type hardness assumption. Except for a little difference in how the inputs to AES are produced, the butterfly expansion for the encryption key is the same as that for the signing key. In the following, we use lowercase letters to represent integers and uppercase letters to represent curve points.

The elliptic curve discrete logarithm problem states that It is difficult to determine the value of a given P and $A = aP$ but not a. The following is how butterfly keys make use of this. There is a predetermined base point of some order l named G. The integer a and the point $A = aG$ make up the caterpillar keypair. A value and an expansion function, $f_k(\iota)$, which is a pseudo-random permutation in the integers mod l, are supplied to RA by the certificate requester. Keep in mind that the RA just iterates the counter ι. Here is defined the expansion function for signing keys $f_k(\iota)$, used to generate points on the NIST curve NISTp256:

$$f_k(\iota) = f_k^{int}(\iota) \ \ mod\, l, \quad where \tag{3.1}$$

1. $f_k^{int}(\iota)$ is the big-endian integer representation of

$$DM_k(x+1) \mid\mid DM_k(x+2) \mid\mid DM_k(x+3), \tag{3.2}$$

2. $DM_k(m)$ is the AES encryption of m using key k in the Davies-Meyer mode. To generate the final output, here the output of the function is XORed with the input, i.e. $DM_k(m) = AES_k(m) \oplus m$,

3. $x + 1, x + 2, x + 3$ are obtained by incrementing x by 1 each time,

4. 128-bit input x for AES is derived from time period $\iota = (i, j) \ as: \ (0^{32}||i||j||0^{32})$.

With the exception of x, which is obtained as: $(1^{32}||i||j||0^{32})$, the expansion function for encryption keys is also defined as above. j is a counter within i and represents the number of certificates per i (for example, 20 certificates each week). i is a global value (for example, representing a week). The SCMS Manager sets both values.
RA can now generate up to 2^{128} *cocoon public keys* as $B_\iota = A + f_k(\iota) * G$, where the corresponding private keys will be $b_\iota = a + f_k(\iota)$. The public keys are now known to the RA but the private keys are only known to the OBE. The certificate requests provided to the PCA by the RA contain the cocoon public keys. The problem that arise is that by using these expanded public keys unaltered by the PCA, the RA could recognize those public keys in the certificates and track the OBE because knows which public keys come from a single request.

To prevent this, the PCA generates a random c_ι for each cocoon public key B_ι in order to obtain $C_\iota = c_\iota G$. The *butterfly public key* which is included in the certificate is $B_\iota + C_\iota$.

The PCA gives the RA the certificate and the value c for the private key reconstruction to be returned to the OBE. The certificate and the reconstruction value c_ι are encrypted to the OBE to prevent the RA from determining which certificate matches a certain public key in a request. The OBE will modify its private keys, $b_\iota^{'}$ with $b_\iota^{'} = b_\iota + c_\iota$. Each certificate must be encrypted with a distinct key in order to prevent the PCA from knowing which certificates belong to which OBE and thos keys are also generatd by using the butterfly key approach. The OBE gives a caterpillar encryption public key $H = hG$, then the RA expands it to cocoon public encryption keys $J_\iota = H + f_e(\iota)G$, and finally the PCA uses these keys to encrypt the response.Figure 3.3 depicts the butterfly key expansion concept for signing keys.

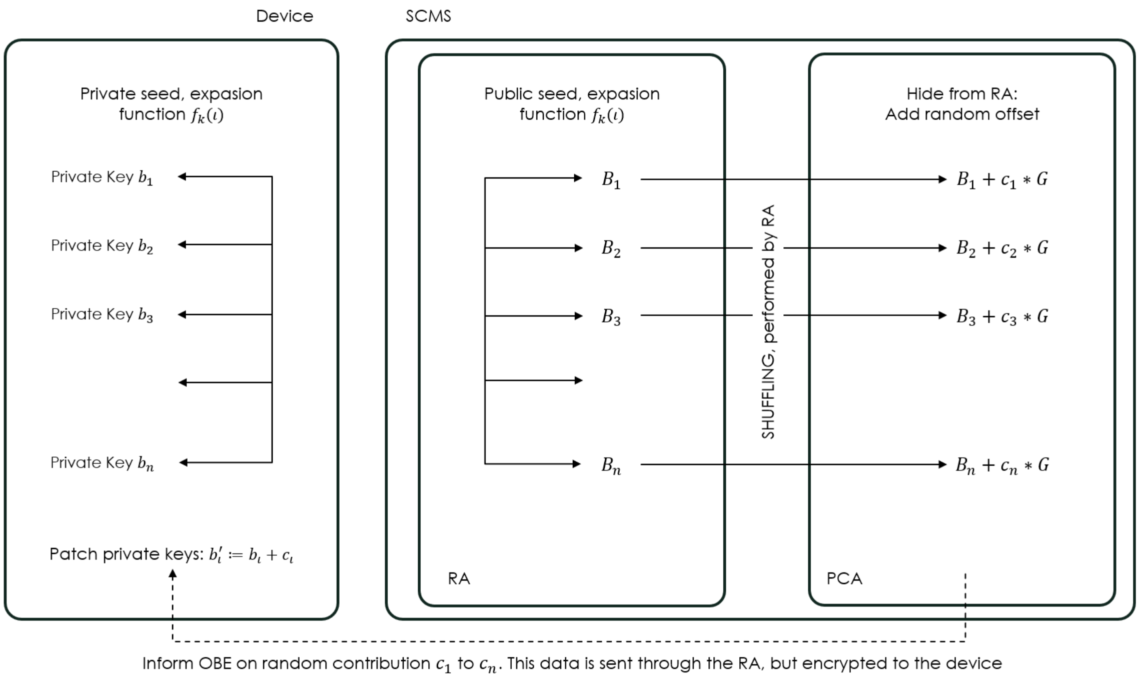

Device
SCMS
Private seed, expasion function $f_k(\iota)$
Private Key b_1
Private Key b_2
Private Key b_3
Private Key b_n
Patch private keys: $b_\iota' := b_\iota + c_\iota$
Public seed, expasion function $f_k(\iota)$
B_1
B_2
B_3
B_n
RA
SHUFFLING, performed by RA
Hide from RA: Add random offset
$B_1 + c_1 * G$
$B_2 + c_2 * G$
$B_3 + c_3 * G$
$B_n + c_n * G$
PCA
Inform OBE on random contribution c_1 to c_n. This data is sent through the RA, but encrypted to the device

Figure 3.3. Butterfly key expansion steps.

3.5 SCMS - Protocol Steps

This section presents the four phases of the protocol, specifically:

1. Bootstrapping
2. Certificate Provisioning
3. Misbehavior reporting
4. Revocation

3.5.1 Bootstrapping

In the SCMS, the life cycle of a device begins with bootstrapping. It provides the device with all the data needed for communication with the SCMS and other devices. It is necessary that the device bootstraps with accurate data and that the CAs only issue certificates to certified devices. A device, the DCM, the ECA, and the certification services component are all parts of the bootstrapping process. It is assumed that the DCM will communicate with the device being booted using an out-of-band channel in a secure environment and that it has established communication channels with other SCMS components, such as the ECA or the policy generator.

The two steps of bootstrapping are initialization and enrollment. The process of initialization is how the device acquires the certificates it requires in order to be able to trust received messages, while the enrollment is the process by which the device get the enrollment certificate it needs to sign messages sent to the SCMS. In the initialzation process information recevied are the the certificates of all electors, all root CAs and of intermediate CAs, PCAs, and moreover the certificates of misbehavior authority, policy generato and CRL generator. In the enrollment process the device acquires the data necessary to communicate with the SCMS and take part in the V2X communications system. This includes the enrollment certificate, the ECA certificate, the RA certificate, and any additional data required to connect to the RA. In addition the certification services provide the DCM with information about device models which are eligible for enrollment. The enrollment process is shown in figure 3.4.

3.5.2 Certificate Provisioning

The OBE pseudonym certificate provisioning procedure must safeguard end-user privacy while requiring the least amount of computing work possible for devices with limited resources. Since the provisioning of certificates for other certificate types is a simple subset of this one, we will concentrate only on pseudonym certificate provisioning in the sections that follow. This procedure is intended to safeguard privacy from both internal and external attackers.

The SCMS design makes sure that no single component has access to or generates the whole set of information needed to track a vehicle. The PCA encrypts the pseudonym certificates before sending them to the device, so even if the RA knows the enrollment certificate of a device that requests them and sends them to it, it is unable to read the content of those certificates. Each pseudonym certificate is independently created by the PCA, but it is unaware of the recipient of those certificates or which certificates the RA sends to the same device. The PCA embeds the masked hash-chain values that the LAs produce as "linkage values" into each certificate. The MA reveals their identities by publishing a secret linkage seed pair on the CRL, which effectively connects and revokes all upcoming pseudonym certificates of a device, . However, only one LA is not able to trace devices via linking certificates or to revoke a device; for the revocation procedure, both LAs, the PCA, and the RA must work together.

This process ensure the following privacy mechanisms in the SCMS:

- **Obscuring Physical Location** – To keep an end-entity device hidden from the RA and the MA, the LOP hides its precise position.
- **Hiding Certificates from RA** – Nobody will be able to connect the public key seeds in requests to the resultant certificates thanks to the butterfly key expansion

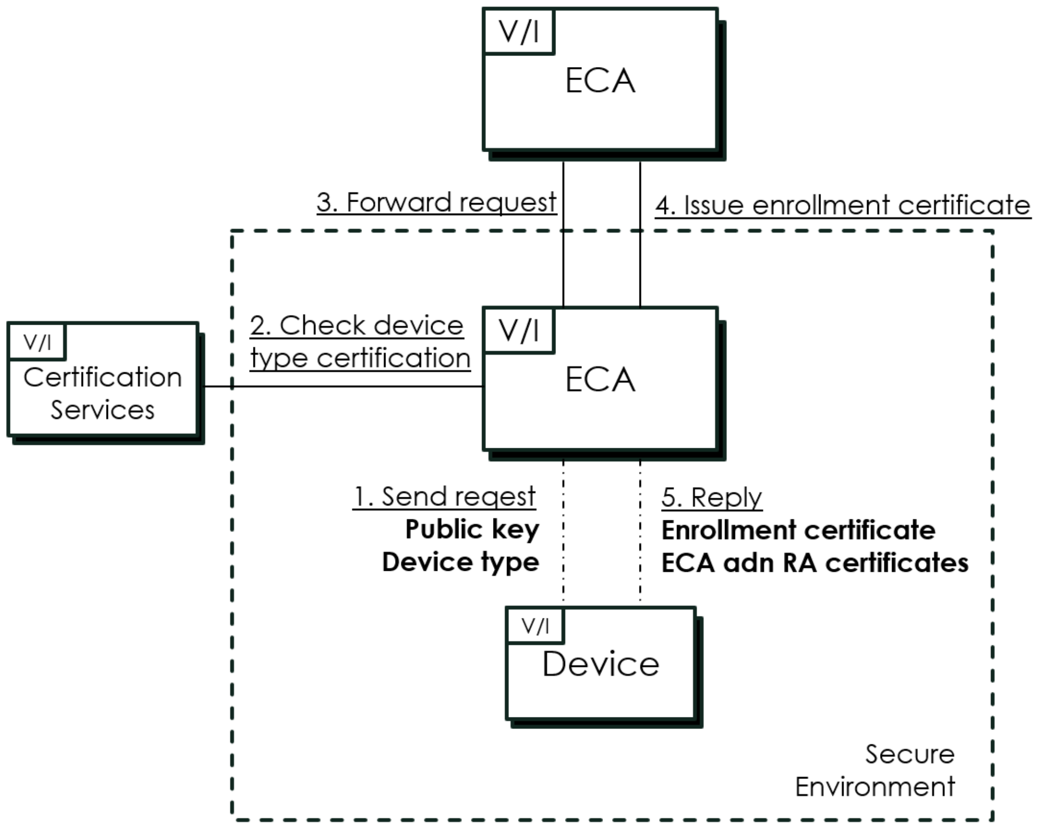


Figure 3.4. Enrollment process.

procedure. The RA cannot associate certificates with a device if the certificates are encrypted to the device.

- **Hiding Receiver and Certificate Linkage from PCA** – The RA separates incoming requests into requests for distinct certificates after expanding them using Butterfly keys. The requests are then sent to the PCA after being shuffled. This prevents the PCA from discovering whether any two certificate requests are from the same device.

By having a brief overview of the process, here follows a detailed explanation of the certificate provisioning process which is illustrated in figure 3.5 :

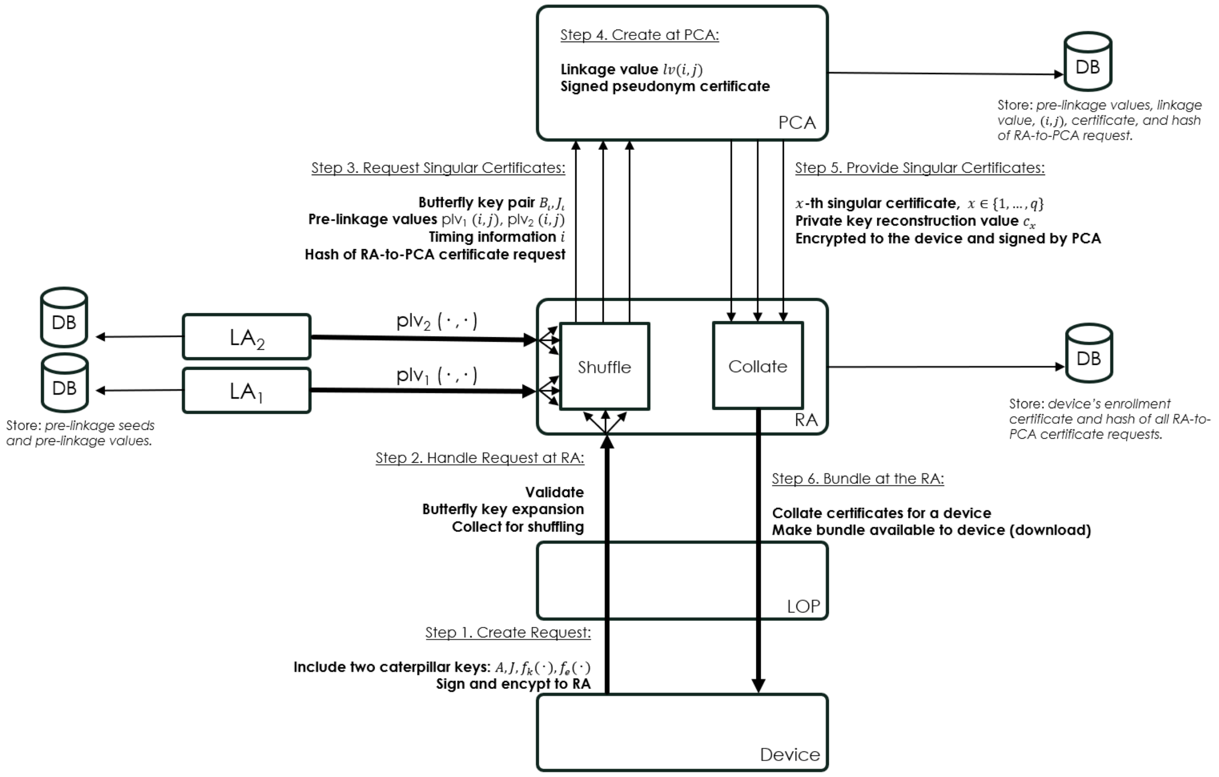

Step 4. Create at PCA:
Linkage value $lv(i,j)$
Signed pseudonym certificate
PCA
DB
Store: *pre-linkage values, linkage value, (i,j), certificate, and hash of RA-to-PCA request.*
Step 3. Request Singular Certificates:
Butterfly key pair B_ι, J_ι
Pre-linkage values plv$_1$ (i,j), plv$_2$ (i,j)
Timing information i
Hash of RA-to-PCA certificate request
Step 5. Provide Singular Certificates:
x-th singular certificate, $x \in \{1,\dots,q\}$
Private key reconstruction value c_x
Encrypted to the device and signed by PCA
DB
LA$_2$
plv$_2$ $(\cdot,\cdot)$
DB
LA$_1$
plv$_1$ $(\cdot,\cdot)$
Store: *pre-linkage seeds and pre-linkage values.*
Shuffle
Collate
RA
DB
Store: *device's enrollment certificate and hash of all RA-to-PCA certificate requests.*
Step 2. Handle Request at RA:
Validate
Butterfly key expansion
Collect for shuffling
Step 6. Bundle at the RA:
Collate certificates for a device
Make bundle available to device (download)
LOP
Step 1. Create Request:
Include two caterpillar keys: $A, J, f_k(\cdot), f_e(\cdot)$
Sign and encypt to RA
Device

Figure 3.5. Certificate provisioning process.

