Politecnico di Torino

Master's Degree in Cybersecurity

Academic Year 2024/2025

Graduation Session October 2025

Formal Security Verification of a Standard Protocol for V2X Communications

Supervisors:
Prof. Riccardo Sisto
Dr. Simone Bussa
Prof. Fulvio Valenza

Candidate:
Angelo Barbera

Abstract

Vehicle-to-Everything communications are technologies that enable message exchange between vehicles and other vehicles or infrastructure components. These technologies are employed for the *Intelligent Transportation Systems* which enable the implementation of safety-related applications such as electronic emergency braking light, cooperative collision avoidance or traffic light announcements. The messages exchanged between vehicles and other vehicles or infrastructure elements, called *Basic Safety Messages*, contain sensitive information such as time, position, and motion data. It is critical to protect this information to prevent users tracking and attacks that could harm users' safety.

To address the security issues, *Crash Avoidance Metrics Partners LLC* developed the *Security Credential Management System* (SCMS) to support the establishment of a Public Key Infrastructure for V2X security, which issues certificates to system participants. The solution adopted to protect the users privacy is the use of *Pseudonym Certificates*, which provide authentication while protecting user identity and making tracking more difficult.

The objective of this thesis is to use formal verification methods to analyse the protocol and verify its security properties. The formal analysis was performed using the Tamarin prover. Each protocol step was formalized and sanity checks were executed to verify the execution of the protocol. The work then focused on the formal verification of security properties such as secrecy and authentication.

Acknowledgements

I would like to thank my supervisors, Prof. Riccardo Sisto and Prof. Fulvio Valenza, for the opportunity to work on this thesis, and Dr. Simone Bussa for his assistance during its development.

I am grateful to my family, especially my parents, for their support and for making it possible to complete my studies in Turin.

I would also like to thank *Giorgia*, *Alessandro*, *Paola*, *Davide*, *Alessandra*, *Alex*, *Dalila*, *Davide*, and *Simone* for their friendship and the wonderful times spent together.

Table of Contents

List of Figures

Acronyms

BSM
: Broadcast Safety Message

CA
: Certification Authority

CRL
: Certificate Revocation List

CRLG
: CRL Generator

CS
: Certification Services

CSR
: Certificate Signing Request

DCM
: Device Configuration Manager

ECA
: Enrollment Certification Authority

EE
: End Entity

GCCF
: Global Certificate Chain File

GPF
: Global Policy File

HSM
: Hardware Security Module

ICA
: Intermediate Certification Authority

ITS
: Intelligent Transportation Systems

LA
: Linkage Authority

LCI
: Linkage Chain Identifier

LOP
: Location Obscurer Proxy

MA
: Misbehavior Authority

OBE
: On-Board Equipment

PCA
: Pseudonym CA

PG
: Policy Generator

PKI
: Public key infrastructure

PP
: Pseudonym Provider

RA
: Registration Authority

RCA
: Root Certificate Authority

RSE
: Roadside Equipment

RSU
: Roadside Unit

SCMS
: Security Credential Management System

TLS
: Transport Layer Security

V2I
: Vehicle-to-Infrastructure

V2V
: Vehicle-to-Vehicle

V2X
: Vehicle-to-Everything

Chapter 1

Introduction

This thesis explains the basics of Vehicle-to-Everything communications and their security issues. It then introduces the asymmetric-cryptography scheme used to achieve the pseudonymity security property, which is relevant in V2X communications.

It then outlines the importance of formal verification methods and introduces the Tamarin prover, the tool used to prove the security properties of the Security Credential Management System protocol.

The SCMS protocol steps and their Tamarin models are presented in detail, and the system's security properties are formally analysed using the Tamarin prover.

The goal of this thesis is to build a Tamarin model for each SCMS protocol step, run preliminary checks to ensure the model's correctness and the protocol's executability, and finally model and verify the main security properties the protocol should satisfy, such as secrecy, authentication, and pseudonymity.

1.1 Thesis Structure

The structure of the thesis is the following:

- **Chapter 1 - Introduction**: This chapter describes the scope and the goal of the thesis, and defines its structure.
- **Chapter 2 - Vehicle-To-Everything Communications**: This chapter describes the Vehicle-To-Everything Communications, the related Security Issues, and the Asymmetric Cryptography Scheme of the Security Credential Management System.
- **Chapter 3 - Formal Methods**: This chapter introduces the Formal Verification, the Security Protocol Verification, and outlines the main characteristics of Tamarin.

- **Chapter 4 - Threat Modeling**: This chapter describes the threat modeling process and the Microsoft Threat Modeling Tool.
- **Chapter 5 - Security Credential Management System**: This chapter defines the SCMS Structure, the SCMS Threat Model, the Certificates types and requirements needed to the system, the Butterfly Key Expansion, the Linkage Values, and the SCMS Steps.
- **Chapter 6 - Security Credential Management System Tamarin Model**: This chapter introduces the Tamarin Model of the SCMS protocol.
- **Chapter 7 - Security Properties Analysis**: This chapter introduces the SCMS Threat Modeling, the Tamarin Sanity Checks, and the verification of the Security Properties.
- **Chapter 8 - Conclusion**: This chapter summarizes the results and propose possible improvements.

Chapter 2

Vehicle-to-Everything Communications

2.1 Vehicle-to-Everything Model

Vehicle-to-Everything (V2X) communications are technologies that enable data exchange between vehicles and everything else. It includes *Vehicle-to-Infrastructure* (V2I) and *Vehicle-to-Vehicle* (V2V) communications, which covers communications between vehicles and other vehicles or infrastructure components such as *Roadside Units* (RSU) [1].

These technologies are necessary to the development of *Intelligent Transportation Systems* (ITS), especially for safety-related applications including local danger warning, electronic emergency braking light and cooperative collision avoidance, traffic light announcements, as well as service announcements and provisioning such as internet access. These applications require *On-Board Equipments* (OBEs) in the vehicles communicating with RSUs and others OBEs.

V2V communications are based on *Broadcast Safety Messages* (BSMs), which include sender's time, position, speed and other useful information. Each BSM is digitally signed by the sender, the receiver verifies the signature then evaluates the message and decides whether to display a warning to the driver.

V2I communications need authentication for broadcast messages sent by RSUs and the establishment of a communication channel between OBE in the vehicle and RSU [2]. A general scheme of V2X communication is shown in Fig. 2.1

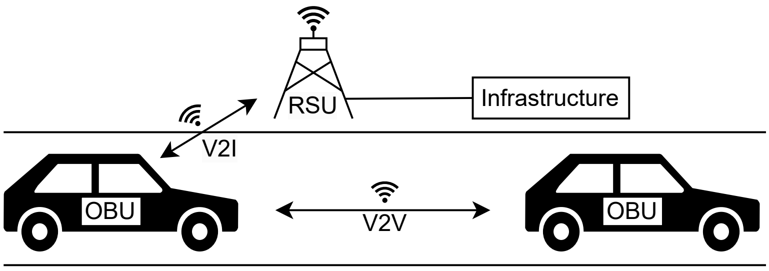


Figure 2.1: Vehicle-to-Everything model

2.2 Security Issues

Since V2X communications are also related to safety applications, it is important to implement appropriate security measures. Other than protecting against attacks which aim to compromise the integrity and the availability of the system, it is also needed to address privacy issues.

The BSMs exchanged contain sensitive information (e.g., position, speed, vehicle path) which can also be used to track the users of the vehicles. A solution to this problem is to use pseudonyms, which can be used for authentication but do not contain information that could reveal the real identity. A user can use different pseudonyms to avoid linking between actions performed using the same pseudonym. Moreover, the possibility to obtain the real identity of a pseudonym should be reserved only to authorized authorities for specific reasons. To improve the privacy of the users it is possible to design the system in order to require the cooperation between multiple parties to resolve a pseudonym to the real identity.

2.3 Asymmetric Cryptography Scheme

There are different schemes to achieve pseudonymity, they differ in the cryptography mechanism used. This section presents the asymmetric cryptography scheme, on which the *Security Credential Management System* (SCMS) is based, which will be presented later.
The vehicles can receive public key certificates and key pairs. The certificates are used as pseudonyms, they do not contain identifying information. The secret keys are used to sign the sent messages, the signature will be verified by the receiver using the corresponding public key. The phases of this scheme are the following:

- **Pseudonym Issuance**: there is a hierarchy of *Certification Authorities* (CAs), the CAs issue long-term identity certificates, while pseudonyms are issued by

Pseudonym Providers (PPs). Each pseudonym is valid for a limited period of time.

- **Pseudonym Use**: Pseudonyms are used to sign the sent messages, the public and private keys are managed by a *Hardware Security Module* (HSM).
- **Pseudonym Change**: When a pseudonym expires the OBE loads a new pseudonym from its store or requests a new one from the PP.
- **Pseudonym Revocation**: A pseudonym can be revoked, as well as the long-term identity, preventing the acquisition of new pseudonyms.

Chapter 3

Formal Methods

3.1 Formal Verification

In the traditional development cycle the correctness is verified after the implementation, and the proper patches are then applied. This approach can lead to security issues because not all the bugs can be detected during the test phase.

Formal verification is a method used to prove the correctness of a model based on its specifications using a formal model of the system. This method provides the assurance that the system satisfies the specifications in a rigorous way. One of the challenges of formal verification is that a problem may be undecidable (it is not possible to create an algorithm that can always solve the problem correctly), and even if a problem is decidable it can be too complex to solve in a reasonable amount of time or resources. Hence the tool may use approximation producing false negatives or false positives and may require human interaction. Moreover, the properties are verified against the model, not the real system, so the results of the verification depends on the quality of the model. Lastly, formal verification tools require an effort that it is not always feasible with limited resources. [3]

The mathematical abstraction of the system and its properties are specified using a language that depends on what is needed to model, mainly a logical language is used. The two possible approaches for verification are: model checking and theorem proving.

A model checker takes as input a model and the property to verify. It returns true if the property holds or it returns false and a counterexample if the model does not satisfy the property.

A theorem prover takes as input a formal system and a property, it may also works interactively with human assistance. It returns true and a proof if the property can be proved, otherwise if the proof is not found it does not provide any information.

Both methods allow to verify if a property is true, the model checker provides a proof of non-validity using a counterexample, while the theorem prover provides a proof of validity. The big difference is that the theorem prover does not provide any information if it is not able to find a proof.

3.2 Security Protocol Verification

Security protocols is a field of application of the formal methods, typically the aim is proving security properties such as authentication, integrity, confidentiality or other security properties that the protocol must satisfy. There are two main approaches to model a security protocol: symbolic modeling and computational modeling.