- **Step 1**: In order to build a request, the device generates butterfly key seeds, signs it with its enrollment certificate, attaches it to the request, and encrypts it before sending it to the RA. The device then uses LOP to communicate the request to the RA. The LOP serves as a request pass-through device. The request appears to the RA as coming from the LOP because it masks the device's identifiers (such the IP address) by substituting its own with them.
- **Step 2**: The RA decrypts the request, authenticates the devices by validating the enrollment certificate, and checks to make sure the devices are not revoked. It also confirms that this is the device's only request. If every check is successful, the RA acknowledges the device and expands the butterfly key. If not, the request is rejected by the RA. Along with the pre-linkage value sets that it has gotten from the LAs, the RA gathers a number of these requests from various devices. When there are enough of these requests, the RA rearranges the individual expanded requests.
- **Step 3**: Each request for a specific pseudonym certificate from the RA to the PCA includes a to-besigned certificate, a response encryption public key, one encrypted pre-linkage value from each of the LAs $(plv_1(i,j), plv_2(i,j))$, and the hash of the RA to PCA pseudonym certificate request.
- **Step 4**: The linkage value $lv(i,j) = plv_1(i,j) \oplus plv_2(i,j)$ is calculated by the PCA after decrypting the pre-linkage values. The to-be-signed certificate is then given the linkage value, and it is implicitly signed to create a pseudonym certificate. Then, a value for private key reconstruction is produced. The response encryption public key is then used to encrypt the pseudonym certificate and the private key reconstruction value.
- **Step 5**: The PCA signs the encrypted packet that was produced in step 4 and sends it to the RA. This prevents a man-in-the-middle attack in which a RA employee changes the valid response encryption key with another key for which an insider at the RA is aware of the private key, enabling the RA to access the contents of the pseudonym certificate, including the linkage value.
- **Step 6**: The RA gathers the encrypted packages for a week and bundles them into so-called batches for a certain device. The device can download the batches from the RA.

3.5.3 Misbehavior reporting

The process of identifying misbehaving devices and determining whether suspicious activities are actually caused by misbehavior is known as misbehavior investigation. It is noted that this part is reported only for completeness since it was not possible to model it in Tamarin-prover.

It is a process that is started by the Misbehavior Detection algorithm running in the MA dependent on inputs from one LA and the PCA. With this division, checks and balances are added to the system. To ensure the highest level of privacy protection, it is recommended a process that restricts the number of requests PCA and LA accept as well as the quantity of data returned to MA. Last but not least, is is advised that the SCMS Manager frequently examines these log files and that PCA and LA maintain records of each request. It is shown a thorough, step-by-step explanation of this method in the paragraphs that follow. It should be noted that Steps 1 and 2—which address Global Misbehavior Detection and Misbehavior Reporting, respectively—are included for completeness.

- **Step 1**: The MA receives misbehavior reports, including a reported pseudonym certificate with linkage value $lv = plv_1 \oplus plv_2$.
- **Step 2**: To decide which reported pseudonym certificates may be of interest, or for which pseudonym certificates it needs to retrieve linkage information, the MA runs global misbehavior detection algorithms.
- **Step 3**: The MA asks the PCA to translate the encrypted pre-linkage values (plv_1 , plv_2) from the PCA's database to the linkage values (lv) of the identified pseudonym certificates. The PCA sends MA the pre-linkage values encrypted.
- **Step 4**: The MA asks LA_1 or LA_2 to determine whether a group of encrypted plv_1 (or plv_2, respectively) point to the same device. The LA won't react unless there are more encrypted plv pointing at the same device than a predetermined threshold (for example, five).

3.5.4 Revocation

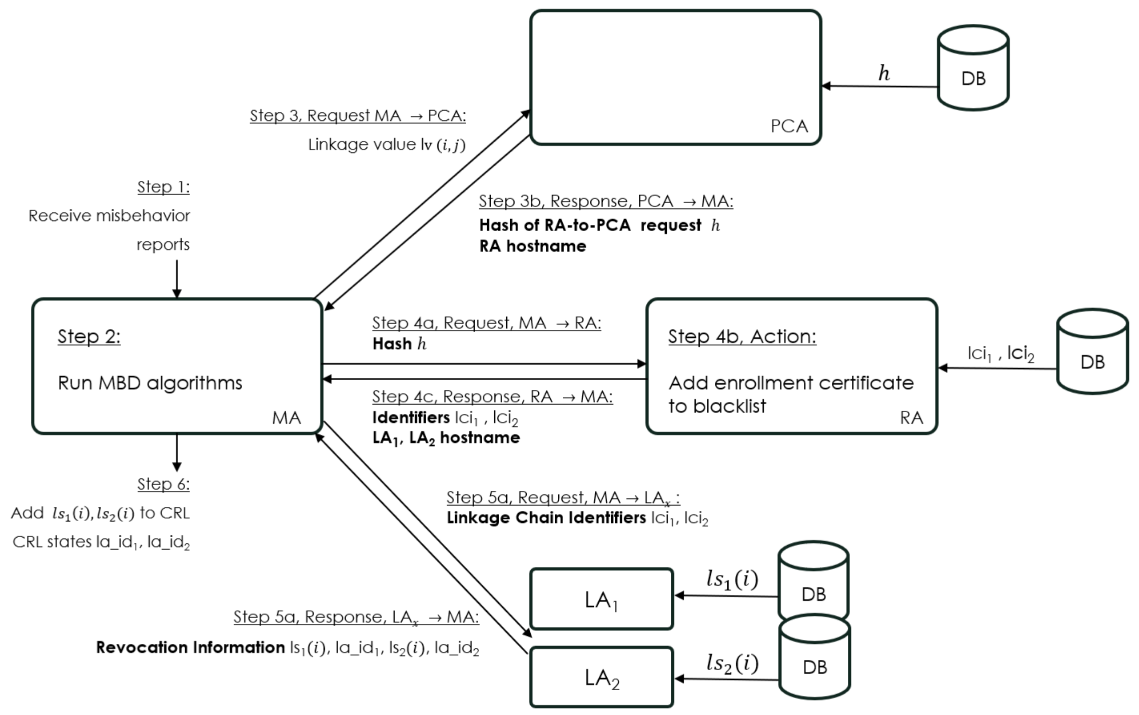

PCA
h
DB
Step 3, Request MA → PCA:
Linkage value lv (i, j)
Step 3b, Response, PCA → MA:
Hash of RA-to-PCA request h
RA hostname
Step 1:
Receive misbehavior reports
Step 2:
Run MBD algorithms
MA
Step 4a, Request, MA → RA:
Hash h
Step 4b, Action:
Add enrollment certificate to blacklist
RA
lci1 , lci2
DB
Step 4c, Response, RA → MA:
Identifiers lci1 , lci2
LA1, LA2 hostname
Step 6:
Add ls1(i), ls2(i) to CRL
CRL states la_id1, la_id2
Step 5a, Request, MA → LAx :
Linkage Chain Identifiers lci1, lci2
LA1
ls1(i)
DB
LA2
ls2(i)
DB
Step 5a, Response, LAx → MA:
Revocation Information ls1(i), la_id1, ls2(i), la_id2

Figure 3.6. Pseudonym certificate revocation process.

If the MA detect during the Misbehavior Investigation that the device was actually misbehaving, it revokes and blacklists the device. The Revocation and Blacklisting method, which identifies the linkage seeds and the enrollment certificate corresponding to a pseudonym certificate, is then thoroughly explained. This procedure is shown in figure 3.6, with the first two steps provided by the Misbehavior Investigation.

- **Step 3**: The MA asks the PCA to map the hash value of the RA-to-PCA pseudonym certificate request to the linkage value lv of the identified pseudonym certificate. This value as well as the hostname of the relevant RA are returned to the MA by the PCA.

- **Step 4**: The RA-to-PCA pseudonym certificate request's hash value is sent to the RA by the MA. The RA can add the hash value to its blacklist after mapping it to the associated enrollment certificate. The enrollment certificate is kept secret from the MA by the RA. The MA receives the following data from the RA, which it utilizes to gather the data required for revocation:
  - The LAs' hostnames used to create the pseudonym certificates linkage value,
  - A collection of LCIs for every LA. The linkage chain and the underlying linkage seed can both be looked up by the LA via an LCI. If a device possesses certificates from many separate linkage chains, which we consider an exception, the RA only returns one linkage chain identifiers.

- **Step 5**: The MA asks the LA_1 (or LA_2) to map the linkage seed $\mathrm{ls}_1(\mathrm{i})$ (or $\mathrm{ls}_1(\mathrm{i})$) to the currently valid time period. Both LAs give the MA their linkage seed back. Additionally, every LA gives the MA its linkage authority ID ($\mathrm{la\_id}_\mathrm{i}$). It should be noted that only the forward linkage seeds (i.e., ls1(j) for j >= i) can be derived given a linkage seed ls1(i) and the matching $\mathrm{la\_id}_\mathrm{i}$, maintaining the backward secrecy of the revoked device.

- **Step 6**: The CRL is updated by MA with the linkage seeds $\mathrm{ls}_1(\mathrm{i})$ and $\mathrm{ls}_2(\mathrm{i})$, as well as the associated pair of LA IDs $\mathrm{la\_id}_1$, $\mathrm{la\_id}_2$. The current time period i is identified globally by the CRL. The CRL may combine entries with the same LA ID pair to maximize efficiency and reduce over-the-air data usage. The CRL is then published and signed by the MA's CRLG.

Chapter 4

Objective of the thesis

The Security Credential Management System (SCMS) is a protocol developed by Crash Avoidance Metrics Partners LLC (CAMP) to provide secure communication and credential management for connected vehicles. As vehicles become increasingly connected and autonomous, ensuring the security and privacy of communication between vehicles and infrastructure becomes crucial. SCMS aims to address these concerns by providing a robust and trustworthy framework for managing security credentials and facilitating secure communication in vehicular networks. The main focus of this thesis work is on formal verification of the protocol described above. Obviously, in order to do this, a preliminary phase of in-depth study of the protocol was necessary in order to decide what to model and what to simplify so as to maintain the highest degree of reliability.

Formal verification of a protocol serves to ensure that the protocol is correct, secure and respects certain specified properties. Also, allows the behaviour of the protocol to be analysed rigorously, identifying potential vulnerabilities, design errors and potential security breaches. The formal verification will be carried out with Tamarin-Prover, an automated tool for the formal analysis of security protocols. It employs the symbolic model approach, which allows the exploration of all possible protocol executions. The formal verification process using Tamarin-Prover involves modeling the SCMS protocol as a set of rules and specifying the desired security properties to be analyzed. One of the strengths of Tamarin-Prover lies in its ability to model and reason about an adversary's capabilities using the Dolev-Yao model.

The main objective that want to be stressed, beyond everything that has been introduced, is the verification of the privacy-related security properties stated. What differentiates this protocol from others in the literature is that it aims to guarantee unlinkability and anonymity. With formal verification, we want to check whether this is indeed the case.

The phases in which the work can be split are:

- Modelling
- Rule Design in tamarin

- Sanity checks
- Security Properties

Modelling: Study the protocol and all its various steps. Decide which components to omit or transform so as to make them compatible with Tamarin-Prover rules.

Rule Desing in Tamarin-Prover: once the models with the various assumptions and simplifications were obtained, the design phase begun in which the various steps were translated into Tamarin rules.

Sanity checks: sanity checks are written to verify that the translation into Tamarin rules is correct and the protocol succeeds in exchanging messages correctly and in reaching the end of all the various steps.

Security Properties: the most important part are the verification of security properties, which is the main purpose of formal verification, i.e., that the protocol guarantees them and prevents possible attacks due to modeling errors are not possible. Specifically, authenticity, anonymity and unlinkability will be analyzed.

Chapter 5

SCMS - Design Model in Tamarin

In the following chapter will be presented how the protocol has been modeled to be adapted and analyzed by Tarmarin-prover. Specifically, the various choices and assumptions made, will be explained in order to ensure the highest possible fidelity to the original protocol.

As deeply described in Chapter 3, the protocol is composed by four main phases:

- Bootstrapping – initial phase where the node obtain the information to communicate inside the protocol.
- Certificate provisioning – in this phase the node, after already been initialized in the previous phase, obtain an enrollment certificate and later a pseudonym certificate.
- Misbehavior reporting – this part will not be modeled since in the original protocol it consists of running software that monitors the messages exchangeg and alerts in case of misbehavior.
- Revocation – after the node has been detected as misbehaving all the SCMS actor will cooperate to revoke the certificate issued to the reported node.

5.1 Preliminary steps and Assumptions

Starting from the model in Fig. 3.1 some assumptions were necessary to best model the protocol and be able to test it in Tamarin. In particular:

- Electors: electors are used to endorse or revoke a Root Certificate Authority (RCA) by signing ballots. It is assumed that the RCA is trusted.
- Intermediate CA: the main purpose is to shield the Root CA from elevated traffic and attacks, so it has been decided to merge it inside the Root CA and have a single entity.

- Enrollment CA and DCM: ECA provide the enrollment certificate for the device to authenticate itself with the RA while the DCM communicate only with the ECA by means of protected channel. In tamarin this is translated by using protected channels and this phase, in a real scenario, involve the device phisically going to the CA to obtain the certificate. To avoid this complexity that is irrelevant to the verification, it has been decided to delegate these function to the Root CA.

- Location Obscurer Proxy: prevents the device from being traceable through source addresses on the network

The model obtained is the one if Fig. 5.1

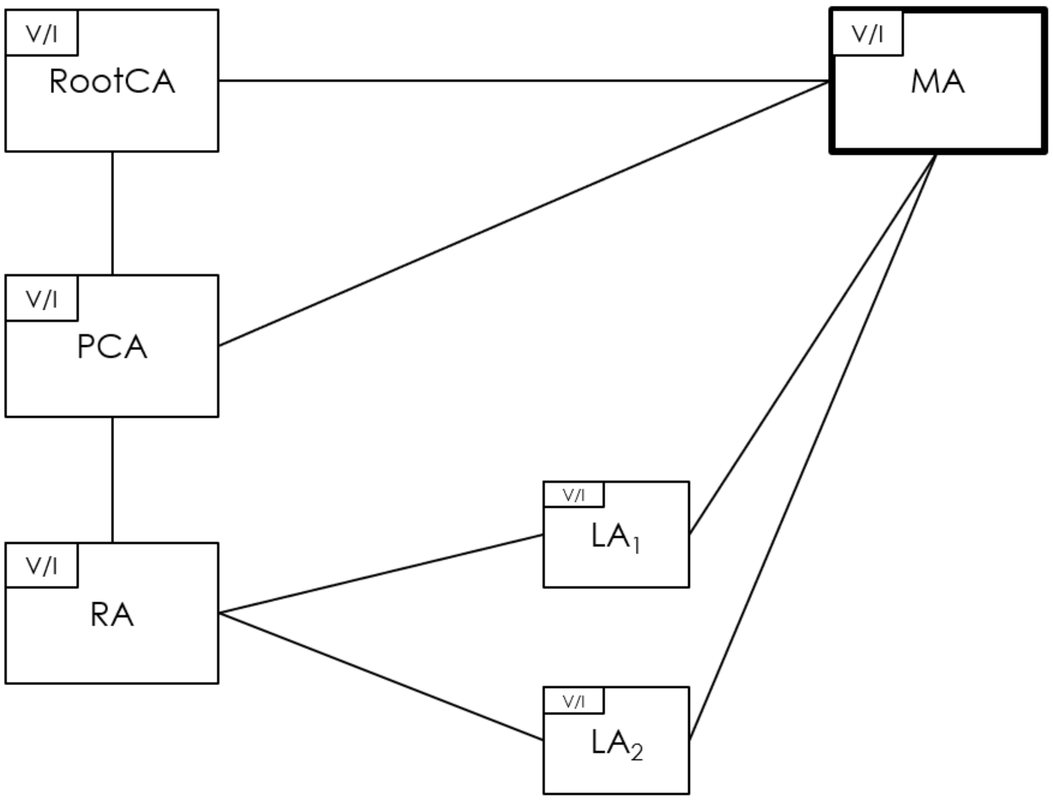


Figure 5.1. Simplified model.

5.2 Tamarin theory setup

Before explaining the four phases of the protocol, here will be introduced the features necessary for proper modeling of the protocol, detailed in Section 2.4.

Builtins

```
builtins: signing, asymmetric-encryption, hashing
```

Functions

User-defined functions have been declared to model the Butterfly Key Expansion (section 3.4)

```
functions: epk/1, esign/2, everify/3, eplus/3, etrue/0,
    eenc/3, edec/3, XOR/2
```

- `epk`: in the BKE the vehicle requesting the pseudonym certificate, submit some parameters, in our case represented by lowercase characters. These values are needed to craft the vehicle's private key. On the other hand the entity receiving the request will multiply these values with a value G, representing a point in a curve, to obtain the public key term associated to the terms. The function `epk` serves to obtain the terms that make up the public key of the vehicle.
- `esign`: is the function devoted to perform the signing of messages from the device with the pseudonym certificate obtained.
- `everify`: is the twin function to the traditional `verify`, used in this case to verify the authenticity of the signature when a message is signed with the pseudonym certificate.
- `eplus`: this function is used to "sum" the terms that make up the public and private key of the pseudonym certificate.
- `etrue`: is the twin term but in the case of verifying operations related to the butterfly key expansion.
- `edec`: it decrypt the messages encrypted with the pseudonym certificate's public key
- `XOR`: it is a simple XOR function replacing the built-in one. This is done to avoid adding complexity at the beginning by uploading theories not useful in our case.

Equations

Used to model properties of the functions defined

```
functions: epk/1, esign/2, everify/3, eplus/3, etrue/0,
    eenc/3, edec/3, XOR/2
```

equations: everify(esign(m, eplus(a, fpk, c)), m, eplus(epk(a), epk(fpk), epk(c))) = true, edec(eenc(m, epk(a), epk(b)), a, b) = m

Restriction

OnlyOnceRestriction force the rule that is evoking it to be executed once. By adding OnlyOnce() as an action fact for the rule, and adding this restriction will make the rule to execute only once. The result is restrict the set of traces of that particular rule to only one and it is particularly useful when dealing with the entities creation because in our scenario we must have a single entity of that particular type.

Equality is used when do not want to employ pattern-matching directly but still want to assure that the decryption of an encrypted value is the original value. This indicates that both of its inputs have the same value for each instance of the Eq operation on the trace. A straightforward example can be one related verification of a signature, where in the first member we put the equation relating to the signature seen above, and this must return true, and in the second member we put the value we wish to compare, in this case true. If both values are equal then the equality succeeds and Tamarin restrict the set of traces to only those in which this property holds.

DeviceCanGetOnlyOneEnrollmentCertificate is declared to make sure a device can obtain only one enrollment certificate. This restriction force the set of traces to only those where a device can get only one enrollment certificate both to avoid unnecessary complexity and also because once an enrolment certificate has been obtained, it cannot have any more.

CertificateRevocation is declared only on the revocation part and the purpose is to avoid to start the revocation for a device already revoked. Tamarin restrict the set of traces where the rule that stated CheckCertificateIsNotRevoked, in a precedent state does not exist a rule that invoked CertificateRevoked on the same certificate in a preceding instant of time.

```
restriction OnlyOnceRestriction:
 "All name #i #j. OnlyOnce(name)@#i & OnlyOnce(name)@#j
    ==> #i = #j"

restriction Equality:
  "All m n #i. Eq(m,n)@i ==> m = n"

restriction DeviceCAnGetOnlyOneEnrollmentCertificate:
  "All id #i #j. FirRequestFromThisId(id)@#i &
     FirRequestFromThisId(id)@#j ==> #i = #j"

restriction CertificateRevocation:
"All certpk #i. CheckCertificateIsNotRevoked(certpk)@i
   ==>
  not(Ex #j. CertificateRevoked(certpk)@j & j<i )"
```

5.2.1 Support rules

Here will be presented the rules that will support all the protocol, in the specific the private channels and the entities definition.

Keys creation

Each entity, in order to communicate within the protocol, will need a key pair, public and private. These are generated through this rule.

```
rule Register_pk:
[ Fr(~ltk) ]
--[]->
[ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)), Out(pk(~ltk)) ]
```

Secure Channel Rules

This particular type of channles are not under control of the Dolev-Yao adversay. The attacker cannot read or modify messages exchanged over this channel. Furthermore, there is authentication of the parts. '!Sec' is a persistent fact, meaning the attacker can reply messages. This channels are used when the communication takes place out-of-band or true a secure environment.

```
rule ChanOut_S:
[ Out_S( $A, $B, x ) ]
--[ ChanOut_S( $A, $B, x ) ]->
[ !Sec( $A, $B, x ) ]

rule ChanIn_S:
[ !Sec( $A, $B, x ) ]
--[ ChanIn_S( $A, $B, x ) ]->
[ In_S( $A, $B, x ) ]
```

Entity Init

the following rules will be used to create the various entities needed to set up the protocol and be able to communicate with each other. The structure will be the same for each entity. In particular, in the premise will be retrieved the long-term key, generated by the Register_pk rule, in the action fact is declared OnlyOnce, to be sure the entity is generated only once as previously explained and in conclusion will be generated the public and private key facts.

```
rule RootCA_init:
[ !Ltk($RootCA, ~ltkRootCA)
]
```

```
--[ OnlyOnce('RootCA'), RootCA_Initialised($RootCA) ]->
[ St_RootCA_1($RootCA, ~ltkRootCA),
  !RootCA_PK($RootCA, pk(~ltkRootCA)),
  !RootCA_SK($RootCA, ~ltkRootCA)
]

rule PCA_init:
[ !Ltk($PCA, ~ltkPCA),
  !RootCA_PK($RootCA, pkRootCA) //out of band
]
--[ OnlyOnce('PCA'), PCA_Initialised($PCA) ]->
[
  St_PCA_1($PCA, ~ltkPCA, pkRootCA),
  !PCA_PK($PCA, pk(~ltkPCA))
]

rule MA_init:
[
  !Ltk($MA, ~ltkMA),
  !RootCA_PK($RootCA, pkRootCA)
]
--[ OnlyOnce('MA'), MA_Initialised($MA) ]->
[
  St_MA_1($MA, ~ltkMA, pkRootCA),
  !MA_PK($MA, pk(~ltkMA))
]

rule RA_init:
[
  !Ltk($RA, ~ltkRA),
  !RootCA_PK($RootCA, pkRootCA)
]
--[ OnlyOnce('RA'), RA_Initialised($RA) ]->
[
  St_RA_1($RA, ~ltkRA, pkRootCA),
  !RA_PK($RA, pk(~ltkRA))
]

rule LA1_init:
[
  !Ltk($LA1, ~ltkLA1),
  !RootCA_PK($RootCA, pkRootCA)
]
--[ OnlyOnce('LA1'), LA1_Initialised($LA1) ]->
[
  St_LA1_1($LA1, ~ltkLA1, pkRootCA),
  !LA1_PK($LA1, pk(~ltkLA1))
]

rule LA2_init:
[
  !Ltk($LA2, ~ltkLA2),
  !RootCA_PK($RootCA, pkRootCA)
]
--[ OnlyOnce('LA2'), LA2_Initialised($LA2) ]->
[
  St_LA2_1($LA2, ~ltkLA2, pkRootCA),
  !LA2_PK($LA2, pk(~ltkLA2))
]
```

5.3 Bootstrapping

The first rule Device_init retrive the long-term key and generate a fresh id. In the conclusion will generate the first state and communicate the request trough a secure channel Out_S to the RA.

```
rule Device_init:
[ Fr(~id),
  !Ltk($Device, ~ltkDevice)
]
--[ Device_initialized($Device, ~id, ~ltkDevice),
    AskForEnrollmentCertificate(~id, pk(~ltkDevice))
  ]->
[ St_Device_1($Device, ~id, ~ltkDevice),
  Out_S($Device, $RootCA, <pk(~ltkDevice), ~id>)
]
```

RootCA_bootstrap_device represents the RA which receive the request from the Device through the private channel, retrive all public keys from the entities necessary to create the certificate, create the certificate and sign it with its private ltk and output it in the private channel.

```
rule RootCA_bootstrap_device:
  let certificate = <pkDevice, sign(pkDevice, ~ltkRootCA
    )>
  in
[ In_S($Device, $RootCA, <pkDevice, ~id>),
  !RootCA_SK($RootCA, ~ltkRootCA),
  !PCA_PK($PCA, pkPCA), //out of band
  !MA_PK($MA, pkMA),    //out of band
  !RA_PK($RA, pkRA)     //out of band
]
--[
    FirRequestFromThisId(~id),
    EnrollmentCertificateReleased(pkDevice, certificate)
  ]->
[  Out_S($RootCA, $Device, <pk(~ltkRootCA), pkPCA, pkMA,
    pkRA, certificate>)
]
```

Device_bootstrap is the rule where the device retrieve the certificate sent by the RA, verify that has been signed by it and output the new state that will be needed in the subsequent phase.

```
rule Device_bootstrap:
  let certificate = <pkdev, signature>
  in
[ In_S($RootCA, $Device, <pkRootCA, pkPCA, pkMA, pkRA,
   certificate>),
  St_Device_1($Device, ~id, ~ltkDevice)
]
--[
    Eq(pkdev, pk(~ltkDevice)),
    Eq(verify(signature, pkdev, pkRootCA), true),
    GotEnrollmentCertificate(~id, pk(~ltkDevice),
       certificate)
  ]->
[ !St_Device_2($Device, ~id, ~ltkDevice, pkRootCA, pkPCA
   , pkMA, pkRA, certificate) ]
```

5.4 Certificate Provisioning

This is the most complex scenario. The rules will map the scheme depicted in Fig. 5.2.

The rule CP_DeviceToRA_step1 represent the first step needed to receive a pseudonym certificate. The device generate a request id and all the parameters for the butterfly key

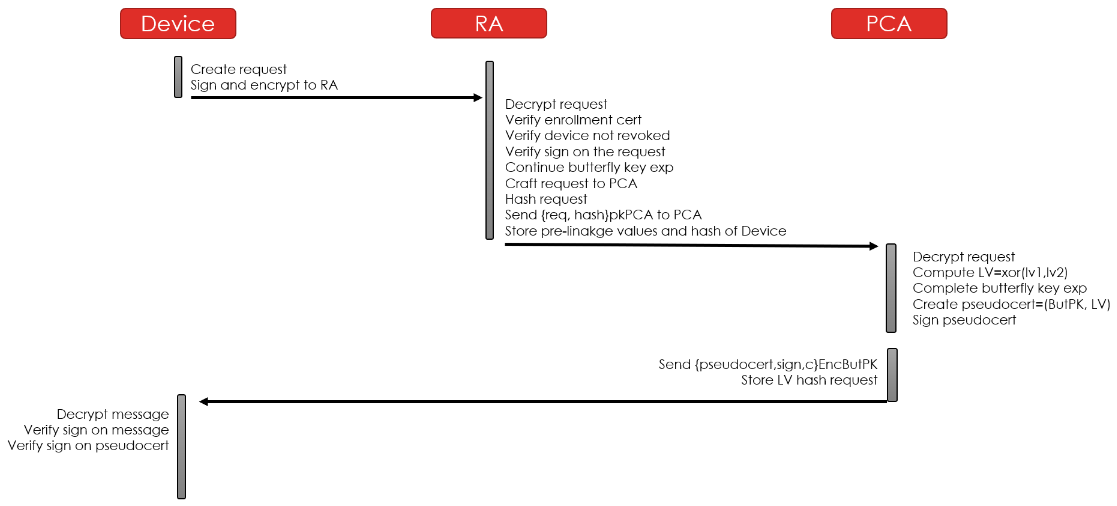

Device
RA
PCA
Create request
Sign and encrypt to RA
Decrypt request
Verify enrollment cert
Verify device not revoked
Verify sign on the request
Continue butterfly key exp
Craft request to PCA
Hash request
Send {req, hash}pkPCA to PCA
Store pre-linakge values and hash of Device
Decrypt request
Compute LV=xor(lv1,lv2)
Complete butterfly key exp
Create pseudocert=(ButPK, LV)
Sign pseudocert
Send {pseudocert,sign,c}EncButPK
Store LV hash request
Decrypt message
Verify sign on message
Verify sign on pseudocert

Figure 5.2. Certificate provisioning scheme.

expansion. Then include all these parameters in the request, sign it with its ltk and encrypt it with the public key of the RA. Then output the encrypted request on the public channel and create the new state St_Device_3.

```
rule CP_DeviceToRA_step1:
  let
    A = epk(~a)
    HH = epk(~hh)
    request = < A, HH, ~fk, ~fe >
    signed_request = sign(request, ~ltkDevice)
    mex = <request, signed_request, certificate>
    enc_request = aenc(mex, pkRA)
  in
  [
    !St_Device_2($Device, ~id, ~ltkDevice, pkRootCA,
       pkPCA, pkMA, pkRA, certificate),
    Fr(~a),
    Fr(~hh),
    Fr(~fk),
    Fr(~fe),
    Fr(~reqid)
  ]
  --[ OUT_CP_DeviceToRA_step1(mex),
      AskForPseudonymCertificate(~id, ~reqid, A, HH),
      Step1(~id, A, HH),
      OUT23(enc_request, pkRootCA)
  ]->
  [
    Out(enc_request),
    St_Device_3($Device, ~id, ~ltkDevice, pkRootCA,
       pkPCA, pkMA, pkRA, certificate, ~reqid, ~a, ~hh,
       ~fk, ~fe)
  ]
```