In a symbolic model the data are abstract data types, the cryptographic operations are modelled using algebraic operators with ideal properties. The attacker can read, delete, modify, create messages and execute cryptographic operations, but it cannot guess secrets. The goal is to prove that attacks are impossible with these assumptions.

In a computational model the data are modelled as bitstrings, the cryptographic algorithms are represented through algorithms, the cryptosystem flaws are modelled, and the protocol runs are probabilistic. The attacker is any polynomial-time algorithm. The goal is to prove that does not exist an attacker which is able to reach a specific objective in a polynomial time with a non-negligible probability.

Computational models are more detailed than the symbolic models, however it is needed more effort to formalise the protocol and to prove its properties.

3.3 Tamarin

The Tamarin prover is a tool for symbolic modeling and analysis of security protocols. It takes as input a model of a system and the specifications of the security properties. Tamarin is a constraint solver that finds system behaviours that are consistent with the system model and the negation of the property, these behaviours represent attacks. Tamarin may not always terminate, there are three different possibilities:

- Tamarin terminates and finds an attack on the protocol.
- Tamarin produces a proof demonstrating that the protocol is secure and that there are no possible attacks.
- Tamarin fails to terminate (e.g., it may run out of memory or require too much time) and and no information about the presence of attacks is obtained.

Tamarin allows user to inspect proofs and interact with the prover during proof construction, which can enable termination or reduce the time required to reach a result. [4] The Tamarin workflow is shown in Fig. 3.1.

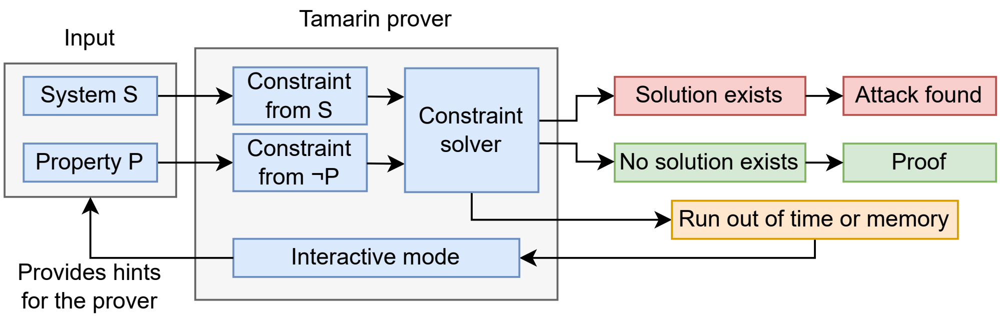


Figure 3.1: Tamarin scheme

Tamarin supports two modeling languages to model the protocol, the adversary, and the properties.

The first one is based on *multiset rewriting rules*, it is Turing-complete, and allows to define a transition system. The state of a transition system is a multiset that represents the states of each protocol participant, the adversary and the communication channels. The transitions model the state evolution. Moreover Tamarin supports equations to specify cryptographic operators.

The second one is used to specify the properties of the protocol, it is based on first-order logic and supports quantification over time. It is used to verify whether a security property holds during the protocol execution.

Tamarin also supports another language for modeling through the SAPIC (Stateful Applied PI-Calculus) module. A protocol can be modelled using rules or as a process. The process is converted to rules according to the semantic of the process calculus.

3.3.1 Modeling with multiset rewriting rules

Multiset rewriting rules are used to specify the transitions of the systems. As an example, the following rule is used to model the registration of a public key in a Public Key Infrastructure (PKI):

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

All rules have the form:

```
rule rule_name: [ L ] --[ A ]-> [ R ]
```

where L (left-hand side), A (action facts), and R (right-hand side) are multiset of facts. It means that if the facts in L exist in the current state of the system, we can make a transition replacing these facts with the facts in R. The facts in A are used to specify the rules that express the properties of the protocol. When the transition system's initial state is empty only the rules with an empty set as L apply. A fact with the '`!`' prefix is a *persistent fact*, meaning that it is never consumed when a rule is applied. All the other facts are *linear facts*, meaning that the fact are removed from the state when a transition occours. It is required that all variables in A and R also occour in L, the only exception is for public variables.

Tamarin includes a special rule used to generate the `Fr` facts:

```
rule Fresh: [ ] --[ ]-> [ Fr(~x) ]
```

where the type of `~x` is fresh, meaning that the value is random, unique and unpredictable.

3.3.2 Terms

The *rules* are formed by *terms*, there are several basic terms:

- **`$x`** represents a public variable that can be instantiated with any public constant, which is known by the adversary.
- **`~x`** represents a fresh variable that can be instantiated with any fresh value, representing randomly generated values.
- **`'x'`** represents a public constant.
- **`x`** represents a variable without type declaration.

3.3.3 Functions and Equations

Rules can also contain *functions* used to model algorithms or known functions. They have a name, an arity (the number of arguments) specified in the function declaration. The adversary can use all functions to build new terms except for functions annotated with `[private]` after their declaration. The properties of functions can be specified using *equations*, as shown in the following example:

```
functions: senc/2, sdec/2

equations: sdec(senc(m, k), k) = m
```

A set of equations (with a set of terms) defines an *equational theory*, which specifies when two terms can be considered equal. The `=` sign represents equality with respect to the equational theory.

Without specifying an equational theory two terms are equal only if they are syntactically identical. Tamarin has default built-in models for cryptographic primitives such as `hashing` defines a hash function, `asymmetric-encryption` and `symmetric-encryption` defines asymmetric and symmetric encryption respectively, `diffie-hellman` defines Diffie-Hellman exponentiation.

3.3.4 Rules and Lemmas

A finite sequence of action facts represents a *trace*, a *trace property* represents a specific protocol behaviour based on sequence of traces, and a trace property is specified using a *lemma*. A lemmma is based on first-order logic formulas, the occourrences of actions can be associated to a timepoint to express time constraints. As an example, considering the following rules and lemma

```
rule send:
    [ Fr(~x) ]
  --[ Sent(~x) ]->
    [ Out(~x) ]

rule receive:
    [ In(~x) ]
  --[ Received(~x) ]->
    [ ]

lemma sent_before_received:
    "All x #i. Received(x) @#i ==> Ex #j. Sent(x) @#j & #j < #i"
```

the rule `send` and `receive` model the sending and the receiving of a message, while the lemma `sent_before_received` states that the action fact `sent(x)` must occour before `received(x)`. **`All`** is the universal qunatifier, **`Ex`** is the existential qunatifier, **`==>`** is the implication, **`&`** is the conjunction, **`#t`** represents a timepoint and **`event @#t`** specifies that **`event`** happened at timepoint **`#t`**.

All lemmas are considered to be *all-traces*, meaning that the formula must hold for all the protocol traces. It is possible to use the keyword *exists-trace*, after the rule name and before the formula, to specify that the formula must hold for at least one trace.

3.3.5 Restrictions

Tamarin provides the *restriction* keyword to specify first-order logic formulas to limt the set of traces to consider. Restrictions are trace properties, but unlike lemmas they are not proved. The following is an example of restriction to model the equality.

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

This restriction ensures that in any rule with `Eq(x, y)` action fact, `x` and `y` are equal.

3.3.6 Observational Equivalence

Tamarin also supports the definition of a property, called *observational equivalence*, over a set of traces. The prover verifies whether the attacker is able to distinguish the behaviour of the two systems originating from the same input theory. These two systems differ for the terms inside the `diff(x, y)` operator.

Tamarin explores all possible executions by replacing the operator with `x` or `y` and checks whether each trace of one system can be mirrored int the other one. If Tamarin cannot to find a mirror, this indicates either the presence of an attack or that the tool failed to find a mirror. In observational equivalence mode Tamarin uses an approximation of the observational equivalence, hence it may fail to prove that the two systems are equivalent.

3.3.7 Partial Deconstruction

Tamarin searches for the *sources* of each protocol and `KU` fact. A source is a partial execution that generates a fact. To generate these sources, Tamarin builds a constraint system containing a constraint that requires the presence of these facts and runs a constraint solver. If a chain constraint cannot be resolved, it remains unsolved, generating a *partial deconstruction*. A partial deconstruction may cause non-termination. There are two approaches to solve partial deconstruction: *source lemmas* and *auto-sources*.

A source lemma is marked by the annotation `[sources]` placed after its name. This type of lemma restricts the sources of a fact, preventing partial deconstructions.

Tamarin can be executed with the `--auto-sources` flag, in this case the prover tries to automatically generate source lemmas. However this process may generate source lemmas that cannot be proved, or it may fail to solve all partial deconstructions.

Chapter 4

Threat Modeling

Threat modeling is a structured process for collecting and analysing of all information needed to evaluate the security of a system. The result of this process is the threat model, a list of the potential threats, and the countermeasures to apply to mitigate or prevent the threats. [5]

The main threat modeling steps are:

1. **Assessment Scope**: Analyse the target system of the threat modeling process and build the threat model.
2. **Threat Identification**: Analyse the model to identify the potential threats.
3. **Countermeasures Identification**: Identify the countermeasures to mitigate or avoid the identified threats.
4. **Final Assessment**: Verify the consistency of the threat modeling process.

There are several tools that support threat modeling, such as *OWASP Threat Dragon* or *Microsoft Threat Modeling Tool*.

4.1 Microsoft Threat Modeling Tool

Microsoft Threat Modeling Tool is a software that allows to build a threat model of the analysed system and to automatically identifies potential threats [6]. The threat model is constructed from basic elements, each element can be customized by specifying specific properties to make the analysis more accurate:

- **Process**: An element that contains specific logic or performs computations.
- **External Interactor**: An actor outside the system boundary that interacts with it.

- **Data Store**: A persistent storage for data.
- **Data Flow**: Movement of data between processes, data stores, and external interactors.
- **Trust Boundary**: A line that separates areas with different trust levels.

The threat analysis is based on the *STRIDE* model:

- **Spoofing**: Pretending to be another entity.
- **Tampering**: Unauthorized modification of data at rest or messages in transit.
- **Repudiation**: Denying having performed an action.
- **Information Disclosure**: Information exposed to unauthorized parties.
- **Denial of Service**: Making a system or a service unavailable.
- **Elevation of Privilege**: Gaining higher privilege than intended.

For each threat, the tool returns the STRIDE category, a description, and a priority (High, Medium, Low).

Chapter 5

Security Credential Management System

The Security Credential Management System (SCMS) [2] implements a PKI with specific characteristics to support the V2X features. The most important aspects are its size, since it has to support about 300 billion certificates per year, and the balance between security and privacy.

The SCMS supports several V2I applications category, such as infrastructure originating broadcast messages and service announcement and provisioning.