Rule CP_LA1_preLinkageValue and CP_LA2_preLinkageValue generate the pre-linkage values needed first by the RA, which will forward them to the PCA.

```
rule CP_LA1_preLinkageValue:
  let
    signplv1 = sign(~plv1, ~ltkLA1)
    encsignplv1 = aenc(<~plv1, signplv1>, pkPCA)

  in
  [
    !Ltk($LA1, ~ltkLA1),
    !PCA_PK($PCA, pkPCA),
    Fr(~plv1)
  ]
  --[ LA1PrelinageValueSent(encsignplv1) ]->
  [ Out(encsignplv1) ]

  rule CP_LA2_preLinkageValue:
  let
    signplv2 = sign(~plv2, ~ltkLA2)
    encsignplv2 = aenc(<~plv2, signplv2>, pkPCA)

  in
  [
    !Ltk($LA2, ~ltkLA2),
    !PCA_PK($PCA, pkPCA),
    Fr(~plv2)
  ]
  --[ LA2PrelinageValueSent(encsignplv2) ]->
  [ Out(encsignplv2) ]
```

In the rule CP_RAtoPCA_step2_3 the RA will retrieve the encrypted request sent by the device and the two encrypted pre-linkage values sent by LA1 and LA2.The RA verify that the certificate is valid, that the Device sending the request is not revoked and the signature on the request. Then will decrypt the request and will continue bke. After having derived all the parameters, it will craft the request for the PCA, and will include the hash for the request and encrypt all with the public key of the PCA. Eventually send the request for the PCA, and store the pre-linkage values associated to the device and the hash of the request. This will be needed to revoke the certificate.

```
rule CP_RAtoPCA_step2_3:
  let
    dec_request = adec(enc_request, ~ltkRA)
    request = fst(dec_request)
    signed_request = fst(snd(dec_request))
    Apub = fst(request)
    Hpub = fst(snd(request))
    fk = fst(snd(snd(request)))
    fe = snd(snd(snd(request)))
    certificate = snd(snd(dec_request))
    pkDevice = fst(certificate)
    rootSignature = snd(certificate)
    Bpub = epk(fk)
    Jpub = epk(fe)
    new_req = < Apub, Bpub, Hpub, Jpub, encsignplv1,
       encsignplv2 >
    h_new_req = h(new_req)
    enc_h_new_req = aenc(<new_req, h_new_req>, pkPCA)

  in
  [
    In(enc_request),
    In(encsignplv1),
    In(encsignplv2),
    !Ltk($RA, ~ltkRA),
    !PCA_PK($PCA, pkPCA)
  ]
  --[ OUT_CP_RAtoPCA_step2_3(<new_req, h_new_req>),
    IN_CP_RAtoPCA_step2_3(dec_request),
      Eq(verify(rootSignature, pkDevice, pk(~ltkRA)),
         true),
      CheckCertificateIsNotRevoked(pkDevice),
      Eq(verify( signed_request, request, pkDevice),
         true),
      RAPseudonymCertificateRequest(Apub, Bpub, Hpub,
         Jpub, h_new_req),


      Step2(Apub, Hpub, Bpub, Jpub, h_new_req),
      Step23_In(enc_request, pk(~ltkRA))
    ]->
  [
    Out(enc_h_new_req),
    RA_DEVICE_PRELINKAGE(pkDevice, encsignplv1,
       encsignplv2),
    RA_DEVICE_HASH( pkDevice, h_new_req )
  ]
```

In the rule CP_PCA_step4 first decrypt the request received by the RA, decrypt the pre-linkage values and verify the signature on them. Then compute the linkage value by xoring them and complete the butterfly key expansion. After that generate a fresh value c, that will be needed by the device, create the pseudonym certificate, sign it and encrypt with the encryption butterfly key. Lastly send the encrypted message and its signature and store the hash of the request and the linkage value.

```
rule CP_PCA_step4:
  let
    dec_h_new_req = adec(enc_h_new_req, ~ltkPCA)
    new_req = fst(dec_h_new_req)
    h_new_req = snd(dec_h_new_req)
    Apub = fst(new_req)
    Bpub = fst(snd(new_req))
    Hpub = fst(snd(snd(new_req)))
    Jpub = fst(snd(snd(snd(new_req))))
    encsignplv1 = fst(snd(snd(snd(snd(new_req)))))
    encsignplv2 = snd(snd(snd(snd(snd(new_req)))))
    signplv1 = snd(adec( encsignplv1 , ~ltkPCA))
    signplv2 = snd(adec( encsignplv2 , ~ltkPCA))
    lv1 = fst(adec( encsignplv1 , ~ltkPCA))
    lv2 = fst(adec( encsignplv2 , ~ltkPCA))
    lv = XOR(lv1,lv2)
    Cpub = epk(~c)
    PKsign = eplus(Apub, Bpub, Cpub)
    pseudocert = < PKsign, lv >
    signature = sign(pseudocert, ~ltkPCA)
    outmex = < pseudocert, signature, ~c >
    encoutmex = eenc(outmex, Hpub, Jpub)
    signencoutmex = sign(encoutmex, ~ltkPCA)
  in
  [
    !Ltk($PCA, ~ltkPCA),
    !LA1_PK($LA1, pkLA1),
    !LA2_PK($LA2, pkLA2),
    In(enc_h_new_req),
    Fr(~c)
  ]
  --[
    IN_CP_PCA_step4(dec_h_new_req),
    OUT_CP_PCA_step4(outmex),
    PseudonymCertificateReleased(h_new_req, PKsign, ~c,
       pseudocert),
    Eq( verify(signplv1, lv1, pkLA1), true ),
    Eq( verify(signplv2, lv2, pkLA2), true ),
    Step4(Cpub, PKsign)
  ]->
  [
    Out(< encoutmex, signencoutmex >),
    PCA_LV_HASH(lv, h_new_req),
    PCA_LV_PLVS(lv, encsignplv1, encsignplv2)
  ]
```

In the last rule the device receive the message and its signature from the PCA. Decrypt the message, retrieve tha pseudonym certificate and the reconstruction value c and update its private key. At the same time verify that the signature is correct and this concludes the certificate provisioning.

```
rule CP_Device_receive_step5:
  let
    outmex = edec(encoutmex, ~hh, ~fe)
    pseudocert = fst(outmex)
    signaturePCA = fst(snd(outmex))
    c = snd(snd(outmex))
    SK = eplus(~a, ~fk, c)
    PKpseudo = fst(pseudocert)
  in
  [
    St_Device_3($Device, ~id, ~ltkDevice, pkRootCA,
       pkPCA, pkMA, pkRA, certificate, ~reqid, ~a, ~hh,
       ~fk, ~fe),
    In(< encoutmex, signencoutmex >),
    !PCA_PK($PCA, pkPCA)
   ]
  --[ IN_CP_Device_receive_step5(outmex),
    Eq( verify(signencoutmex, encoutmex, pkPCA), true ),
    Eq( verify(signaturePCA, pseudocert, pkPCA ), true )
      ,
    GotPseudonymCertificate(~id, ~reqid, PKpseudo, c),

    Step5(~reqid, ~id, pseudocert),
    Recv( encoutmex, signencoutmex )
   ]->
  [
    !St_Device_4($Device, ~id, ~ltkDevice, pkRootCA,
      pkPCA, pkMA, pkRA, certificate, SK, pseudocert,
      signaturePCA)
  ]
```

5.5 Revocation

The revocation part of pseudonymous certificates will be explained in the following. Before proceeding, it is needed to make a few remarks. In the original protocol this part comes right after misbehavior detection, i.e., a node that begins to behave maliciously is detected. This is done through a detection algorithm that is run on OBEs. Since in Tamarin this part could not be modeled, it was shaped assuming that a detection message of a malicious node was received and then the revocation procedure starts.

Moreover it was necessary to split the two parts, certificate provisioning and revocation, for reasons due to the performance inadequacy of the machines in our possession because the protocol is very onerous.

Moreover, the initial part that includes setup, private channels, and entity initializations will not be re-explained as the same as the part of the certificate provisioning found in section 5.2.

The same modelling approach, already adopted for certificate provisioning will be followed, so the various steps will be described following the diagram in Figure 5.3.

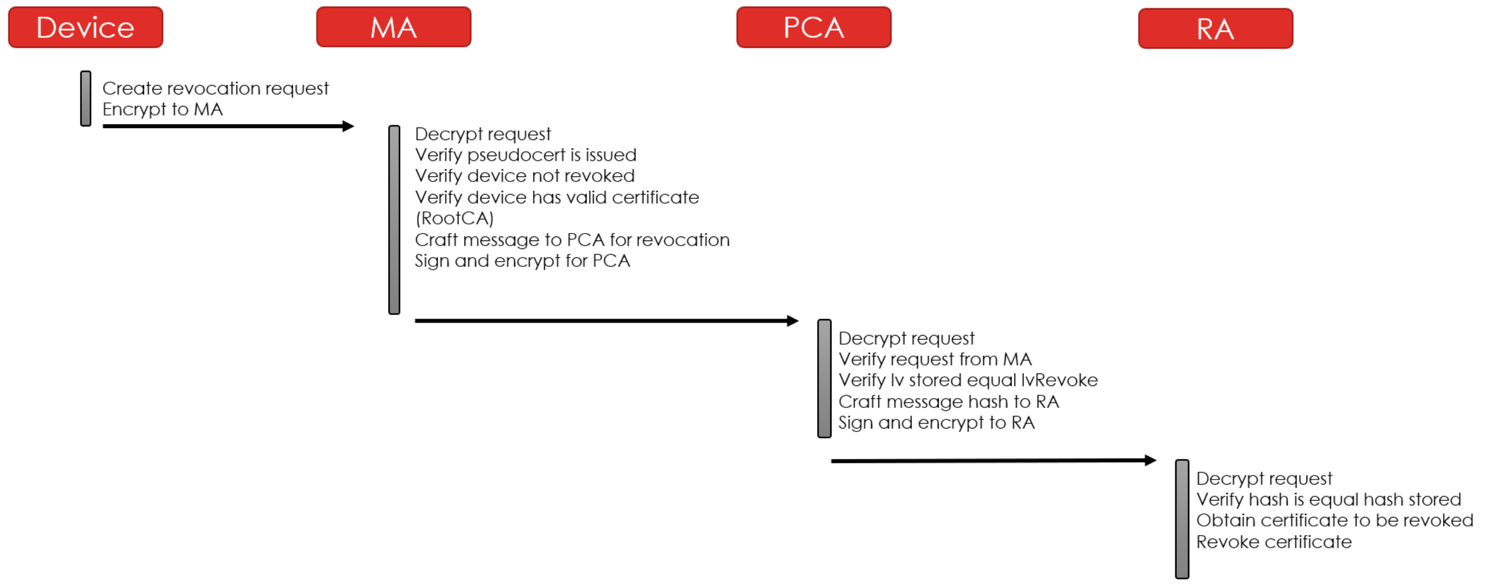

Device
MA
PCA
RA
Create revocation request
Encrypt to MA
Decrypt request
Verify pseudocert is issued
Verify device not revoked
Verify device has valid certificate (RootCA)
Craft message to PCA for revocation
Sign and encrypt for PCA
Decrypt request
Verify request from MA
Verify lv stored equal lvRevoke
Craft message hash to RA
Sign and encrypt to RA
Decrypt request
Verify hash is equal hash stored
Obtain certificate to be revoked
Revoke certificate

Figure 5.3. Revocation scheme.

The rule INIT_REVOCATION has the task of preparing the initial state of the revocation part, as if we were starting over from the end of certificate provisioning. That means, the device has successfully obtained its pseudonymous certificate and is ready to be able to communicate within the protocol. It generate fresh parameters for the butterfly key expansion, the pre-linkage and linkage values, craft the certificate and output the state of where the device is ready. Lastly, save the hash of the request that will be needed for the revocation.

```
rule INIT_REVOCATION:
    let
      Apub = epk(~a)
      Hpub = epk(~hh)
      Bpub = epk(~fk)
      Jpub = epk(~fe)
      Cpub = epk(~c)
      signplv1 = sign(~plv1, ~ltkLA1)
      encsignplv1 = aenc(<~plv1, signplv1>, pkPCA)
      signplv2 = sign(~plv2, ~ltkLA2)
      encsignplv2 = aenc(<~plv2, signplv2>, pkPCA)
      h_new_req = h(<Apub, Bpub, Hpub, Jpub, encsignplv1
          , encsignplv2>)
      lv = XOR(~plv1, ~plv2)
      PKsign = eplus(Apub, Bpub, Cpub)
      pseudocert = <PKsign, lv>
      signaturePCA = sign(pseudocert, ~ltkPCA)
      SK = eplus(~a, ~fk, ~c)
      pkDevice = pk(~ltkDevice)
      certificate = <pkDevice, sign(pkDevice, ~ltkRootCA
          )>
    in
    [
        Fr(~a),
        Fr(~hh),
        Fr(~fk),
        Fr(~fe),
        Fr(~c),
        Fr(~plv1),
        Fr(~plv2),
        Fr(~ltkDevice),
        !Ltk($PCA, ~ltkPCA),
        !Ltk($LA1, ~ltkLA1),
        !Ltk($LA2, ~ltkLA2),
        !PCA_PK($PCA, pkPCA),
        !RootCA_SK($RootCA, ~ltkRootCA)
    ]
    --[
        INIT_DONE('REVOCATION')
     ]->
    [
      !St_Device_1( ~ltkDevice, pkDevice, certificate,
         pseudocert, signaturePCA, SK),
      PCA_LV_HASH(lv, h_new_req),
      RA_DEVICE_HASH( pkDevice, h_new_req )
    ]
```

Device_Send_message simulate a vehicle sending a message inside the protocol. Generate a random mex and include in the out message the pseudonym certificate. Then, output everything in the public channel.

```
rule Device_Send_message:
  let
    signmex = esign(pseudocert, SK)
    outmex = <mex, signmex, pseudocert>
  in
  [ !St_Device_1( ~ltkDevice, pkDevice, certificate,
     pseudocert, signaturePCA, SK),
    Fr(mex)
  ]
  --[ Step1(),
    OUT_Device_Send_message(outmex)
   ]->
  [ Out(outmex)
  ]
```

Device_ask_revocation simulates the receipt of a malicious message and then once detected creates a request for MA. The steps include receiving the message, create the request including the misbehaving pseudonym certificate, sign the request with its own ltk and encrypt the whole message with the MA's public key.

```
rule Device_ask_revocation:
  let
    mex = fst(outmex)
    signmex = fst(snd(outmex))
    pseudocertToRevoke = snd(snd(outmex))
    PKsign = fst(pseudocertToRevoke)
    signPseudocertToRevoke = sign(pseudocertToRevoke, ~
       ltkDevice)
    mexToMA = <pseudocertToRevoke ,
       signPseudocertToRevoke , pseudocert , signaturePCA ,
        certificate >
    encmexToMA = aenc(mexToMA , pkMA)
  in
  [
    !St_Device_1( ~ltkDevice , pkDevice , certificate ,
       pseudocert , signaturePCA , SK),
    In(outmex),
    !MA_PK($MA , pkMA),
    !PCA_PK($PCA , pkPCA)
  ]
  --[
    Step2(),
    Eq( verify(signaturePCA , pseudocert , pkPCA), true ),
    Eq( everify(signmex , pseudocertToRevoke , PKsign),
       true )

   ]->
  [ Out(encmexToMA) ]
```