5.1 SCMS Structure

A SCMS component is *intrinsically-central* if it has exactly one distinct instance overall, and *central* if it has exactly one distinct instance in the specific instantiation of the system. Different instances of components have different identifiers and cryptographic materials. The components that are not central can have multiple instances. The only intrinsically-central components are the Misbehavior Authority (MA), the Policy Generator (PG) and the SCMS Manager.

The overview of the SCMS architecture is shown in Fig. 5.1. The solid lines represent the communications between the components, the dashed lines represent the chain of trust, while the dotted lines indicate secure out-of-band communication.

The connections with the Location Obscurer Proxy (LOP) are anonymized, with all location information removed from messages.

The components marked *V/I* provide separate V2V and V2I functionality, while the components marked *X* provide general functionality for the V2X system. All the components communicate using protected and reliable communication channel such as Transport Layer Security (TLS).

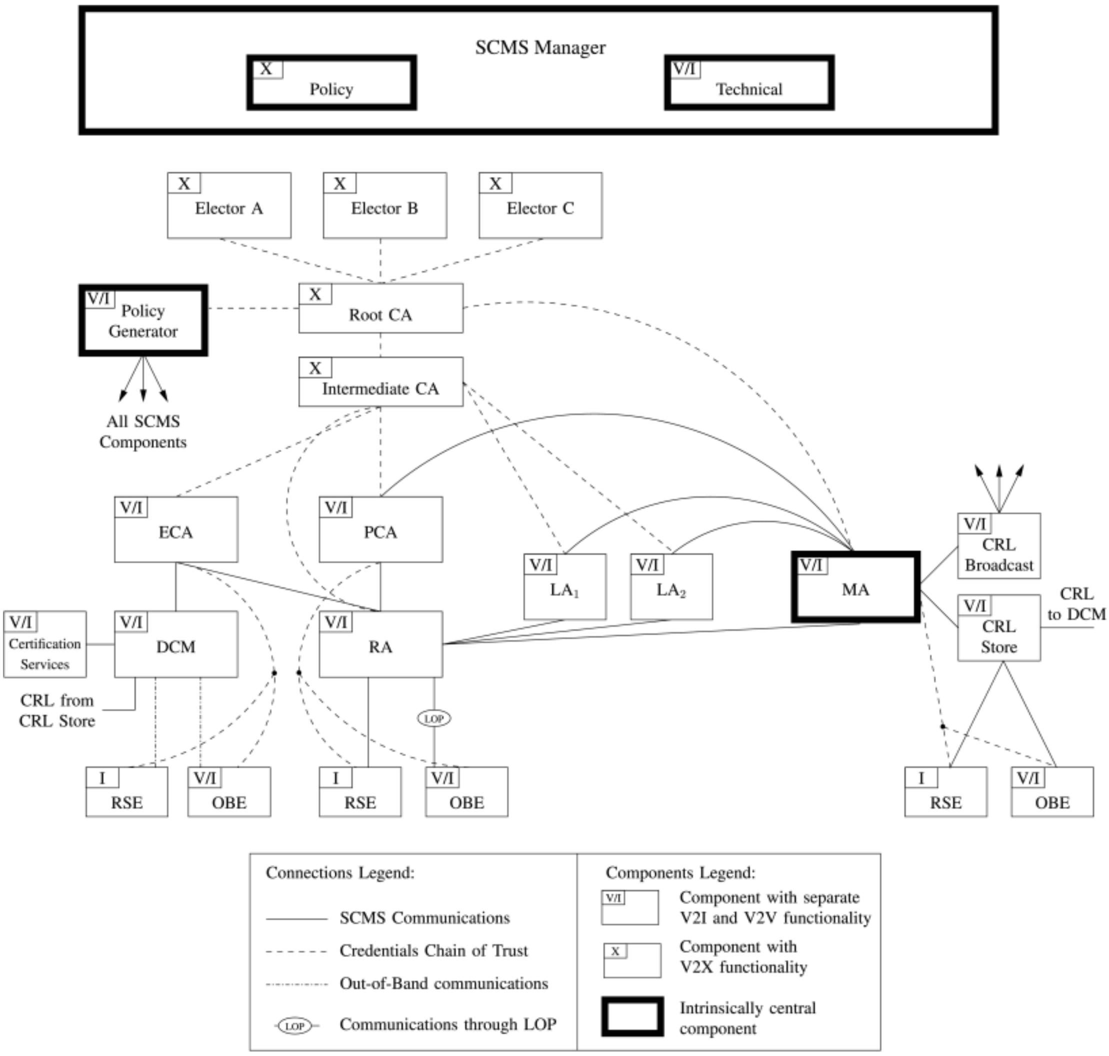


Figure 5.1: SCMS architecture scheme

The components that constitute the SCMS are:

- **SCMS manager**: Ensures the SCMS runs efficiently and fairly by setting policies and procedures for handling misbehavior reports and revocation requests in an accurate way.
- **Certification Services (CS)**: Defines the certification process and identifies the types of devices eligible to receive digital certificates.
- **CRL store**: Stores and distributes CRLs.
- **CRL broadcast**: Broadcasts the CRLs.

- **Device**: An End Entity (EE) that sends or receives messages (e.g., a RSU or a OBE).
- **Device Configuration Manager (DCM)**: Confirms to the ECA that a device qualifies for enrollment certificate, and provides the necessary configuration settings and certificates during bootstrapping.
- **Enrollment Certification Authority (ECA)**: Issues enrollment certificates.
- **Intermediate Certification Authority (ICA)**: Acts as a secondary CA to protect the RCA from traffic and attacks. The RCA issues the ICA's certificate.
- **Linkage Authority (LA)**: Generates *pre-linkage* values, which are used to support certificate revocation.
- **Location Obscurer Proxy (LOP)**: Changes source addresses to hide the requesting device's location, preventing network addresses from being tied to physical locations.
- **Misbehavior Authority (MA)**: Processes reports of device misbehavior, flags devices that are malfunctioning or acting maliciously, and adds them to the CRL to revoke their access. Additionally, it starts the procedure to associate a certificate identifier with the matching enrollment certificates and record them in the RA's internal blacklist. The MA contains the Global Misbehavior Detection used to identify the misbehaving devices and the CRL Generator (CRLG) which generates and distributes the CRL.
- **Registration Authority (RA)**: Validates and processes device requests, creating individual certificate requests to the PCA from each validated device request. Additionally, the RA supplies devices with authenticated information about SCMS configuration changes.
- **Pseudonym CA (PCA)**: Issues pseudonym, identification, and application certificates.
- **Policy Generator (PG)**: Manages the Global Policy File (GPF) containing global configuration information, and the Global Certificate Chain File (GCCF) containing the trust chains of the SCMS.
- **Root Certificate Authority (RCA)** : It is the root of the certificate chain. It issues the certificates for the ICA and SCMS components. It holds a self-signed certificate, and a ballot in which a quorum of electors votes establishes trust in the RCA.

- **Electors**: Sign ballots to endorse or revoke a RCA or an elector. Electors have self-signed certificates, all the entities of the system trust the initial set of electors.

5.2 SCMS Threat Model

The threat model is based on an unpublished risk assessment, considering only V2V applications for driver notification, not for control or traffic management. A V2V safety application is designed to notify the driver in case of danger but it is not able to control the vehicle.

The main risk is that a users disable the safety system because of too many false warning originating from malicious users. If too many users disable the system, it will fail because it is depends on the collaborative participation of most vehicles.

Moreover, the are privacy risks resulting from insider and outsider attacks on the SCMS.

In conclusion, the SCMS must be protected against attacks on users' privacy and against misbehaving messages that cause false warnings.

The SCMS is designed assuming the actual cryptographic algorithms are secure, but at the same tame it is modular and flexible, allowing the cryptographic algorithms to be upgraded if they are broken by quantum computers.

Privacy by Design and *Misbehavior Detection and Revocation* are the two elements used to protect the SCMS from the risks mentioned above:

- **Privacy by Design**: The system must be designed to make it difficult to track the vehicle (and thus its owner) using the transmitted data. For this purpose the applications using unicast and multicast messages should employ encryption and identity protection mechanisms, and these applications should be opt-in, giving participants the freedom to choose whether to use the service after evaluating privacy risks and benefits. Applications that use broadcast messages must prevent an attacker from determining whether BSMs transmitted in two different locations originate from the same device. To protect against insider attackers (who have access to BSMs and other information), the SCMS is divided into components managed by different organizations, requiring that at least two components must cooperate to gain enough information to track a device. To protect against outsider attackers (who have access only to BSMs), the EE devices use a large number of certificates that are rotated frequently (e.g., every 5 minutes).

- **Misbehavior Detection and Revocation**: A CRL is periodically distributed. The devices use the CRL to reject messages originating from revoked devices. Moreover, the SCMS stores a list of revoked devices to deny any certificate requests from them. Misbehavior detection is the process used to identity misbehaving devices. It includes *local misbehavior detection* performed by vehicles to detect malfunctions and send misbehavior reports to the SCMS, and a *global misbehavior detection* to analyse the misbehavior reports and decide whether to revoke a device.

5.3 Certificates

Given the security risks of the SCMS, certificates must satisfy the following requirements:

- **Privacy, size and connectivity**: Certificates should have a limited validity period to protect user privacy, but devices cannot store a large number of certificates due to memory and cost limitations. Moreover, vehicles cannot continuously connect to the SCMS to obtain new certificates.
- **CRL size**: Instead of including all revoked certificates in a CRL, a mechanism is used to revoke certificates without revealing those used by the device prior to revocation.
- **Certificate waste and Sybil attack**: Periodically changing certificates for privacy reasons results in a large number of unused certificates. Another solution is to use certificates that remain valid simultaneously for a long period of time, however this enables spoofing attacks such as the Sybil attack if a single device is compromised.

The proposed model uses certificates with a validity period of 1 week and a minimum of 20 certificates valid simultaneously, these parameters balance privacy and storage requirements. Moreover, devices do not need to request new certificates, since the SCMS will continue generating them until a device stops collecting them for an extended period of time. The devices can download the certificates in batches provided by the RA.

The SCMS supports different application types, hence, it is designed to work with different certificate types:

- **RSE enrollment certificates**: Provided during the bootstrap phase, they are used to request signing and encryption certificates for RSEs and provide *pseudonymity*.
- **RSE application certificates**: Used by RSEs to sign broadcast and service announcement messages, and to provide an encryption key to OBEs to send encrypted data.

- **OBE enrollment certificates**: Provided during the bootstrap phase, they are used to request signing and encryption certificates for OBEs.
- **OBE pseudonym certificates**: They provide *pseudonymity*, *unlinkability* and efficient revocation using *shuffling*, *linkage values*, *butterfly key expansion* and encryption of certificates by the PCA to OBE. They are used to sign the BSMs broadcasted by OBEs and for authorization purposes.
- **OBE identification certificates**: Used for OBE identification, they do not provide pseudonymity or unlinkability. They may employ *shuffling*, encryption by the PCA to OBEs and *butterfly key expansion*.

The properties a certificate may satisfy, depending on its type, are:

- **Unlinkability**: A device using different certificates, and therefore different identities, cannot be traced.
- **Pseudonymity**: The certificate does not contain real-word identifiers.
- **Butterfly Key Expansion**: A mechanism for efficiently generating cryptographic keys for certificates.
- **Shuffling**: Performed by the RA, when combined with butterfly key expansion does not allow the PCA to identify the certificates assigned to a specific device.
- **Linkage values**: They are values contained in the certificates allowing efficient revocation.

5.4 Butterfly Key Expansion

The standard process for requesting a certificate requires generating a key pair on the device and sending the Certificate Signing Request (CSR) to the PKI. Then, the CA signs the certificate and returns it to the requester. This approach requires generating a large number of cryptographic key pairs on the device.

Butterfly Keys allow an OBE to request certificates that use different signing keys, and to have those certificates encrypted under different encryption keys, all with a single request that contains a signing public key seed, an encryption public key seed and two expansion functions. The PCA encrypts the certificates for delivery to the OBE, preventing the RA from associating the certificate contents with a specific OBE. A scheme of butterfly key expansion is shown in Fig. 5.2.

The butterfly key expansion is based on elliptic cryptography. Given P and $A = aP$, it is hard to compute a. The parties agree on a base point G of order l. The *caterpillar keypair* is an integer a and a point $A = aG$. The value A and the signing expansion function $f_k(l)$ are provided to the RA by the certificate requester. The expansion function is a pseudo-permutation on the integers mod l, it is used to generate points on the NIST curve NISTp256, and it is defined as follows:

$$f_k(l) = f_k^{int}(l) \bmod l$$

where $f_k^{int}(l)$ is the big-endian integer representation of

$$DM_k(x+1) \parallel DM_k(x+2) \parallel DM_k(x+3)$$

where $DM_k(m) = AES_k(m) \oplus m$ is the AES encryption of m using key k in Davies-Meyer mode, $x+1$, $x+2$ and $x+3$ are computed incrementing x by 1 each time.

The 128 bit input x for AES is obtained from the counter $l = (i, j)$, iterated by the RA, as $(0^{32} \parallel i \parallel j \parallel 0^{32})$. i is a value which may represent a week index, j is a counter within i representing the number of certificates per i. These values are set by the SCMS Manager. The expansion function for encryption keys is defined analogously, except x is derived as $(1^{32} \parallel i \parallel j \parallel 0^{32})$.

The RA generates up to 2^{128} *cocoon public keys* as:

$$B_l = A + f_k(l) * G$$

The corresponding private keys, known only to the OBE are:

$$b_l = a + f_k(l)$$

The RA includes the cocoon public keys in the certificate request sent to the PCA. However, these public keys are not directly used by the PCA, because the RA could recognize the public keys in the certificates and track the OBE, knowing which public keys originate from each request. For each cocoon public key B_l, the PCA generates a random value c_l and computes:

$$C_l = c_l G$$

The butterfly public key included in the certificates is:

$$B_l + C_l$$

The PCA returns to the RA the certificate and the private key reconstruction value c, encrypted to the OBE. The private keys are updated by the OBE as:

$$b_l' = b_l + c_l$$

To prevent the PCA from matching the certificates with a specific OBE, each certificate is encrypted using a different encryption key.

The encryption keys are generated using the butterfly mechanism. The OBE generates a *caterpillar encryption public key*:

$$H = hG$$

The RA expands it to generate the *cocoon public encryption keys*, which the PCA will use to encrypt the responses:

$$J_l = H + f_e(l)G$$

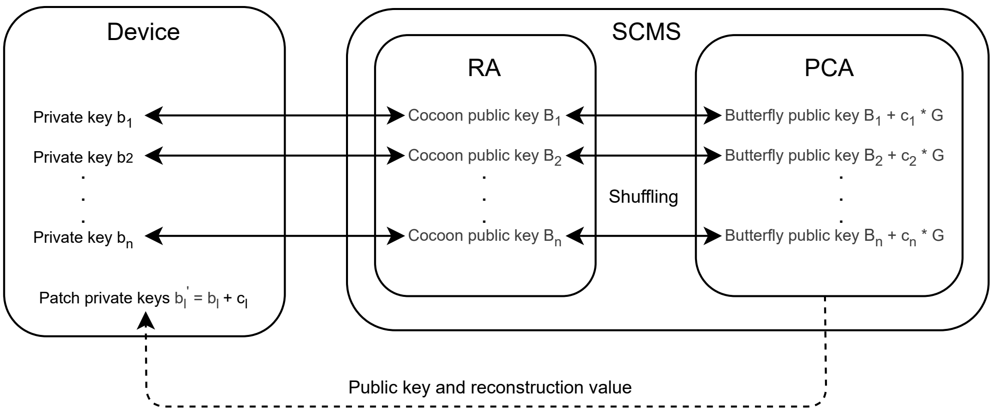


Figure 5.2: Butterfly Key Expansion scheme

5.5 Linkage Values

For each set of pseudonym certificates issued to a device, the SCMS inserts the *linkage values* to support revocation of certificates with validity greater than or equal to the time period i.

The linkage values are computed by the PCA by XORing the linkage values provided by LA1 and LA2. Each LA is assigned a 32-bit identity string, denoted la_id_1 and la_id_2. For a certificate set, each LA chooses a random 128-bit string called the *initial linkage seed*, denoted $ls_1(0)$ and $ls_2(0)$. The *linkage chain identifiers* (LCIs) are the initial linkage seeds. For each time period $i > 0$, each LA computes the *linkage seed* as follows:

$$ls_1 = H_u(la\_id_1 \parallel ls_1(i-1)),\ ls_2 = H_u(la\_id_2 \parallel ls_2(i-1))$$

where $H_u(m)$ denotes the u (suggested value: 16) most significant bytes of the SHA-256 hash of m. Each LA encrypts the linkage seeds and uses the result to compute pre-linkage values:

$$plv_1(i,j) = [E(ls_1(i), (la\_id_1 \parallel j)) \oplus (la\_id_1 \parallel j)]_v$$

$$plv_2(i,j) = [E(ls_2(i), (la\_id_2 \parallel j)) \oplus (la\_id_2 \parallel j)]_v$$

where $E(k,m)$ denotes AES encryption of m using the key k, and $[a]_v$ denotes the v (suggested value: 9) significant bytes of a.

The LAs encrypt the pre-linkage values for the PCA, and forward them to the RA. The PCA XORs the pre-linkage values to compute the linkage value $lv = plv_1 \oplus plv_2$.

5.6 SCMS Steps

The SCMS supports four main functionalities:

- **Device Bootstrapping**: The device receives all the information needed to join the SCMS.
- **Certificate Provisioning**: The device receives all the certificates needed to participate in the SCMS.
- **Misbehavior Reporting**: Process used to report misbehaving devices to the SCMS.
- **Global Misbehavior Detection and Revocation**: Process to detect misbehavior and, when necessary, revoke certificates of misbehaving devices.

5.6.1 Device Bootstrapping

The device bootstrapping phase provides the device with all the information required to communicate with all SCMS components. This process requires the participation of the device, the DCM, the ECA, and the CS components. The DCM communicates with the device over an out-of-band secure channel. A scheme of device bootstrapping process is shown in Fig. 5.3. The bootstrapping process is divided into two steps:

1. **Initialization**: The device receives the certificates needed to verify and trust received messages. The device also receives the certificates of all electors, all root CAs, the ICAs, the PCAs, the MA, the PG, and the CRLG.

2. **Enrollment**: The device receives the enrollment certificate used to sign outgoing messages. The device also receives the certificates of the ECA and the RA. During enrollment, the CS provide the DCM with information about eligible devices.

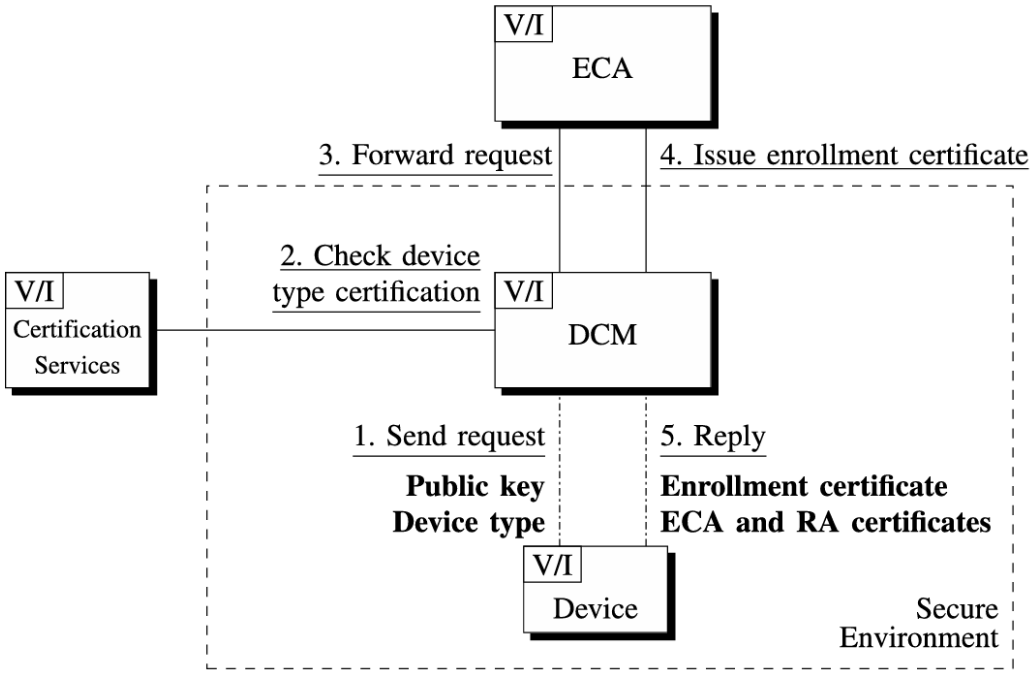


Figure 5.3: Device Bootstrapping scheme

5.6.2 Certificate Provisioning

The certificate provisioning phase provides the device with the certificates needed to participate in the SCMS. A scheme of this process is shown in Fig. 5.4. The pseudonym certificate provisioning explained in this section requires specific processes to protect user privacy and reduce the computational effort required by devices:

- **Provisioning**: The SCMS is designed such that the RA, which delivers pseudonym certificates to devices, cannot read certificate contents because the certificates are encrypted for the devices by the PCA. The PCA generates the pseudonym certificates but is not aware of their recipients, since the certificate requests are shuffled by the RA before being sent to the PCA.