In this rule the MA decrypt the request, verify the signature on the pseudonym certificate that sended the revocation request, verify the signature on the pseudocert to revoke and that the certificate has not been already revoked. Then craft the request to be sended to the PCA by signing with its own ltk and encrypting the message with the public key of the PCA.

```
rule MA_request_to_PCA_mapLV:
  let
    mexToMA = adec(encmexFromMA , ~ltkMA)
    pseudocertToRevoke = fst(mexToMA)
    signPseudocertToRevoke = fst(snd(mexToMA))
    pseudocert = fst(snd(snd(mexToMA)))
    signaturePCA = fst(snd(snd(snd(mexToMA))))
    certificate = snd(snd(snd(snd(mexToMA))))
    pkDevice = fst(certificate)
    signCertificate = snd(certificate)

    messageToPCA = <pseudocertToRevoke >
    signMessageToPCA = sign(messageToPCA , ~ltkMA)
    outmexToPCA = <messageToPCA , signMessageToPCA >
    encOutmexToPCA = aenc(outmexToPCA , pkPCA)
  in
  [
    In(encmexFromMA),
    !Ltk($MA, ~ltkMA),
    !PCA_PK($PCA, pkPCA),
    !RootCA_PK($RootCA, pkRootCA)
  ]
  --[
    Step3(),
    Eq( verify(signaturePCA, pseudocert, pkPCA), true ),
    Eq( verify(signCertificate, pkDevice, pkRootCA),
       true ),
    Eq( verify(signPseudocertToRevoke,
       pseudocertToRevoke, pkDevice), true ),
    CheckCertificateIsNotRevoked(pkDevice)
  ]->
  [ Out(encOutmexToPCA)  ]
```

The PCA decrypt the request and check the signature. Then retrieve the linkage value associated with the pseudonym certificate to be revoked and the hash of the request when it has been issued. After that create the message intended for the RA including the hash of the reqeust and sign and encrypt it.

```
rule PCA_receive_request:
// pseudocert <PK, lvreceived>
  let
    outmexToPCA = adec(encOutmexToPCA, ~ltkPCA)
    messageToPCA = fst(outmexToPCA)
    signMessageToPCA = snd(outmexToPCA)
    pseudocertToRevoke = messageToPCA
    lvToRevoke = snd(pseudocertToRevoke)
    signHashReqToRA = sign(h_new_req, ~ltkPCA)
    outMexToRA = <h_new_req, signHashReqToRA>
    encSignHashReqToRA = aenc(outMexToRA, pkRA)
  in
  [ In(encOutmexToPCA),
    !Ltk($PCA, ~ltkPCA),
    !MA_PK($MA, pkMA),
    !RA_PK($RA, pkRA),
    PCA_LV_HASH(lv, h_new_req)
  ]
  --[
    Step4(),
    Eq( verify(signMessageToPCA, messageToPCA, pkMA),
       true ),
    Eq(lvToRevoke, lv)
  ]->
  [
    Out(encSignHashReqToRA)
  ]
```

In the last step the RA has everything needed to proceed with the revocation. Decrypt the request from the PCA, verify the hash received with the one stored during the issuing phase. When everyting matches proceed to revoke the certificate and store it.

```
rule RA:
  let
    outMexToRA = adec(encSignHashReqToRA, ~ltkRA)
    h_new_req = fst(outMexToRA)
    signHashReqToRA = snd(outMexToRA)

  in
  [
    In(encSignHashReqToRA),
    !Ltk($RA, ~ltkRA),
    !PCA_PK($PCA, pkPCA),
    RA_DEVICE_HASH(pkDevice, houtreq)

   ]
  --[
    Step5(),
    Eq( verify(signHashReqToRA, h_new_req, pkPCA), true
       ),
    Eq(houtreq, h_new_req),
    Recv(pkDevice),
    CertificateRevoked(pkDevice)
  ]->
  [ ]
```

Chapter 6

Results

This chapter will show the results obtained from the analysis of the modelled protocol using the Tamarin-Prover tool. To do this, the lemmas have been written down, which in the tool correspond to the security properties we want the protocol to satisfy. As a first step, the sanity checks will be highlighted, which are necessary for the correct modelling and termination of the protocol under consideration.

6.1 Sanity checks

The necessity of suspecting the system or specification of having an error, even in the event that model checking is successful, has come to light more recently. Such suspects are mostly justified by the possibility of modeling or specification flaws. A simple test to determine quickly if a claim or the outcome of a computation can conceivably be accurate is known as a "sanity check". Sanity checks are intended to find such problems through additional automatic reasoning. The process of determining whether the content created is logical is straightforward. A sanity check's goal is to exclude some categories of obviously incorrect results, not to capture every possible mistake

6.1.1 Executability

The executability sanity check, is a preliminary check aimed at verifying whether a protocol, once executed, can successfully reach its intended termination point.

The main objective of the executability sanity check is to ascertain that the protocol is designed in such a way that it is executable without incurring blocking conditions or wrong paths and that it is able to terminate correctly after fulfilling its objectives.

Sanity checks: Certificate Provisioning

Lemma `sanity_check_Success` is characterised by 'exists-trace' which means that it is true if and only if there is at least one trace satisfying the property. Where by trace we mean a sequence of action facts. The trace of a protocol, records action facts from the rules that are applied during their execution. So, basically it says if exists at least a trace where Recv has been reached then the execution reach the end of the protocol run.

On the other hand lemma `exec` generate anche action fact called Step* for each rule in the protocol. This serves to 'force' the execution of the protocol by following the various steps in order, starting with the enrolment request and ending with the delivery of the pseudonym certificate.

```
lemma sanity_check_Success:
exists-trace
  "Ex #i encoutmex signencoutmex.
        Recv( encoutmex, signencoutmex ) @i
  "
lemma exec:
exists-trace
"Ex id A HH B C J hash PKsign reqid pseudocert #i1 #i2 #
   i4 #i5 .
  Step1(id, A, HH)@i1 &
  Step2(A, HH, B, J, hash)@i2 &
  Step4(C, PKsign)@i4 &
  Step5(reqid, id, pseudocert)@i5 &
  i1 < i2 & i2 < i4 & i4 < i5
"
```

Sanity checks: Revocation

Here the considerations regarding the two lemmas are equivalent to the previous case of Certificate Provisioning.

```
lemma sanity_check_Success:
    exists-trace
        "Ex #i pkDevice .
               Recv( pkDevice) @i
        "
```

```
lemma sanityCheckTerm:
exists-trace
    "Ex  #i1 #i2 #i3 #i4 #i5 .
        Step1()@i1 &
        Step2()@i2 &
        Step3()@i3 &
        Step4()@i4 &
        Step5()@i5 &
        i1 < i2 & i2 < i3 & i3 < i4 & i4 < i5
    "
```

6.1.2 Authentication

In our context, the authentication sanity check is an essential preliminary check to ensure that the entities involved in the communication process are able to authenticate each other securely and reliably. The authentication of messages exchanged between entities is crucial to prevent spoofing, replay and forgery attacks that could compromise the integrity and security of the communication. For each stage of the protocol, the authentication of the exchanged messages was verified, so specifically when the device sends the request to the RA, the forwarding of the request from the RA to the PCA, and finally the sending of the pseudonymous certificate to the requesting device.

The lemma states that if the `Authentic_DevRA` event occurs, then the `Send_DevRA` event necessarily occurred first. And similarly if the `Authentic_PCADev` event occurs, then there was necessarily the `Send_PCADev` event first. This is true in the case where no attacker has obtained an enrolment certitificate and the result is shown in fig. 6.1

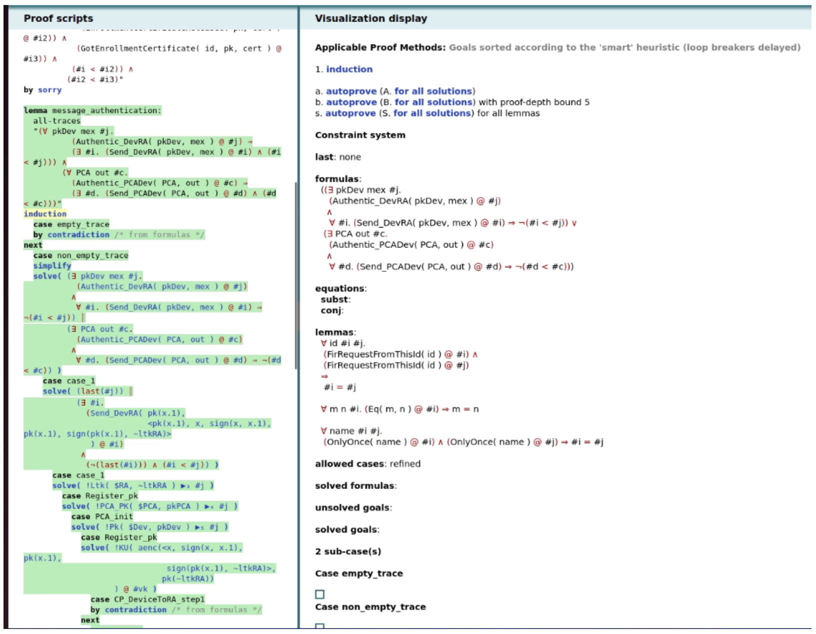
```
lemma message_authentication:
  "
  (All pkDev mex  #j. Authentic_DevRA(pkDev, mex) @j ==>
      Ex #i. Send_DevRA(pkDev, mex) @i &i<j)

  & (All PCA out #c. Authentic_PCADev(PCA, out) @c ==>
     Ex #d. Send_PCADev(PCA, out) @d &d<c)

  "
```

If the attacker, however, has obtained an enrolment certificate, he can request the pseudonym certificate from the RA. Fig. 6.2 show that the adversary can craft the request and send it to the RA in fig. 6.3.

```
rule Adversary_enrollment_certificate:
let cert = <pkAttacker, sign(pkAttacker, ~ltkRootCA)>
in
[ In(< attackerid, pkAttacker >),
  !RootCA_SK($RootCA, ~ltkRootCA)
]
--[
    FirRequestFromThisId(attackerid),
    EnrollmentCertificateReleasedForAttacker(pkAttacker,
        cert)
  ]->
[
  Out(cert)
]
```



Proof scripts

```
@ #i2)) ∧
          (GotEnrollmentCertificate( id, pk, cert ) @
#i3)) ∧
         (#i < #i2)) ∧
        (#i2 < #i3)"
by sorry

lemma message_authentication:
  all-traces
  "(∀ pkDev mex #j.
          (Authentic_DevRA( pkDev, mex ) @ #j) →
          (∃ #i. (Send_DevRA( pkDev, mex ) @ #i) ∧ (#i
< #j))) ∧
        (∀ PCA out #c.
          (Authentic_PCADev( PCA, out ) @ #c) →
          (∃ #d. (Send_PCADev( PCA, out ) @ #d) ∧ (#d
< #c)))"
induction
  case empty_trace
  by contradiction /* from formulas */
next
  case non_empty_trace
  simplify
  solve( (∃ pkDev mex #j.
           (Authentic_DevRA( pkDev, mex ) @ #j)
          ∧
           ∀ #i. (Send_DevRA( pkDev, mex ) @ #i) →
¬(#i < #j)) ‖
         (∃ PCA out #c.
           (Authentic_PCADev( PCA, out ) @ #c)
          ∧
           ∀ #d. (Send_PCADev( PCA, out ) @ #d) → ¬(#d
< #c)) )
    case case_1
    solve( (last(#j)) ‖
           (∃ #i.
             (Send_DevRA( pk(x.1),
                          <pk(x.1), x, sign(x, x.1),
pk(x.1), sign(pk(x.1), ~ltkRA)>
              ) @ #i)
            ∧
             (¬(last(#i)) ∧ (#i < #j)) )
      case case_1
      solve( !Ltk( $RA, ~ltkRA ) ▶₃ #j )
        case Register_pk
        solve( !PCA_PK( $PCA, pkPCA ) ▶₄ #j )
          case PCA_init
          solve( !Pk( $Dev, pkDev ) ▶₅ #j )
            case Register_pk
            solve( !KU( aenc(<x, sign(x, x.1),
pk(x.1),
                                  sign(pk(x.1), ~ltkRA)>,
                                 pk(~ltkRA))
                     ) @ #vk )
              case CP_DeviceToRA_step1
              by contradiction /* from formulas */
            next
```

Visualization display

```
Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (loop breakers delayed)

1. induction

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5
s. autoprove (S. for all solutions) for all lemmas

Constraint system

last: none

formulas:
 ((∃ pkDev mex #j.
   (Authentic_DevRA( pkDev, mex ) @ #j)
   ∧
   ∀ #i. (Send_DevRA( pkDev, mex ) @ #i) ⇒ ¬(#i < #j)) ∨
  (∃ PCA out #c.
   (Authentic_PCADev( PCA, out ) @ #c)
   ∧
   ∀ #d. (Send_PCADev( PCA, out ) @ #d) ⇒ ¬(#d < #c)))

equations:
 subst:
 conj:

lemmas:
 ∀ id #i #j.
  (FirRequestFromThisId( id ) @ #i) ∧
  (FirRequestFromThisId( id ) @ #j)
 ⇒
  #i = #j

 ∀ m n #i. (Eq( m, n ) @ #i) ⇒ m = n

 ∀ name #i #j.
  (OnlyOnce( name ) @ #i) ∧ (OnlyOnce( name ) @ #j) ⇒ #i = #j

allowed cases: refined

solved formulas:

unsolved goals:

solved goals:

2 sub-case(s)

Case empty_trace

☐
Case non_empty_trace

☐
```

Figure 6.1. Authentication sanity check success.

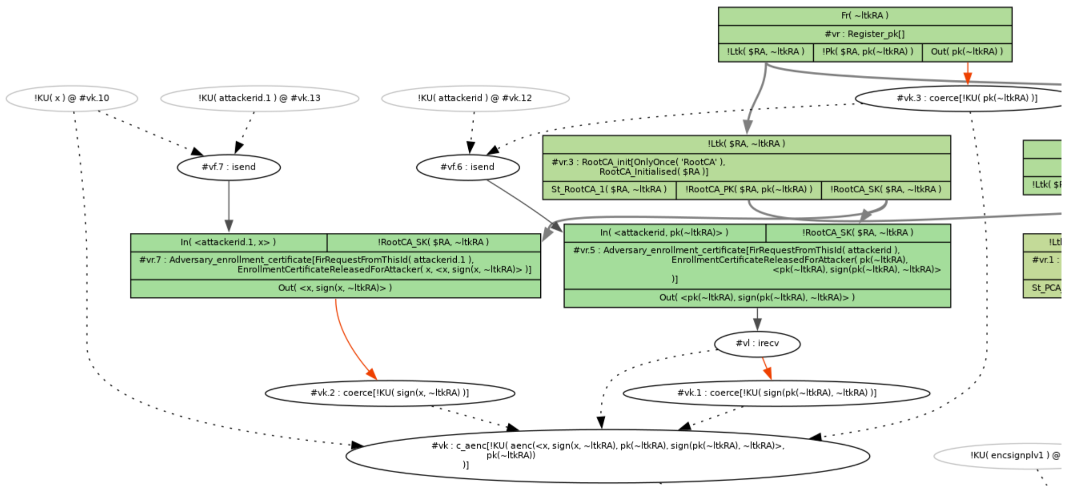

Fr(~ltkRA)
#vr : Register_pk[]
!Ltk($RA, ~ltkRA)
!Pk($RA, pk(~ltkRA))
Out(pk(~ltkRA))
!KU(x) @ #vk.10
!KU(attackerid.1) @ #vk.13
!KU(attackerid) @ #vk.12
#vk.3 : coerce[!KU(pk(~ltkRA))]
#vf.7 : isend
#vf.6 : isend
!Ltk($RA, ~ltkRA)
#vr.3 : RootCA_init[OnlyOnce('RootCA'), RootCA_Initialised($RA)]
St_RootCA_1($RA, ~ltkRA)
!RootCA_PK($RA, pk(~ltkRA))
!RootCA_SK($RA, ~ltkRA)
In(<attackerid.1, x>)
!RootCA_SK($RA, ~ltkRA)
#vr.7 : Adversary_enrollment_certificate[FirRequestFromThisId(attackerid.1), EnrollmentCertificateReleasedForAttacker(x, <x, sign(x, ~ltkRA)>)]
Out(<x, sign(x, ~ltkRA)>)
In(<attackerid, pk(~ltkRA)>)
!RootCA_SK($RA, ~ltkRA)
#vr.5 : Adversary_enrollment_certificate[FirRequestFromThisId(attackerid), EnrollmentCertificateReleasedForAttacker(pk(~ltkRA), <pk(~ltkRA), sign(pk(~ltkRA), ~ltkRA)>)]
Out(<pk(~ltkRA), sign(pk(~ltkRA), ~ltkRA)>)
#vl : irecv
#vk.2 : coerce[!KU(sign(x, ~ltkRA))]
#vk.1 : coerce[!KU(sign(pk(~ltkRA), ~ltkRA))]
#vk : c_aenc[!KU(aenc(<x, sign(x, ~ltkRA), pk(~ltkRA), sign(pk(~ltkRA), ~ltkRA)>, pk(~ltkRA)))]
!KU(encsignplv1) @

Figure 6.2. Authentication - attacker with enrollment certificate - 1.