- **Revocation**: The PCA inserts linkage values generated by the LAs into the certificates. The MA unmasks the linkage values by publishing a secret linkage seed pair on the CRL for certificate revocation. However, to track or revoke a device, the LAs, the PCA, and the RA must collaborate.

- **Obscuring Physical Location**: The LOP obscures the physical location of the device from the RA and the MA.

- **Butterfly Key Expansion**: This mechanism prevents correlating the public key seeds in the certificate requests with the generated certificates.

The certificate provisioning process is devided in 6 steps:

1. **Device Certificate Request**: The device creates a certificate request generating the butterfly key seeds and signs it using its enrollment certificate. The device sends the request and the enrollment certificate encrypted to the RA via the LOP.

2. **RA handles request**: The RA decrypt the request, verifies the device's enrollment certificate signature and that it is not revoked, and checks if the only request by the device. If all checks succeed the RA performs the butterfly key expansion. The RA collects several requests, the linkage values received by the LAs, and shuffles them.

3. **RA sends certificates requests**: The RA sends the pseudonym certificate requests to the PCA. Each request contains a to-be-signed certificate, the response encryption public key, the pre-linkage values $plv_1(i, j)$ and $plv_2(i, j)$, and the hash of the pseudonym certificate request itself.

4. **PCA generates certificate**: The PCA decrypts the pre-linkage values and computes the linkage value $lv(i, j) = plv_1(i, j) \oplus plv_2(i, j)$. It adds the linkage values to the certificate and signs it. Then the PCA generates a private key reconstruction value, and encrypts that value and the pseudonym certificate using the response encryption public key.

5. **PCA sends certificates to the RA**: The PCA signs the message created in step 4 and sends it to the RA.

6. **RA bundles certificates**: The RA receives the responses from the PCA and bundles them into batches per device for download.

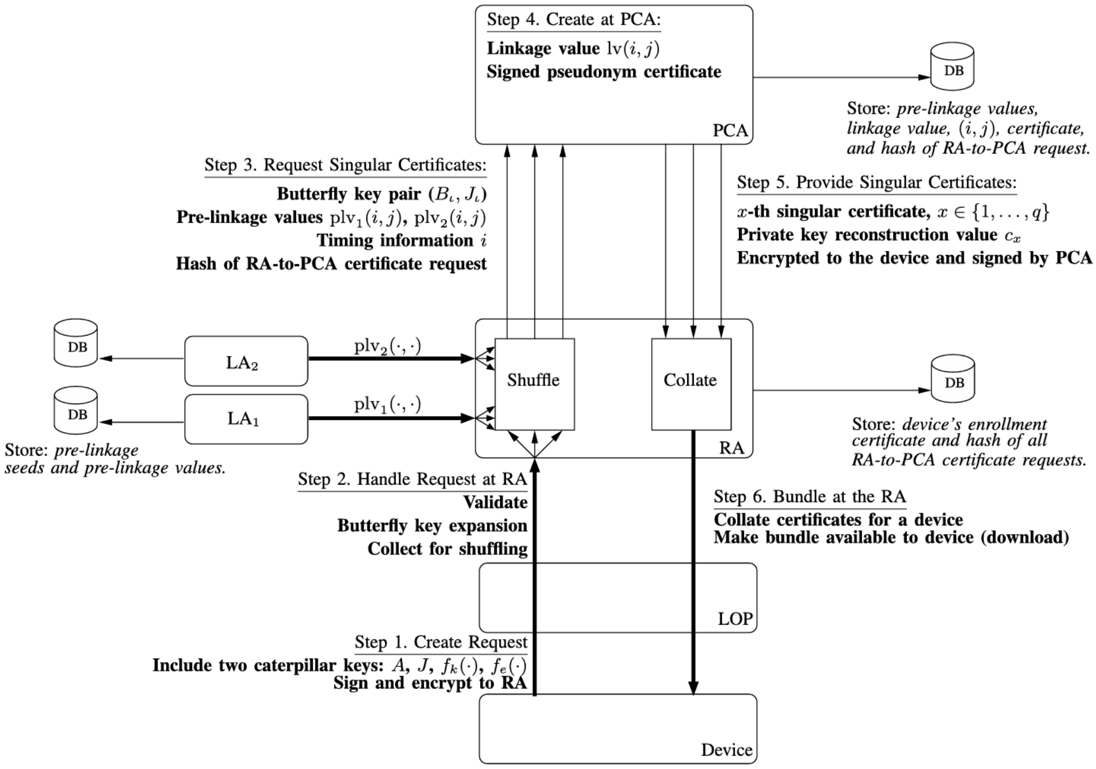


Figure 5.4: Certificate Provisioning scheme

5.6.3 Misbehavior Reporting

Misbehavior reporting aims to report the intentional or unintentional misbehaving devices to the SCMS. The misbehavior detection algorithms can be executed on the devices or within the SCMS. In the second case the SCMS informs participants about the certificates that are no longer trustworthy.

Misbehavior reports are sent by devices to the RA, which shuffles the reports to prevent reporter tacking and then forwards them to the MA. A report includes the potentially bogus BSMs, the corresponding pseudonym certificates, the misbehavior type, and the reporter's pseudonym certificate. The report is signed, encrypted, and sent to the MA.

5.6.4 Global Misbehavior Detection and Revocation

Global Misbehavior Detection aims to detect system misbehaviors, investigate them, and, if necessary, revoke the certificates of those devices. This process is executed by the MA. Fig. 5.5 illustrates this process. The following steps explain the phases of the process. Step 1 is Misbehavior Reporting, Step 2 is Global Misbehavior Detection, and Steps 3-4 cover Misbehavior Investigation.

- **Step 1**: The MA receives misbehavior reports containing the reported pseudonym certificate.
- **Step 2**: The MA runs the global misbehavior detection algorithms to identify the device to investigate.
- **Step 3**: The MA requests that PCA maps linkage values contained in the pseudonym certificate to the corresponding encrypted pre-linkage values. The PCA returns the encrypted pre-linkage values to the MA.
- **Step 4**: The MA requests to the LAs to find the set of encrypted pre-linkage values that correspond to the same device.

If the Misbehavior Investigation discovers that a device has misbehaved, the following steps are executed to revoke the device's certificates. Steps 3-4 relate to Misbehavior Investigation.

- **Step 3**: The MA requests to the PCA to map the linkage value of the target pseudonym certificate to the corresponding hash value of the RA-to-PCA pseudonym certificate request. The PCA returns the requested information.
- **Step 4**: The MA sends the hash of the pseudonym certificate request to the RA which is able to map it to the corresponding enrollment certificate to add it in its blacklist. The MA also receives the hostnames of the LAs that participated in the issuance of the pseudonym certificates linkage values, and their LCIs.
- **Step 5**: The MA requests to the LAs to map the LCIs, lci_1 and lci_2 to the linkage seeds $ls_1(i)$ and $ls_2(i)$, where i is a time period. The LAs also returns to the MA their LA IDs, la_id_1 and la_id_2. A linkage seed and the corresponding LA ID only allow to compute the forward linkage seeds (for $j \geq i$), providing backward privacy.
- **Step 6**: The MA adds the linkage seeds $ls_1(i)$ and $ls_2(i)$ and the LA IDs la_id_1 and la_id_2 to the CRL. Finally the CRLG signs and publishes the CRL.

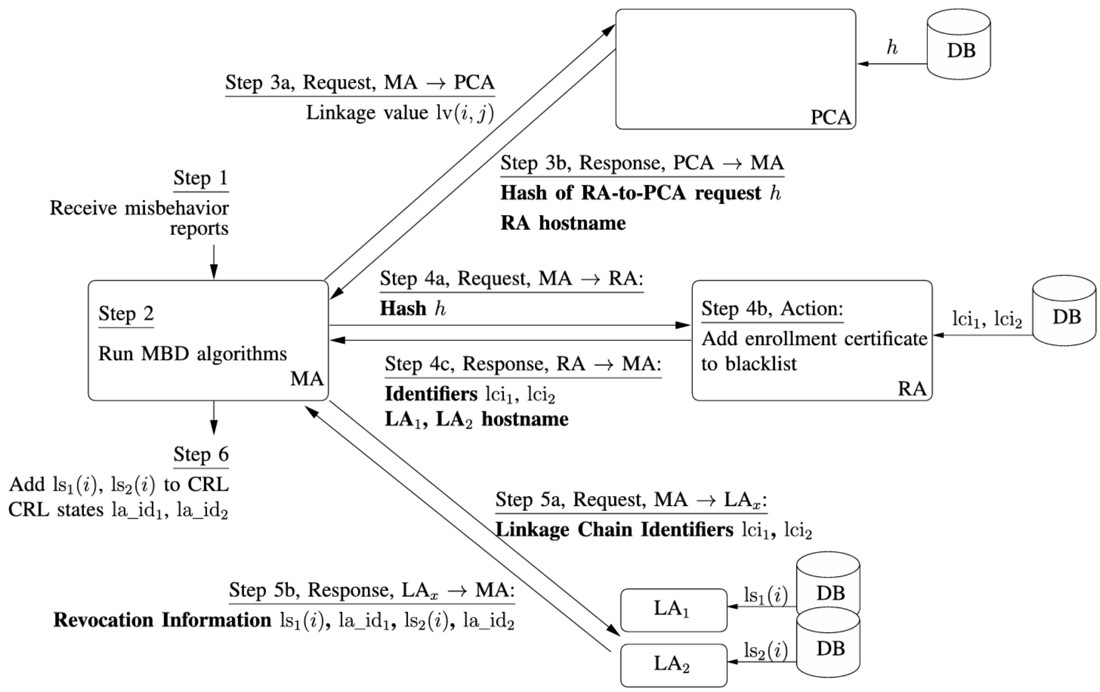


Figure 5.5: Misbehavior Detection and Revocation Scheme

Chapter 6

Security Credential Management System Tamarin Model

In the following sections, the Tamarin model of the SCMS for the Device Bootstrapping, Certificate Provisioning, and Certificate-Revocation protocol steps is presented.

6.1 Builtins

The built-in models used are `signing`, `asymmetric-encryption`, and `hashing`.

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

Signing
This model adds the following functions:

- **`true/0`**: Denotes the output value returned by a successful signature verification.
- **`pk/1`**: Denotes the algorithm to compute a public key from a private key.
- **`sign/2`**: Represents a signature algorithm. It takes as input a message and a private key.
- **`verify/3`**: Represents a signature verification algorithm. It takes as input the signature, the message and a public key. The output is equal to true if the signature is verified successfully.

and defines the following equation:
`verify(sign(m, k), m, pk(k)) = true`.

Asymmetric-encryption
This model adds the following functions:

- **`aenc/2`**: Represents an asymmetric encryption algorithm. It takes as input a message and a public key.
- **`adec/2`**: Represents an asymmetric decryption algorithm. It takes as input an encrypted message and a private key.

and defines the following equation:
`adec(aenc(m, pk(k)), k) = m`.

Hashing
This model adds the function **`h/1`** to model a hash function that takes a message as input.

6.2 Custom Function and Equations

In addition to the functions and rules provided by the built-in Tamarin models, we define custom functions specifically to model the butterfly-key expansion.

```
functions: bksum/2, bksign/2, bkverify/3, bpk/1, bkaenc/2, bkadec/2,
           myxor/2
```

- **`bksum/2`**: Models the sum operation.
- **`bksign/2`**: Represents a signature algorithm. It takes as input a message and a butterfly generated private key.
- **`bkverify/3`**: Represents a signature verification algorithm. It takes as input the signature, the message and a butterfly public key.
- **`bpk/1`**: Denotes the algorithm to generate a public key from a private key.
- **`bkaenc/2`**: Models an asymmetric encryption algorithm. It takes as input a message and a butterfly public key.
- **`bkadec/2`**: Models an asymmetric decryption algorithm. It takes as input an encrypted message and a butterfly private key.
- **`myxor/2`**: Models a simple XOR operation.

The defined equations are the following:

```
equations:
    bkadec(bkaenc(message,bksum(bpk(enc_sk_seed),bpk(enc_expansion_fun))),
    bksum(enc_sk_seed,enc_expansion_fun)) = message

    bkverify(bksign(message,bksum(bksum(sign_sk_seed, sign_expansion_fun),
    rec_value)),m,bksum(bksum(bpk(sign_sk_seed),bpk(sign_expansion_fun)),
    bpk(rec_value))) = true
```

6.3 Restrictions

Two restrictions are needed to model the protocol: `equality` and `onlyonce`. The equality restriction ensures that, in any rule with an `Eq(x,y)` action, `x` and `y` are equal. The onlyonce restriction is used to specify that an action occours at most once in a trace.

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

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

6.4 Preliminary Rules

The following section explains the rules common to all protocol steps.

Public Key Registration

This rule models the registration of a public key in a PKI infrastructure. The premise contains the generation of a secret long term key (`~ltk`), while in the conclusion the public key (`pk(~ltk)`) is transmitted over the public output channel.

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

Authorities Initialization

These rules model the initalization of the RCA, the MA, the ICA, the PCA, the ECA, the RA, the LA1, and the LA2. The `OnlyOnce` fact used in these rules is needed to ensure that each authority is initialized only once.

```
rule rca_initialization:
    [ !Ltk($RCA, ~ltk_rca) ]
  --[ OnlyOnce('RCA_initialization'),
      OnlyOnce($RCA),
      RCA_Initialization($RCA) ]->
    [ !RCA_Initialized($RCA, ~ltk_rca, pk(~ltk_rca)) ]

rule ma_initialization:
    [ !Ltk($MA, ~ltk_ma) ]
  --[ OnlyOnce('MA_initialization'),
      OnlyOnce($MA),
      MA_Initialization($MA) ]->
    [ !MA_Initialized($MA, ~ltk_ma, pk(~ltk_ma)) ]

rule ica_initialization:
    [ !Ltk($ICA, ~ltk_ica) ]
  --[ OnlyOnce('ICA_initialization'),
      OnlyOnce($ICA),
      ICA_Initialization($ICA) ]->
    [ !ICA_Initialized($ICA, ~ltk_ica, pk(~ltk_ica)) ]

rule pca_initialization:
    [ !Ltk($PCA, ~ltk_pca) ]
  --[ OnlyOnce('PCA_initialization'),
      OnlyOnce($PCA),
      PCA_Initialization($PCA) ]->
    [ !PCA_Initialized($PCA, ~ltk_pca, pk(~ltk_pca)) ]

rule eca_initialization:
    [ !Ltk($ECA, ~ltk_eca) ]
  --[ OnlyOnce('ECA_initialization'),
      OnlyOnce($ECA),
      ECA_Initialization($ECA) ]->
    [ !ECA_Initialized($ECA, ~ltk_eca, pk(~ltk_eca)) ]

rule ra_initialization:
    [ !Ltk($RA, ~ltk_ra) ]
  --[ OnlyOnce('RA_initialization'),
      OnlyOnce($RA),
      RA_Initialization($RA) ]->
    [ !RA_Initialized($RA, ~ltk_ra, pk(~ltk_ra)) ]

rule la1_initialization:
    [ !Ltk($LA1, ~ltk_la1) ]
  --[ OnlyOnce('LA1_initialization'),
```

```
      OnlyOnce($LA1),
      LA1_Initialization($LA1) ]->
    [ !LA1_Initialized($LA1, ~ltk_la1, pk(~ltk_la1)) ]

rule la2_initialization:
    [ !Ltk($LA2, ~ltk_la2) ]
  --[ OnlyOnce('LA2_initialization'),
      OnlyOnce($LA2),
      LA2_Initialization($LA2) ]->
    [ !LA2_Initialized($LA2, ~ltk_la2, pk(~ltk_la2)) ]
```

Secure Channel
The following rules model a secure channel that provides confidentiality and authenticity. The `Out_s` rule is used by a sender `A` to send a message `x` to a receiver `B`. The `In_s` rule is used by a receiver `B` to receive a message `x` from a sneder `B`. The adversary cannot obtain the message contents or modify the sender or the receiver, but it can reply messages since the conclusion of the output rule is a persistent fact.

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

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

6.5 Device Bootstrapping Model

In this phase, the device receives the enrollment certificate and all the other certificates required to participate in the SCMS. The secure environment described by the protocol is modelled using the secure-channel rules introduced above.

In the **`device_initialization`** rule, the device is initialized generating an identifier (∼`id`), and obtaining the secret long term key (∼`ltk`). The `OnlyOnce` fact ensures that the device can be initialized only once.

```
rule device_initialization:
    [ Fr(~id),
      !Ltk($Device, ~ltk_device)
    ]
```

```
  --[ Device_Initialization($Device, ~id),
      OnlyOnce(<'Device_Initialization', ~ltk_device>)
    ]->
    [ Device_Initialized($Device, ~id, ~ltk_device) ]
```

In the **device_requests_enrollment_certificate** rule, the device sends the enrollment certificate request to the ECA. For simplification, the request is modelled as being sent directly to the ECA rather than to the DCM, which would normally forward it to the ECA. Also, the device type certification check is omitted for the same reason.

```
rule device_requests_enrollment_certificate:
    let
        request = <$Device, ~id, pk(~ltk_device)>
    in
    [ Device_Initialized($Device, ~id, ~ltk_device) ]
  --[ Bootstrapping_St_1($Device, ~id) ]->
    [ Device_Requested_Enrollment_Certificate($Device, ~id, ~ltk_device),
      Out_S($Device, $ECA, request) ]
```

In the **eca_issues_enrollment_certificate** rule, the ECA receives the enrollment certificate request from the device, generates a certificate signing the device's public key, and sends the certificate to the device. The ECA also sends to the device the certificates of the RCA, the ICA, the PCA, the ECA, and the RA.

```
rule eca_issues_enrollment_certificate:
    let
        enrollment_certificate = <pk_device, sign(pk_device, ~ltk_eca)>
        rca_certificate = <pk(~ltk_rca), sign(pk_rca, ~ltk_rca)>
        ica_certificate = <pk(~ltk_ica), sign(pk_ica, ~ltk_rca)>
        pca_certificate = <pk(~ltk_pca), sign(pk_pca, ~ltk_ica)>
        eca_certificate = <pk(~ltk_eca), sign(pk_eca, ~ltk_ica)>
        ra_certificate = <pk(~ltk_ra), sign(pk_ra, ~ltk_pca)>
    in
    [ In_S($Device, $ECA, <$Device, ~id, pk_device>),
      !RCA_Initialized($RCA, ~ltk_rca, pk_rca),
      !ICA_Initialized($ICA, ~ltk_ica, pk_ica),
      !PCA_Initialized($PCA, ~ltk_pca, pk_pca),
      !ECA_Initialized($ECA, ~ltk_eca, pk_eca),
      !RA_Initialized($RA, ~ltk_ra, pk_ra)
    ]
  --[ Bootstrapping_St_2($Device, ~id),
      Enrollment_Certificate_Issued($Device, $ECA, <$Device, ~id,
      pk_device>),
      OnlyOnce(<'Enrollment_Certificate_Issuance', $Device, $ECA,
```

```
      <$Device, ~id, pk_device>>)
    ]->
    [ Out_S($ECA, $Device, <enrollment_certificate, rca_certificate,
      ica_certificate, pca_certificate, eca_certificate, ra_certificate>) ]
```

The **device_receives_enrollment_certificate** rule is the final step of the device bootstrapping phase, where the device receives the enrollment certificate sent by the ECA.

```
rule device_receives_enrollment_certificate:
    let
        pk_device = fst(enrollment_certificate)
    in
    [ In_S($ECA, $Device, <enrollment_certificate, rca_certificate,
      ica_certificate, pca_certificate, eca_certificate, ra_certificate>),
      Device_Requested_Enrollment_Certificate($Device, ~id, ~ltk_device)
    ]
  --[ Bootstrapping_St_3($Device, ~id),
      Eq(pk_device, pk(~ltk_device))
    ]->
    [ Device_Bootstrapping_Complete($Device, ~id, ~ltk_device,
      enrollment_certificate) ]
```

6.6 Certificate Provisioning Model

In this phase, the device requests and receives the pseudonym certificate used to sign the BSMs it sends. To simplify the model, the LOP and the databases described in the protocol are not modelled.