Regarding the messages exchanged between the RA and PCA since no signature is present in this step, the authentication property does not hold, as shown in fig. 6.4. The adversy can craft the message 6.5 and send it to the PCA 6.3, invalidating the authentication property.

```
lemma message_authentication_RAtoPCA:
```

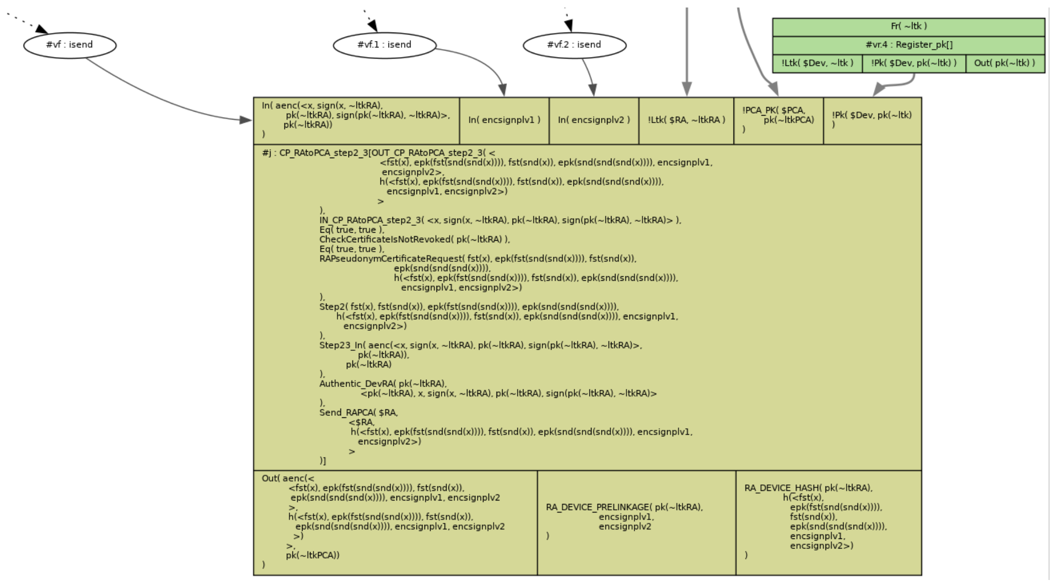

#vf : isend
#vf.1 : isend
#vf.2 : isend
Fr(~ltk)
#vr.4 : Register_pk[]
!Ltk($Dev, ~ltk)
!Pk($Dev, pk(~ltk))
Out(pk(~ltk))
In(aenc(<x, sign(x, ~ltkRA), pk(~ltkRA), sign(pk(~ltkRA), ~ltkRA)>, pk(~ltkRA)))
In(encsignplv1)
In(encsignplv2)
!Ltk($RA, ~ltkRA)
!PCA_PK($PCA, pk(~ltkPCA))
!Pk($Dev, pk(~ltk))
#j : CP_RAtoPCA_step2_3[OUT_CP_RAtoPCA_step2_3(< <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2>, h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2>) >),
IN_CP_RAtoPCA_step2_3(<x, sign(x, ~ltkRA), pk(~ltkRA), sign(pk(~ltkRA), ~ltkRA)>),
Eq(true, true),
CheckCertificateIsNotRevoked(pk(~ltkRA)),
Eq(true, true),
RAPseudonymCertificateRequest(fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2>)),
Step2(fst(x), fst(snd(x)), epk(fst(snd(snd(x)))), epk(snd(snd(snd(x)))), h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2>)),
Step23_In(aenc(<x, sign(x, ~ltkRA), pk(~ltkRA), sign(pk(~ltkRA), ~ltkRA)>, pk(~ltkRA)), pk(~ltkRA)),
Authentic_DevRA(pk(~ltkRA), <pk(~ltkRA), x, sign(x, ~ltkRA), pk(~ltkRA), sign(pk(~ltkRA), ~ltkRA)>),
Send_RAPCA($RA, <$RA, h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2>) >)]
Out(aenc(< <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2 >, h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2 >) >, pk(~ltkPCA)))
RA_DEVICE_PRELINKAGE(pk(~ltkRA), encsignplv1, encsignplv2)
RA_DEVICE_HASH(pk(~ltkRA), h(<fst(x), epk(fst(snd(x))), fst(snd(x)), epk(snd(snd(snd(x)))), encsignplv1, encsignplv2>))

Figure 6.3. Authentication - attacker with enrollment certificate - 2.

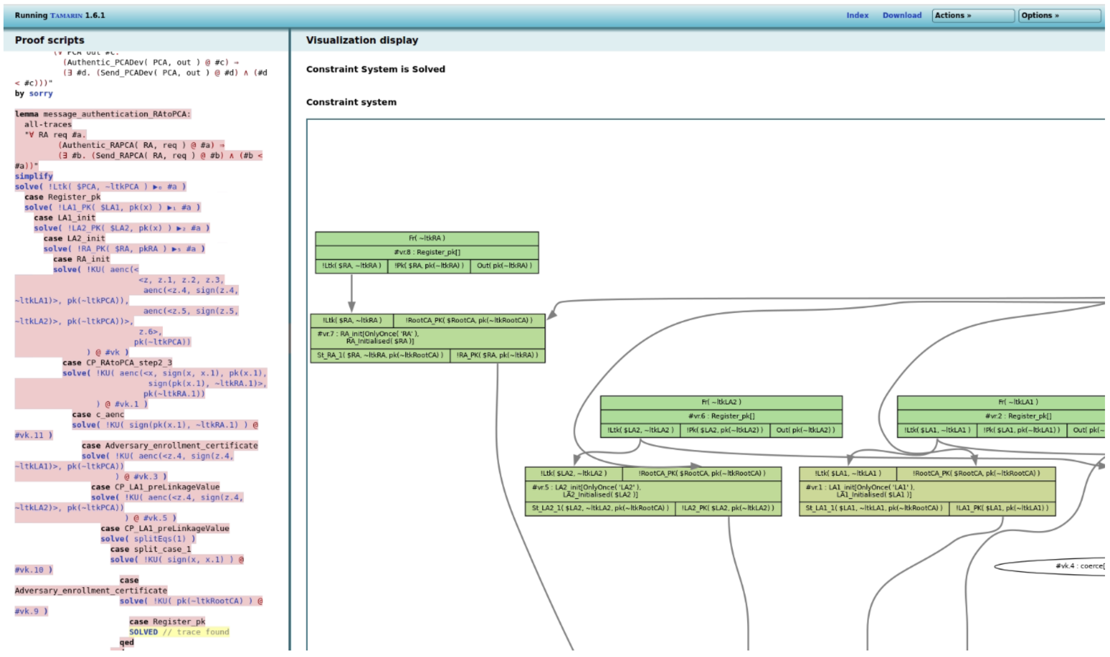
```
  "
  (All RA req  #a. Authentic_RAPCA(RA, req) @a ==> Ex #b
     . Send_RAPCA(RA, req) @b &b<a)

  "
```

6.1.3 Bootstrapping success

This sanity check relates to the first phase of the SCMS protocol, i.e. the bootstrap phase of the device. Like the previous lemmas, it is characterised by the ’exists-trace’ clause and states that the property holds if a request is made to obtain the enrollment certificate (AskForEnrollmentCertificate), that the RootCA issues it (EnrollmentCertificateReleased), and finally that the device correctly receives the (GotEnrollmentCertificate) and all these steps have occurred in order.

```
lemma sanity_check_Bootstrapping:
exists-trace
  "Ex #i #i2 #i3 id pk cert.
        AskForEnrollmentCertificate(id, pk) @i &
        EnrollmentCertificateReleased(pk, cert) @i2 &
        GotEnrollmentCertificate(id, pk, cert) @i3 &
        i < i2 &
        i2 < i3
```


Running Tamarin 1.6.1
Index
Download
Actions »
Options »
Proof scripts
Visualization display
Constraint System is Solved
Constraint system
Fr(~ltkRA)
#vr.8 : Register_pk[]
!Ltk($RA, ~ltkRA)
!Pk($RA, pk(~ltkRA))
Out(pk(~ltkRA))
!Ltk($RA, ~ltkRA)
!RootCA_PK($RootCA, pk(~ltkRootCA))
#vr.7 : RA_init[OnlyOnce('RA'), RA_Initialised($RA)]
St_RA_1($RA, ~ltkRA, pk(~ltkRootCA))
!RA_PK($RA, pk(~ltkRA))
Fr(~ltkLA2)
#vr.6 : Register_pk[]
!Ltk($LA2, ~ltkLA2)
!Pk($LA2, pk(~ltkLA2))
Out(pk(~ltkLA2))
Fr(~ltkLA1)
#vr.2 : Register_pk[]
!Ltk($LA1, ~ltkLA1)
!Pk($LA1, pk(~ltkLA1))
!Ltk($LA2, ~ltkLA2)
!RootCA_PK($RootCA, pk(~ltkRootCA))
#vr.5 : LA2_init[OnlyOnce('LA2'), LA2_Initialised($LA2)]
St_LA2_1($LA2, ~ltkLA2, pk(~ltkRootCA))
!LA2_PK($LA2, pk(~ltkLA2))
!Ltk($LA1, ~ltkLA1)
!RootCA_PK($RootCA, pk(~ltkRootCA))
#vr.1 : LA1_init[OnlyOnce('LA1'), LA1_Initialised($LA1)]
St_LA1_1($LA1, ~ltkLA1, pk(~ltkRootCA))
!LA1_PK($LA1, pk(~ltkLA1))
#vk.4 : coerce[

```
        (∀ PCA out #c.
          (Authentic_PCADev( PCA, out ) @ #c) ⇒
          (∃ #d. (Send_PCADev( PCA, out ) @ #d) ∧ (#d
< #c)))"
by sorry

lemma message_authentication_RAtoPCA:
  all-traces
  "∀ RA req #a.
         (Authentic_RAPCA( RA, req ) @ #a) ⇒
         (∃ #b. (Send_RAPCA( RA, req ) @ #b) ∧ (#b <
#a))"
simplify
solve( !Ltk( $PCA, ~ltkPCA ) ▶₀ #a )
  case Register_pk
  solve( !LA1_PK( $LA1, pk(x) ) ▶₁ #a )
    case LA1_init
    solve( !LA2_PK( $LA2, pk(x) ) ▶₂ #a )
      case LA2_init
      solve( !RA_PK( $RA, pkRA ) ▶₅ #a )
        case RA_init
        solve( !KU( aenc(<
                          <z, z.1, z.2, z.3,
                           aenc(<z.4, sign(z.4,
~ltkLA1)>, pk(~ltkPCA)),
                           aenc(<z.5, sign(z.5,
~ltkLA2)>, pk(~ltkPCA))>,
                          z.6>,
                         pk(~ltkPCA))
               ) @ #vk )
          case CP_RAtoPCA_step2_3
          solve( !KU( aenc(<x, sign(x, x.1), pk(x.1),
                            sign(pk(x.1), ~ltkRA.1)>,
                           pk(~ltkRA.1))
                 ) @ #vk.1 )
            case c_aenc
            solve( !KU( sign(pk(x.1), ~ltkRA.1) ) @
#vk.11 )
              case Adversary_enrollment_certificate
              solve( !KU( aenc(<z.4, sign(z.4,
~ltkLA1)>, pk(~ltkPCA))
                     ) @ #vk.3 )
                case CP_LA1_preLinkageValue
                solve( !KU( aenc(<z.4, sign(z.4,
~ltkLA2)>, pk(~ltkPCA))
                       ) @ #vk.5 )
                  case CP_LA1_preLinkageValue
                  solve( splitEqs(1) )
                    case split_case_1
                    solve( !KU( sign(x, x.1) ) @
#vk.10 )
                      case
Adversary_enrollment_certificate
                      solve( !KU( pk(~ltkRootCA) ) @
#vk.9 )
                        case Register_pk
                        SOLVED // trace found
                      qed
```

Figure 6.4. Authentication - RA to PCA.

```
  "
```

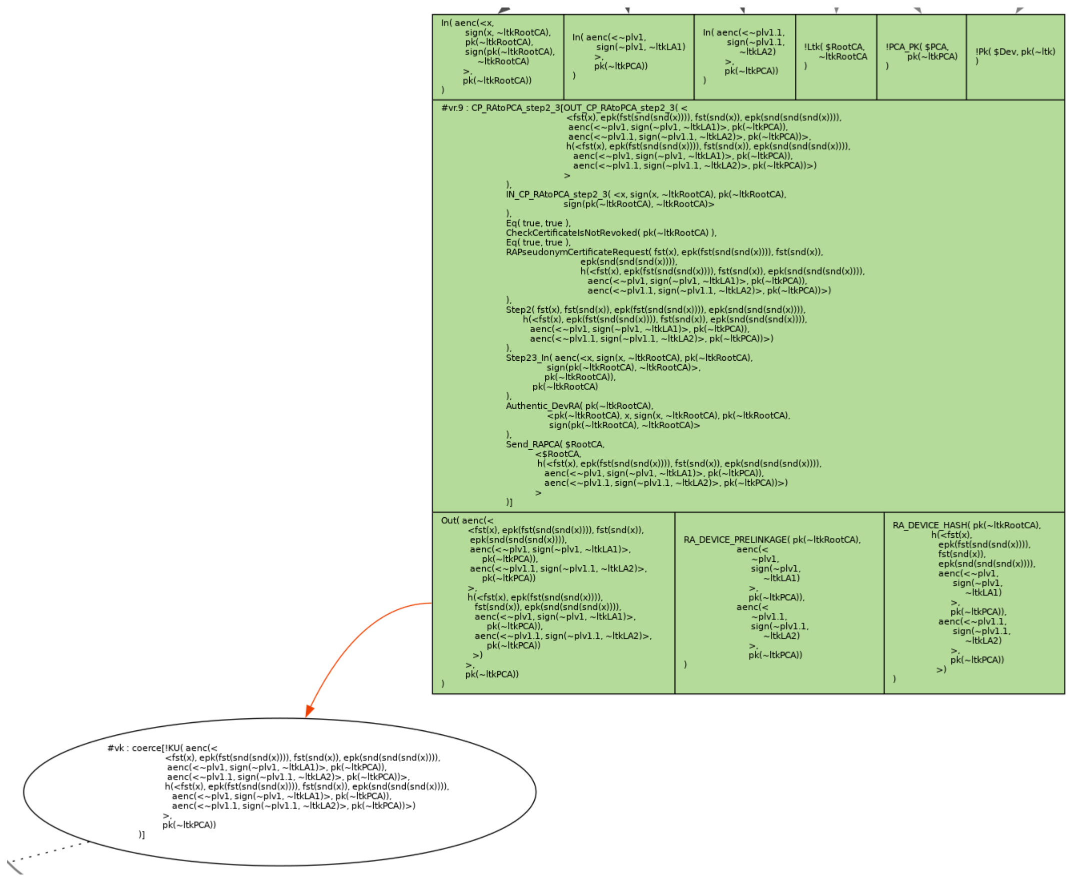
6.1.4 Partial deconstruction

Tamarin pre-calculate a list of potential sources for each fact and examines all rules and premises. It could happen that Tamarin is unable to solve where a fact come from, for some rules. In this case it says that is left a partial deconstruciton. Such partial deconstructions make automated proof creation more difficult and frequently result in non-termination.