In the **device_pseudonym_certificate_request** rule, the device generates the public elements `AA` (for the signing key) and `HH` (for the encryption key) to include in the certificate request, that will be used by the RA to performs the butterfly key expansion and to generate the cocoon public keys. Then it sends the pseudonym certificate request, containing also the expansion functions, on the public output channel to the RA.

```
rule device_pseudonym_certificate_request:
    let
        AA = bpk(~aa)  // AA = aa * G
        HH = bpk(~hh)  // HH = hh * G
        request = <AA, HH, ~fk, ~fe>
        signed_request = sign(request, ~ltk_device)
        message = <request, signed_request, enrollment_certificate>
```

```
        encrypted_message = aenc(message, pk_ra)
    in
    [ !RA_Initialized($RA, ~ltk_ra, pk_ra),
      Device_Bootstrapping_Complete($Device, ~id, ~ltk_device,
      enrollment_certificate),
      Fr(~aa),  // signing private key seed
      Fr(~hh),  // encryption private key seed
      Fr(~fk),  // expansion function for signing keys
      Fr(~fe)   // expansion function for encryption keys
    ]
  --[ Pseudonym_Certificate_Provisioning_St_1($Device, ~id, request),
      Pseudonym_Certificate_Request_Sending(~ltk_device, ~ltk_ra,
      encrypted_message),
      Secret(~aa),
      Secret(~hh)
    ]->
    [ Pseudonym_Certificate_Request_Sent($Device, ~ltk_device,
      encrypted_message),
      Device_Private_Keys_Generated($Device, ~id, ~aa, ~hh, ~fk, ~fe),
      Out(encrypted_message) ]
```

The **la1_sends_plv** and **la2_sends_plv** rules, model the sending of the linkage values by LA1 and LA2 to the PCA via the public channel. The linkage values are signed by the LAs, and encrypted with the PCA's public key.

```
rule la1_sends_plv:
    let
        signed_plv1 = sign(~plv1, ~ltk_la1)
        encrypted_plv1 = aenc(~plv1, pk_pca)
    in
    [ !LA1_Initialized($LA1, ~ltk_la1, pk_la1),
      !Ltk($LA1, ~ltk_la1),
      !PCA_Initialized($PCA, ~ltk_pca, pk_pca),
      Fr(~plv1)
    ]
  --[ Plv1_Sending(~ltk_la1, ~ltk_pca, <~plv1, signed_plv1>) ]->
    [ Out(<encrypted_plv1, signed_plv1>) ]

rule la2_sends_plv:
    let
        signed_plv2 = sign(~plv2, ~ltk_la2)
        encrypted_plv2 = aenc(~plv2, pk_pca)
    in
    [ !LA2_Initialized($LA2, ~ltk_la2, pk_la2),
      !Ltk($LA2, ~ltk_la2),
```

```
      !PCA_Initialized($PCA, ~ltk_pca, pk_pca),
      Fr(~plv2)
    ]
  --[ Plv2_Sending(~ltk_la2, ~ltk_pca, <~plv2, signed_plv2>) ]->
    [ Out(<encrypted_plv2, signed_plv2>) ]
```

In the **ra_handle_request** rule, the RA receives the pseudonym certificate request from the device and the linkage values from the LAs. The RA verifies the device's enrollment certificate signature and the signature on the request, then performs the butterfly-key expansion to generate cocoon public keys for signing (B) and encryption (J). Finally, the RA creates a certificate request that includes the hash of the original request, encrypts it with the PCA's public key, and sends it on the public channel to the PCA.

```
rule ra_handle_request:
    let
        message = adec(encrypted_message, ~ltk_ra)
        request = fst(message)
        request_signature = fst(snd(message))
        enrollment_certificate = snd(snd(message))
        enrollment_certificate_key = fst(enrollment_certificate)
        enrollment_certificate_signature = snd(enrollment_certificate)
        AA = fst(request)
        HH = fst(snd(request))
        fk = fst(snd(snd(request)))
        fe = snd(snd(snd(request)))
        B = bksum(AA, bpk(fk)) // B = AA + (fk * G)
        J = bksum(HH, bpk(fe)) // J = HH + (fe * G)
        new_request = <B, J, encrypted_plv1, signed_plv1,
        encrypted_plv2, signed_plv2>
        new_message = <new_request, h(new_request)>
        new_encrypted_message = aenc(new_message, pk_pca)
    in
    [ !RA_Initialized($RA, ~ltk_ra, pk_ra),
      !ECA_Initialized($ECA, ~ltk_eca, pk_eca),
      !PCA_Initialized($PCA, ~ltk_pca, pk_pca),
      Pseudonym_Certificate_Request_Sent($Device, ~ltk_device,
      encrypted_message),
      In(encrypted_message),
      In(<encrypted_plv1, signed_plv1>),
      In(<encrypted_plv2, signed_plv2>)
    ]
  --[ Eq(verify(enrollment_certificate_signature,
      enrollment_certificate_key, pk_eca), true),
      Eq(verify(request_signature, request, enrollment_certificate_key),
```

```
      true),
      Pseudonym_Certificate_Provisioning_St_2(request),
      Pseudonym_Certificate_Request_Receiving(~ltk_device, ~ltk_ra,
      encrypted_message),
      RA_Handling_Request(~ltk_ra, ~ltk_pca, new_encrypted_message)
    ]->
    [ Out(new_encrypted_message) ]
```

In the **pca_handle_request** rule, the PCA decrypts the message from the RA, verifies the signatures of the linkage values, and checks the hash included in the message. Then the PCA generates the private keys reconstruction value and the device's public key. Finally the PCA creates the pseudonym certificate, signs it, encrypts it for the device, and sends the encrypted message on the public channel.

```
rule pca_handle_request:
    let
        message = adec(new_encrypted_message, ~ltk_pca)
        new_request = fst(message)
        request_hash = snd(message)
        computed_hash = h(new_request)
        B = fst(new_request)
        J = fst(snd(new_request))
        Bc = bksum(B, bpk(~c))  // B + c * G
        plv1 = adec(fst(snd(snd(new_request))), ~ltk_pca)
        plv2 = adec(fst(snd(snd(snd(snd(new_request))))), ~ltk_pca)
        signed_plv1 = fst(snd(snd(snd(new_request))))
        signed_plv2 = snd(snd(snd(snd(snd(new_request)))))
        lv = myxor(plv1, plv2)
        tbs_certificate = <Bc, lv>
        certificate_signature = sign(tbs_certificate, ~ltk_pca)
        certificate = <tbs_certificate, certificate_signature>
        encrypted_response = bkaenc(<certificate, ~c>, J)
        signed_encrypted_response = sign(encrypted_response, ~ltk_pca)
    in
    [ !RA_Initialized($RA, ~ltk_ra, pk_ra),
      !PCA_Initialized($PCA, ~ltk_pca, pk_pca),
      !LA1_Initialized($LA1, ~ltk_la1, pk_la1),
      !LA2_Initialized($LA2, ~ltk_la2, pk_la2),
      In(new_encrypted_message),
      Fr(~c)  // private keys reconstruction value
    ]
  --[ Eq(verify(signed_plv1, plv1, pk_la1), true),
      Eq(verify(signed_plv2, plv2, pk_la2), true),
      Eq(request_hash, computed_hash),
      Pseudonym_Certificate_Provisioning_St_3(encrypted_response),
```

```
      Plv1_receiving(~ltk_la1, ~ltk_pca, <plv1, signed_plv1>),
      Plv2_receiving(~ltk_la2, ~ltk_pca, <plv2, signed_plv2>),
      PCA_Handling_Request(~ltk_ra, ~ltk_pca, new_encrypted_message),
      PCA_Sending_Pseudonym_Certificate(~ltk_pca, <encrypted_response,
      signed_encrypted_response>)
    ]->
    [ Out(<encrypted_response, signed_encrypted_response>) ]
```

The **device_receives_pseudonym_certificate** rule, is the final step of the certificate provisioning process. For simplicity, the model assumes the device receives certificates directly rather than routing them through the RA. The device decrypts the received message, verifies the signatures of the certificate and on the message, and generates its signature private keys using the reconstruction value provided by the PCA.

```
rule device_receives_pseudonym_certificate:
    let
        j = bksum(~hh, ~fe)  // encryption private key
        response = bkadec(encrypted_response, j)
        certificate = fst(response)
        tbs_certificate = fst(certificate)
        certificate_signature = snd(certificate)
        c = snd(response)
        b = bksum(bksum(~aa, ~fk), c) // signature private key
    in
    [ !PCA_Initialized($PCA, ~ltk_pca, pk_pca),
      Device_Private_Keys_Generated($Device, ~id, ~aa, ~hh, ~fk, ~fe),
      In(<encrypted_response, signed_encrypted_response>)
    ]
  --[ Eq(verify(signed_encrypted_response, encrypted_response, pk_pca),
      true),
      Eq(verify(certificate_signature, tbs_certificate, pk_pca), true),
      Pseudonym_Certificate_Provisioning_St_4($Device, ~id),
      Device_Receiving_Pseudonym_Certificate(~ltk_pca,<encrypted_response,
      signed_encrypted_response>)
    ]->
    [ Pseudonym_Certificate_Created($Device, ~id, certificate, b, j) ]
```

6.7 Certificate Revocation Model

In this phase, a device which receives a bogus BSM may send a misbehavior report to the MA. If the MA estabilish that the reported device is misbehaving, it starts the process to revoke its certificate.

The **device_revocation_initialization** rule, models the state in which the device is initialized, enrolled and already received a pseudonym certificate from the PCA.

```
rule device_revocation_initialization:
    let
        tbs_certificate = <pk(~ltk_device), ~lv>
        certificate_signature = sign(tbs_certificate, ~ltk_pca)
        pseudonym_certificate = <tbs_certificate, certificate_signature>
    in
    [ !Ltk($Device, ~ltk_device),
      !PCA_Initialized($PCA, ~ltk_pca, pk(~ltk_pca)),
      Fr(~lv)  // likage values
    ]
  --[ Device_Initialization($Device, pseudonym_certificate),
      OnlyOnce(<'Device_Initialization', ~ltk_device>),
      OnlyOnce($Device)
    ]->
    [ Device_Initialized($Device, ~ltk_device, pseudonym_certificate) ]
```

The **device_sends_message** rule, represents a device that sends a BSM on the public channel, cotaining also its pseudonym certificate and the BSM signature.

```
rule device_sends_message:
    let
        bsm_signature = sign(bsm, ~ltk_device)
        message = <bsm, bsm_signature, pseudonym_certificate>
    in
    [ Device_Initialized($Device, ~ltk_device, pseudonym_certificate),
      Fr(bsm)  // basic safety message
    ]
  --[ BSM_Sending(pseudonym_certificate, bsm),
      Device_Sending_Message($Device, bsm)
    ]->
    [ Out(message) ]
```