These lemma are introduced to solve partial deconstructions.

Partial deconstruction certificate provisioning

```
lemma CP_step1_2_3 [sources]:
  " (All m #i.
    IN_CP_RAtoPCA_step2_3(m)@i
    ==>
      (
        (Ex #j. KU(m)@j & j<i)
      | (Ex #j. OUT_CP_DeviceToRA_step1(m)@j )
      )
    )
```



```
In( aenc(<x,
        sign(x, ~ltkRootCA),
        pk(~ltkRootCA),
        sign(pk(~ltkRootCA),
            ~ltkRootCA)
       >,
       pk(~ltkRootCA))
)
```

```
In( aenc(<~plv1,
        sign(~plv1, ~ltkLA1)
       >,
       pk(~ltkPCA))
)
```

```
In( aenc(<~plv1.1,
        sign(~plv1.1,
            ~ltkLA2)
       >,
       pk(~ltkPCA))
)
```

```
!Ltk( $RootCA,
     ~ltkRootCA
)
```

```
!PCA_PK( $PCA,
        pk(~ltkPCA)
)
```

```
!Pk( $Dev, pk(~ltk)
)
```

```
#vr.9 : CP_RAtoPCA_step2_3[OUT_CP_RAtoPCA_step2_3( <
                      <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))),
                       aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)),
                       aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>,
                      h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))),
                        aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)),
                        aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>)
                      >
           ),
           IN_CP_RAtoPCA_step2_3( <x, sign(x, ~ltkRootCA), pk(~ltkRootCA),
                      sign(pk(~ltkRootCA), ~ltkRootCA)>
           ),
           Eq( true, true ),
           CheckCertificateIsNotRevoked( pk(~ltkRootCA) ),
           Eq( true, true ),
           RAPseudonymCertificateRequest( fst(x), epk(fst(snd(snd(x)))), fst(snd(x)),
                        epk(snd(snd(snd(x)))),
                        h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))),
                          aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)),
                          aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>)
           ),
           Step2( fst(x), fst(snd(x)), epk(fst(snd(snd(x)))), epk(snd(snd(snd(x)))),
              h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))),
                aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)),
                aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>)
           ),
           Step23_In( aenc(<x, sign(x, ~ltkRootCA), pk(~ltkRootCA),
                  sign(pk(~ltkRootCA), ~ltkRootCA)>,
                  pk(~ltkRootCA)),
              pk(~ltkRootCA)
           ),
           Authentic_DevRA( pk(~ltkRootCA),
                  <pk(~ltkRootCA), x, sign(x, ~ltkRootCA), pk(~ltkRootCA),
                   sign(pk(~ltkRootCA), ~ltkRootCA)>
           ),
           Send_RAPCA( $RootCA,
                <$RootCA,
                 h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))),
                   aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)),
                   aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>)
                >
           )]
```

```
Out( aenc(<
       <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)),
        epk(snd(snd(snd(x)))),
        aenc(<~plv1, sign(~plv1, ~ltkLA1)>,
           pk(~ltkPCA)),
        aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>,
           pk(~ltkPCA))
       >,
       h(<fst(x), epk(fst(snd(snd(x)))),
         fst(snd(x)), epk(snd(snd(snd(x)))),
         aenc(<~plv1, sign(~plv1, ~ltkLA1)>,
            pk(~ltkPCA)),
         aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>,
            pk(~ltkPCA))
        >)
      >,
      pk(~ltkPCA))
)
```

```
RA_DEVICE_PRELINKAGE( pk(~ltkRootCA),
              aenc(<
                ~plv1,
                sign(~plv1,
                   ~ltkLA1)
               >,
               pk(~ltkPCA)),
              aenc(<
                ~plv1.1,
                sign(~plv1.1,
                   ~ltkLA2)
               >,
               pk(~ltkPCA))
)
```

```
RA_DEVICE_HASH( pk(~ltkRootCA),
          h(<fst(x),
            epk(fst(snd(snd(x)))),
            fst(snd(x)),
            epk(snd(snd(snd(x)))),
            aenc(<~plv1,
               sign(~plv1,
                  ~ltkLA1)
              >,
              pk(~ltkPCA)),
            aenc(<~plv1.1,
               sign(~plv1.1,
                  ~ltkLA2)
              >,
              pk(~ltkPCA))
           >)
)
```

```
#vk : coerce[!KU( aenc(<
                <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))),
                 aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)),
                 aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>,
                h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))),
                  aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)),
                  aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>)
               >,
               pk(~ltkPCA))
         )]
```

Figure 6.5. Authentication - RA to PCA message craft - 1.

```
    "
```

6.2 Secrecy

Secrecy is the property of information being protected from disclosure to unauthorized parties. In this particular case, states that the attacker can not obtain the long-term key of the vehicle. In particular, for all ltk at instant i such that ltk is secret does not exists a time j where the adversary knows it or it has been compromised and so Revealed. This property is always verified since the vehicle'id is used only during enrollment phase and exchanged in a secure environment. 6.7

```
lemma secrecy:
    "All ltk #i.
    Secret(ltk) @i ==>
     (not (Ex #j. K(ltk)@j))
```

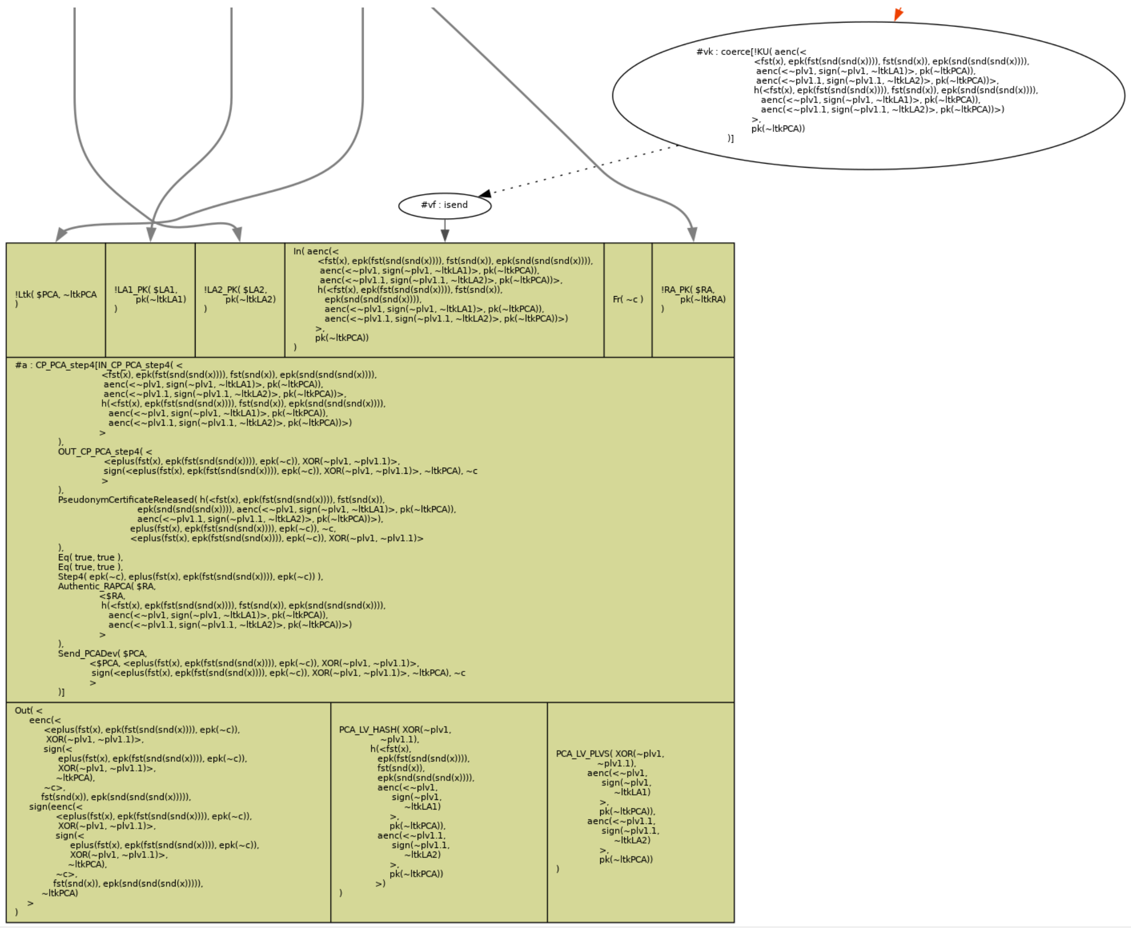

#vk : coerce[!KU(aenc(< <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>, h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>) >, pk(~ltkPCA)))]
#vf : isend
!Ltk($PCA, ~ltkPCA)
!LA1_PK($LA1, pk(~ltkLA1))
!LA2_PK($LA2, pk(~ltkLA2))
In(aenc(< <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>, h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>) >, pk(~ltkPCA)))
Fr(~c)
!RA_PK($RA, pk(~ltkRA))
#a : CP_PCA_step4[IN_CP_PCA_step4(< <fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>, h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>) >), OUT_CP_PCA_step4(< <eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, sign(<eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, ~ltkPCA), ~c >), PseudonymCertificateReleased(h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>), eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), ~c, <eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>), Eq(true, true), Eq(true, true), Step4(epk(~c), eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c))), Authentic_RAPCA($RA, <$RA, h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1)>, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2)>, pk(~ltkPCA))>) >), Send_PCADev($PCA, <$PCA, <eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, sign(<eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, ~ltkPCA), ~c >)]
Out(< eenc(< <eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, sign(< eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, ~ltkPCA), ~c>, fst(snd(x)), epk(snd(snd(snd(x))))), sign(eenc(< <eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, sign(< eplus(fst(x), epk(fst(snd(snd(x)))), epk(~c)), XOR(~plv1, ~plv1.1)>, ~ltkPCA), ~c>, fst(snd(x)), epk(snd(snd(snd(x))))), ~ltkPCA) >)
PCA_LV_HASH(XOR(~plv1, ~plv1.1), h(<fst(x), epk(fst(snd(snd(x)))), fst(snd(x)), epk(snd(snd(snd(x)))), aenc(<~plv1, sign(~plv1, ~ltkLA1) >, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2) >, pk(~ltkPCA)) >))
PCA_LV_PLVS(XOR(~plv1, ~plv1.1), aenc(<~plv1, sign(~plv1, ~ltkLA1) >, pk(~ltkPCA)), aenc(<~plv1.1, sign(~plv1.1, ~ltkLA2) >, pk(~ltkPCA)))

Figure 6.6. Authentication - RA to PCA message craft - 2.

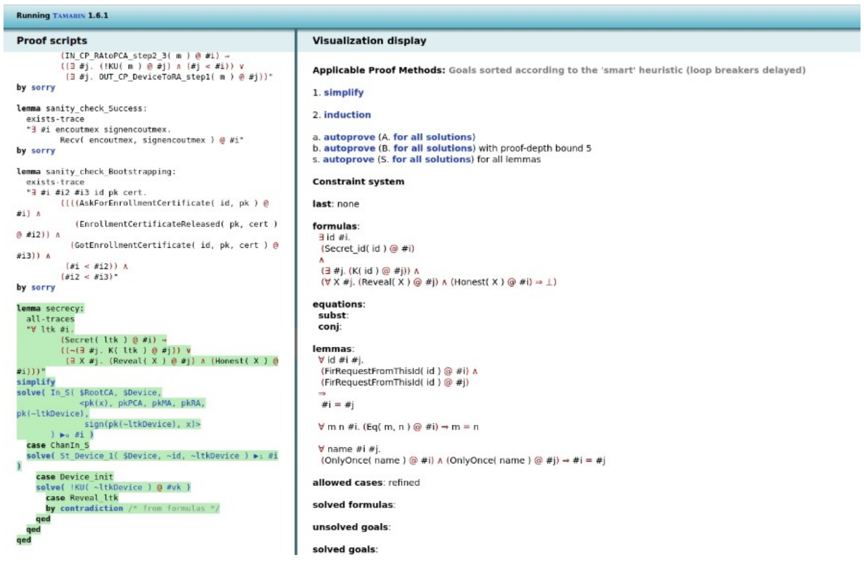
```
5     | (Ex X #j. Reveal(X)@j & Honest(X)@i)"
```

6.3 Privacy Properties

Here we come to the crucial part of security properties, those related to privacy. As mentioned earlier, the main purpose is to verify that these properties are satisfied by the protocol. A first part will therefore be devoted to the formal modelling of the privacy properties anonymity and unlinkability and then will be shown the strategy used to try to get the verification completed. Finally the results will be presented.

6.3.1 Anonymity

The definition for anonymity is highly dependent on the context of the application, and a variety of definitions are proposed in the literature. In [2] it states *Anonymity is the state*



Running Tamarin 1.6.1

Proof scripts

```
          (IN_CP_RAtoPCA_step2_3( m ) @ #i) →
          ((∃ #j. (!KU( m ) @ #j) ∧ (#j < #i)) ∨
           (∃ #j. OUT_CP_DeviceToRA_step1( m ) @ #j))"
by sorry

lemma sanity_check_Success:
  exists-trace
  "∃ #i encoutmex signencoutmex.
          Recv( encoutmex, signencoutmex ) @ #i"
by sorry

lemma sanity_check_Bootstrapping:
  exists-trace
  "∃ #i #i2 #i3 id pk cert.
          ((((AskForEnrollmentCertificate( id, pk ) @
#i) ∧
             (EnrollmentCertificateReleased( pk, cert )
@ #i2)) ∧
            (GotEnrollmentCertificate( id, pk, cert ) @
#i3)) ∧
           (#i < #i2)) ∧
          (#i2 < #i3)"
by sorry

lemma secrecy:
  all-traces
  "∀ ltk #i.
          (Secret( ltk ) @ #i) →
          ((¬(∃ #j. K( ltk ) @ #j)) ∨
           (∃ X #j. (Reveal( X ) @ #j) ∧ (Honest( X ) @
#i)))"
simplify
solve( In_S( $RootCA, $Device,
                <pk(x), pkPCA, pkMA, pkRA,
pk(~ltkDevice),
                 sign(pk(~ltkDevice), x)>
       ) ▶₀ #i )
  case ChanIn_S
  solve( St_Device_1( $Device, ~id, ~ltkDevice ) ▶₁ #i
)
    case Device_init
    solve( !KU( ~ltkDevice ) @ #vk )
      case Reveal_ltk
      by contradiction /* from formulas */
    qed
  qed
qed
```

Visualization display

Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (loop breakers delayed)

1. simplify

2. induction

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5
s. autoprove (S. for all solutions) for all lemmas

Constraint system

last: none

formulas:
```
 ∃ id #i.
  (Secret_id( id ) @ #i)
 ∧
  (∃ #j. (K( id ) @ #j)) ∧
  (∀ X #j. (Reveal( X ) @ #j) ∧ (Honest( X ) @ #i) ⇒ ⊥)
```

equations:
 subst:
 conj:

lemmas:
```
 ∀ id #i #j.
  (FirRequestFromThisId( id ) @ #i) ∧
  (FirRequestFromThisId( id ) @ #j)
 ⇒
  #i = #j

 ∀ m n #i. (Eq( m, n ) @ #i) → m = n

 ∀ name #i #j.
  (OnlyOnce( name ) @ #i) ∧ (OnlyOnce( name ) @ #j) → #i = #j
```

allowed cases: refined

solved formulas:

unsolved goals:

solved goals:

Figure 6.7. Lemma secrecy.

of being not identifiable within a set of subjects, the anonymity set. The anonymity set is the set of all possible subjects who might cause an action. In our case, the adversary cannot distinguish wether the message has been signed with a valid pseudonym or a fake one. When a pseudonym certificate is requested, a vehicle can make n requests to the RA and an adversary and other entities are unable to associate a pseudonym certificate with the real identity of the vehicle. It will be used the observational equivalence by putting `diff(~ltkDevice, ~fakeltk))` when signing the request to the RA.

```
rule CP_DeviceToRA_step1:
  let
    A = epk(~a)
    HH = epk(~hh)
    request = < A, HH, ~fk, ~fe >
    signed_request = sign(request, diff(~ltkDevice, ~
       fakeltk))
    mex = <request, signed_request, certificate>
    enc_request = aenc(mex, pkRA)
  in
  [
    !St_Device_2($Device, ~id, ~ltkDevice, pkRootCA,
       pkPCA, pkMA, pkRA, certificate),
    Fr(~a),
    Fr(~hh),
    Fr(~fk),
  Fr(~fe),
  Fr(~reqid),
  Fr(~fakeltk)
]
--[ OUT_CP_DeviceToRA_step1(mex),
    AskForPseudonymCertificate(~id, ~reqid, A, HH),
    Step1(~id, A, HH),
    OUT23(enc_request, pkRootCA)
]->
[
  Out(enc_request),
  St_Device_3($Device, ~id, ~ltkDevice, pkRootCA,
     pkPCA, pkMA, pkRA, certificate, ~reqid, ~a, ~hh,
     ~fk, ~fe)
]
```

6.3.2 Unlinkability

Always referring to [2], unlinkability of two or more items means that within a system, these items are no more and no less related, concerning the a-priori knowledge, with respect to the system of which we want to describe. This means that the probability of those items being related stays the same before (a-priori knowledge) and after the run within the system (a-posteriori knowledge of the attacker). In the protocol, the vehicle has n pseudonymous certificates that need to be changed regularly. As long as the vehicle uses the same pseudonym certificate it is always traceable. To prove this, it is sent a message signed with the pseudonymous certificate and since anyone has the public key associated with it, it can be proven that when it is verified with the real key, it will give a positive result, negative otherwise.

```
rule Device_send_message:
let
  mex = sign(~nonce, diff(SK, ~fakeltk))
in
[ !St_Device_4($Device, ~id, ~ltkDevice, certificate,
    SK, pseudocert),
  Fr(~nonce),
  Fr(~fakeltk)

]
--[ Send(~nonce, mex) ]->
[ Out(mex)  ]
```

By checking the Oberservational equivalence lemma, automatically generated by Tamarin, the protocol thus defined unfortunately fails. In the following, all strategies presented were tested for both properties separately, i.e. a specific .spthy file was created for each property and executed independently.

Heuristics

A first approach was to use heuristics. A heuristic is a technique for ranking a constraint system's open goals, and it is defined as a list of goal rankings where each of them is represented as a single character from the set "s,S,c,C,i,I,o,O".

Original protocol and Oracle

Here the test was made on the original protocol and by adding the oracle, described in section 2.4.1. Below is shown a version of the code used for the oracle. Several versions have been tested, changing both the order and the facts themselves.

```
#!/usr/bin/python

import re
import os
import sys
debug = True

lines = sys.stdin.readlines()
lemma = sys.argv[1]


rank = []              # list of list of goals, main list
    is ordered by priority
maxPrio = 110
for i in range(0,maxPrio):
  rank.append([])

if lemma != "AnyLemma":
  print("applying lemma")
  for line in lines:
    num = line.split(':')[0]

    if re.match('.*_PK\(.*', line): rank[109].append(num
       )
    elif re.match('.*_SK\(.*', line): rank[108].append(
       num)
    elif re.match('.*FirRequestFromThisId.*', line): rank[107].append(num)
    elif re.match('.*St_Device_2.*', line): rank[80].append(num)
    elif re.match('.*St_Device_3.*', line): rank[80].append(num)
    elif re.match('.*St_Device_4.*', line): rank[80].append(num)

else:

  exit(0)

# Ordering all goals by ranking (higher first)
for listGoals in reversed(rank):
  for goal in listGoals:
    sys.stderr.write(goal)
    print(goal)
```

Persistent facts

The ! symbol indicates a persistent fact, i.e. that Tamarin can consume it an indefinite number of times during the simulation. In this case, `!St_Device_2` indicates that the same vehicle (which has received an enrolment certificate) can request several pseudonym certificates. By removing ! we restrict the same vehicle from requesting more than one pseudonymous certificate, thus greatly simplifying the trace that Tamarin generates to find a solution.

```
rule Device_bootstrap:
  let certificate = <pkdev, signature>
  in
[ In_S($RootCA, $Device, <pkRootCA, pkPCA, pkMA, pkRA, certificate>),
  St_Device_1($Device, ~id, ~ltkDevice)
]
--[ //check  pkdev == pk(~ltkDevice) && signature corretta usando pkRootCA
    Eq(pkdev, pk(~ltkDevice)),
    Eq(verify(signature, pkdev, pkRootCA), true),
    GotEnrollmentCertificate(~id, pk(~ltkDevice), certificate)
  ]->
```

```
[ !St_Device_2($Device, ~id, ~ltkDevice, pkRootCA, pkPCA
    , pkMA, pkRA, certificate) ]
```

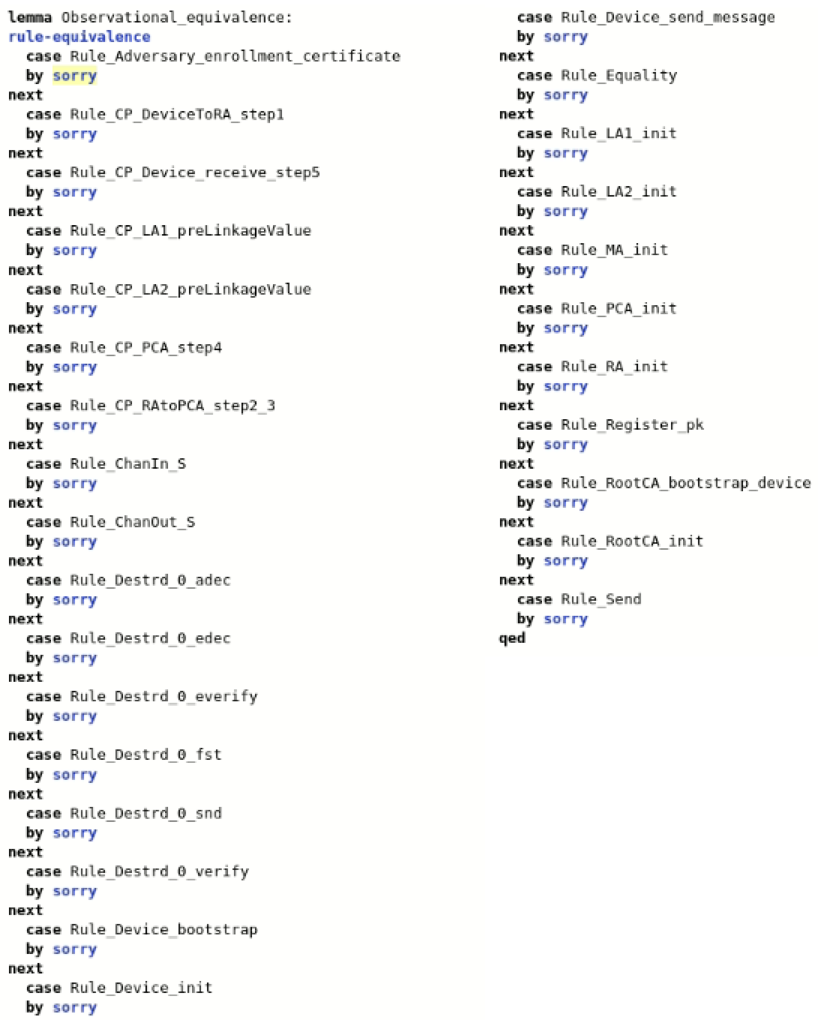
Persistent facts and Oracle

The previous point was performed without an oracle. Then the test was repeated using the different oracle versions presented above.

Observational equivalence: rule-equivalence

The last test performed was on the rule-equivalence mode of the Obervational equivalence lemma. In this way, Tamarin decomposes the lemma into sub-cases as can be seen in the figure 6.8. Each sub-case is decomposed in turn into two further sub-cases, which are LHS and RHS, i.e. if both are tested, it does not matter what the opponent does, he will always see equivalent executions in all cases.

Unfortunately, none of these attempts resulted in the termination of the verification for the two properties. In all of the above cases, the protocol execution saturated the available ram memory, which in our case was 128GB. In general, the average execution time required to saturate all resources was 2-3 days, whereas with the use of an oracle this time decreased to about 1 day. Although the use of the oracle brought benefits in terms of speed, it was still not enough to complete the audit.



```
lemma Observational_equivalence:
rule-equivalence
  case Rule_Adversary_enrollment_certificate
  by sorry
next
  case Rule_CP_DeviceToRA_step1
  by sorry
next
  case Rule_CP_Device_receive_step5
  by sorry
next
  case Rule_CP_LA1_preLinkageValue
  by sorry
next
  case Rule_CP_LA2_preLinkageValue
  by sorry
next
  case Rule_CP_PCA_step4
  by sorry
next
  case Rule_CP_RAtoPCA_step2_3
  by sorry
next
  case Rule_ChanIn_S
  by sorry
next
  case Rule_ChanOut_S
  by sorry
next
  case Rule_Destrd_0_adec
  by sorry
next
  case Rule_Destrd_0_edec
  by sorry
next
  case Rule_Destrd_0_everify
  by sorry
next
  case Rule_Destrd_0_fst
  by sorry
next
  case Rule_Destrd_0_snd
  by sorry
next
  case Rule_Destrd_0_verify
  by sorry
next
  case Rule_Device_bootstrap
  by sorry
next
  case Rule_Device_init
  by sorry
next
  case Rule_Device_send_message
  by sorry
next
  case Rule_Equality
  by sorry
next
  case Rule_LA1_init
  by sorry
next
  case Rule_LA2_init
  by sorry
next
  case Rule_MA_init
  by sorry
next
  case Rule_PCA_init
  by sorry
next
  case Rule_RA_init
  by sorry
next
  case Rule_Register_pk
  by sorry
next
  case Rule_RootCA_bootstrap_device
  by sorry
next
  case Rule_RootCA_init
  by sorry
next
  case Rule_Send
  by sorry
qed
```

Figure 6.8. Observational equivalence: rule-equivalence.

Chapter 7

Conclusion

This chapter concludes the thesis work. This thesis explained the Security Credential Management System protocol, which represents the state of the art in V2X communications. Its main feature and strong point is this new cryptographic mechanism, the *Butterfly Key Expansion*, which allows for the efficient and secure management of pseudonymous certificates and, of course, the distribution of secrets between the various entities in such a way that users' privacy is unlikely to be compromised.

In this work, we were able to formally model and verify security properties such as authentication and secrecy of exchanged messages. Unfortunately, although the privacy properties, specifically Anonymity and Unlinkability, were modelled and formally defined, it was not possible to verify them given the complexity of the protocol and the scarce computational resources.

For future work, this protocol could be analysed again using a more powerful machine in terms of computational resources and ram size. On the one hand, this would improve the execution speed and on the other hand, it could help in getting the verification completed without further modifications. If this were not sufficient, then different formal verification tools could be used. One example would be to translate the model and all formalised properties into the ProVerif tool.

Bibliography

[1] Benedikt Brecht , Dean Therriault, André Weimerskirch, William Whyte, Virendra Kumar, Thorsten Hehn, and Roy Goudy. «A Security Credential Management System for V2X Communications». In: *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS* VOL.19.NO.12 (December 2018) (cit. on p. 31).

[2] Andreas Pfitzmann and Marit Köhntopp. «Anonymity, unobservability, and pseudonymity - A proposal for terminology». In: *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA,USA* (July 25-26, 2000). DOI: 10.1007/3-540-44702-4_1 (cit. on pp. 70, 72).

[3] Simon Parkinson, Paul Ward, Kyle Wilson, and Jonathan Miller. «Cyber Threats Facing Autonomous and Connected Vehicles: Future Challenges». In: *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS* VOL.18.NO.11 (November 2017).

[4] Jorden Whitefield, Liqun Chen, Frank Kargl, Andrew Paverd, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. «Formal Analysis of V2X Revocation Protocols». In: ().

[5] Katharina Hofer-Schmitz, Branka Stojanovi ć. «Towards formal verification of IoT protocols: A Review». In: *DIGITAL –Institute for Information and Communication Technologies, JOANNEUM RESEARCH Forschungsgesellschaft mbH, 17 Steyrergasse, Graz, Austria* ().

[6] Jonathan Petit, Florian Schaub, Michael Feiri, and Frank Kargl. «Pseudonym Schemes in Vehicular Networks: A Survey». In: *IEEE COMMUNICATION SURVEYS TUTORIALS* VOL.17.NO.1 (2015).

[7] Monowar Hasan, Sibin Mohan, Takayuki Shimizu, and Hongsheng Lu. «Securing Vehicle-to-Everything (V2X) Communication Platforms». In: *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS* VOL.5.NO.4 (December 2020).

[8] Zachary MacHardy, Ashiq Khan, Kazuaki Obana, Member, IEEE, and Shigeru Iwashina. «V2X Access Technologies: Regulation, Research, and Remaining Challenges». In: *IEEE COMMUNICATIONS SURVEYS TUTORIALS* VOL.20.NO.3 (2018).

[9] Singam Bhargav Ram, Vanga Odelu. «Security Analysis of a Key Exchange Protocol under Dolev-Yao Threat Model Using Tamarin Prover». In: *IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)* (2022). DOI: `10.1109/CCWC54503.2022.9720852`.

[10] Jannik Dreier Simon Meier Ralf Sasse Benedikt Schmidt David Basin Cas Cremers. *Tamarin-Prover Manual*. URL: `https://tamarin-prover.github.io/manual/book/001_introduction.html`.

[11] Riccardo Sisto. *Security Verification and Testing Slides*. Materiale didattico. Politecnico di Torino. 2021-2022.

[12] David Basin. *Formal Methods for Security Knowledge Area Version 1.0.0*. CyBOK, 2021.