In the **device_reports_misbehavior** rule, the device receives a BSM from the public channel and sends a misbehavior report encrypted to the MA containing its own pseudonym certificate and the BSM sender's pseudonym certificate.

```
rule device_reports_misbehavior:
    let
        bsm = fst(message)
        bsm_signature = fst(snd(message))
        sender_pseudonym_certificate = snd(snd(message))
```

```
        sender_pseudonym_certificate_signature =
        snd(sender_pseudonym_certificate)
        tbs_certificate = fst(sender_pseudonym_certificate)
        sender_pk = fst(fst(sender_pseudonym_certificate))
        misbehavior_report = <sender_pseudonym_certificate,
        pseudonym_certificate>
        misbehavior_report_signature = sign(misbehavior_report,
        ~ltk_device)
        report_message = aenc(<misbehavior_report,
        misbehavior_report_signature>,
        pk(~ltk_ma))
    in
    [ Device_Initialized($Device, ~ltk_device, pseudonym_certificate),
      !MA_Initialized($MA, ~ltk_ma, pk(~ltk_ma)),
      !PCA_Initialized($PCA, ~ltk_pca, pk(~ltk_pca)),
      In(message)
    ]
  --[ BSM_Receiving(sender_pseudonym_certificate, bsm),
      Eq(verify(bsm_signature, bsm, sender_pk),true),
      Eq(verify(sender_pseudonym_certificate_signature, tbs_certificate,
      pk(~ltk_pca)), true),
      Certificate_Revocation_St_1($Device, pseudonym_certificate,
      report_message)
    ]->
    [ Misbehavior_Report_Sent($Device, ~ltk_device, report_message),
      Out(report_message) ]
```

In the **ma_handles_revocation_request** rule, the MA receives the misbehavior report, extracts the linkage values from the pseudonym certificate of the misbehaving device, and sends them encrypted to the PCA.

```
rule ma_handles_revocation_request:
    let
        misbehavior_message = adec(encrypted_misbehavior_message,~ltk_ma)
        misbehavior_report = fst(misbehavior_message)
        misbehavior_report_signature = snd(misbehavior_message)
        bsm_sender_pseudonym_certificate = fst(misbehavior_report)
        reporter_pseudonym_certificate = snd(misbehavior_report)
        lv = snd(fst(bsm_sender_pseudonym_certificate))
        signed_lv = sign(lv, ~ltk_ma)
        reporter_pk = fst(fst(reporter_pseudonym_certificate))
        reporter_pseudonym_certificate_signature =
        snd(reporter_pseudonym_certificate)
        bsm_sender_pseudonym_certificate_signature =
        snd(bsm_sender_pseudonym_certificate)
```

```
        request = aenc(<lv, signed_lv>, pk(~ltk_pca))
    in
    [ !MA_Initialized($MA, ~ltk_ma, pk(~ltk_ma)),
      !PCA_Initialized($PCA, ~ltk_pca, pk(~ltk_pca)),
      Misbehavior_Report_Sent($Device, ~ltk_device,
      encrypted_misbehavior_message),
      In(encrypted_misbehavior_message)
    ]
  --[ Eq(verify(reporter_pseudonym_certificate_signature,
      fst(reporter_pseudonym_certificate), pk(~ltk_pca)), true),
      Eq(verify(bsm_sender_pseudonym_certificate_signature,
      fst(bsm_sender_pseudonym_certificate), pk(~ltk_pca)), true),
      Eq(verify(misbehavior_report_signature, misbehavior_report,
      reporter_pk), true),
      Certificate_Revocation_St_2(request)
    ]->
    [ Out(request) ]
```

In the **pca_handles_revocation_request** rule, the PCA sends the encrypted hash of the RA-to-PCA pseudonym certificate request to the MA.

```
rule pca_handles_revocation_request:
    let
        request = adec(encrypted_request, ~ltk_pca)
        lv = fst(request)
        signed_lv = snd(request)
        signed_hash = sign(~ra_to_pca_hash, ~ltk_pca)
        response = <~ra_to_pca_hash, signed_hash>
        encrypted_response = aenc(response, pk(~ltk_ma))
    in
    [ !MA_Initialized($MA, ~ltk_ma, pk(~ltk_ma)),
      !PCA_Initialized($PCA, ~ltk_pca, pk(~ltk_pca)),
      In(encrypted_request),
      Fr(~ra_to_pca_hash)
    ]
  --[ Eq(verify(signed_lv, lv, pk(~ltk_ma)), true),
      Certificate_Revocation_St_3(encrypted_request)
    ]->
    [ Out(encrypted_response) ]
```

In the **ma_sends_hash** rule, the MA sends the RA-to-PCA pseudonym certificate request to the RA.

```
rule ma_sends_hash:
    let
```

```
        response = adec(encrypted_response, ~ltk_ma)
        ra_to_pca_hash = fst(response)
        signed_hash = snd(response)
        message = <ra_to_pca_hash, signed_hash>
        message_signature = sign(message, ~ltk_ma)
        encrypted_message = aenc(<message,message_signature>,pk(~ltk_ra))
    in
    [ !MA_Initialized($MA, ~ltk_ma, pk(~ltk_ma)),
      !PCA_Initialized($PCA, ~ltk_pca, pk(~ltk_pca)),
      !RA_Initialized($RA, ~ltk_ra, pk(~ltk_ra)),
      In(encrypted_response)
    ]
  --[ Eq(verify(signed_hash, ra_to_pca_hash, pk(~ltk_pca)), true),
      Certificate_Revocation_St_4(message)
    ]->
    [ Out(encrypted_message) ]
```

The **`ra_revokes_certificate`** rule, is the last step of this phase. The RA finally reovkes the certificate of the misbehaving device.

```
rule ra_revokes_certificate:
    let
        decrypted_message = adec(encrypted_message, ~ltk_ra)
        message = fst(decrypted_message)
        message_signature = snd(decrypted_message)
        ra_to_pca_hash = fst(message)
        signed_hash = snd(message)
    in
    [ !MA_Initialized($MA, ~ltk_ma, pk(~ltk_ma)),
      !PCA_Initialized($PCA, ~ltk_pca, pk(~ltk_pca)),
      !RA_Initialized($RA, ~ltk_ra, pk(~ltk_ra)),
      In(encrypted_message)
    ]
  --[ Eq(verify(message_signature, message, pk(~ltk_ma)), true),
      Eq(verify(signed_hash, ra_to_pca_hash, pk(~ltk_pca)), true),
      Certificate_Revocation_St_5(message)
    ]->
    [  ]
```

Chapter 7

Security Properties Analysis

7.1 SCMS Threat Modeling

This section analyses the security properties for the Device Bootstrapping and Certificate Provisioning steps of the SCMS protocol.

7.1.1 Device Bootstrapping Threat Modeling

The threat model for the Device Bootstrapping steps is shown in Fig. 7.1. The involved entities are:

- **ECA**: Interacts with the DCM to request and receive the Device enrollment certificate.
- **Certification Services**: Interacts with the DCM to provide the information about the eligible device models.
- **Device**: Interacts with the DCM to request and receive all the bootstrapping information.

The ECA, the DCM and the Certification Services are modelled as processes, the Device is modelled as an external interactor. All communications between entities are modelled using HTTPS secure channels.

The tool reported 44 high priority threats belonging to the following STRIDE categories:

- **Spoofing**: An attacker may spoof the ECA, the Device, the DCM, or the Certification Services.
- **Repudiation**: The ECA, the Device, the DCM or the Certification Services may repudiate the sent data.

- **Denial of Service**: The ECA, the DCM, the Certification Services may be targeted by DoS attack or the data flows between the entities may be interrupted.

- **Elevation of Privilege**: The ECA, the DCM or the Certification Services may be subject of attacks to obtain higher privileges using impersonation, remote code execution, or changing the execution flow.

The Elevation of Privilege and the Denial of Service threats cannot be modelled using Tamarin. The Spoofing and Repudiation threats can be modelled using Tamarin and can be mitigated by signing messages with the sender's private key.

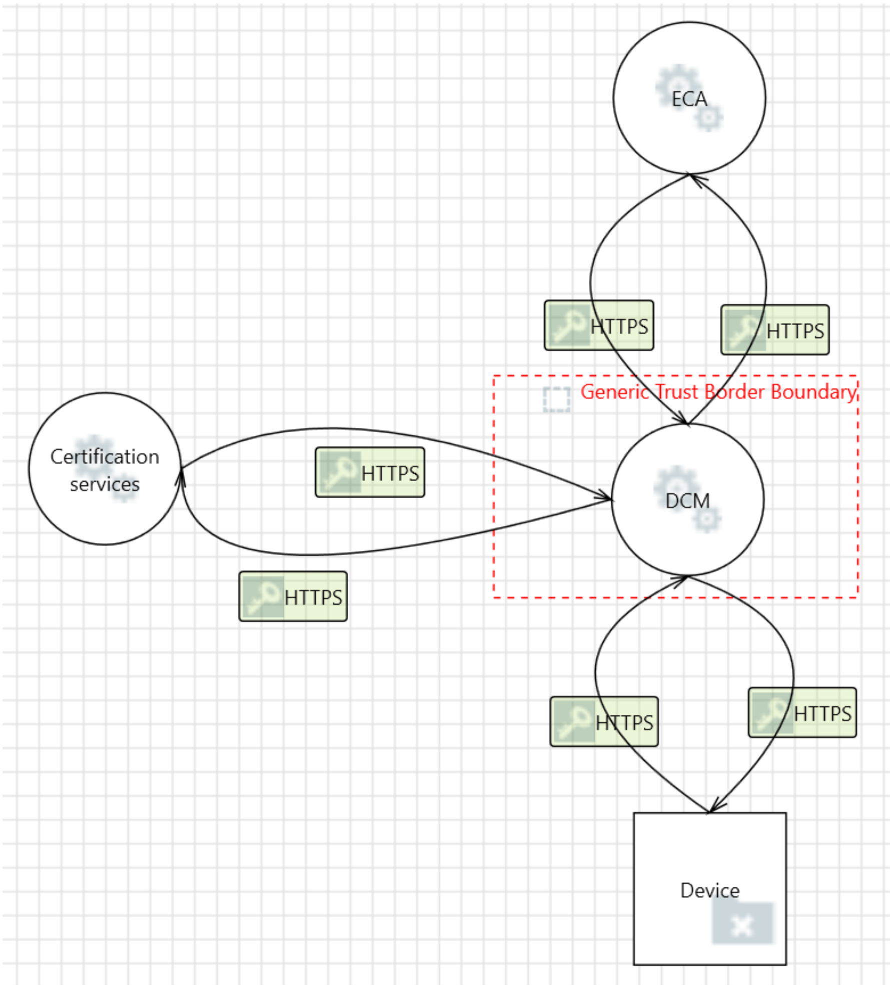


Figure 7.1: Device Bootstrapping Threat Model

7.1.2 Certificate Provisioning Threat Model

The threat model for the Certificate Provisioning step is shown in Fig. 7.2. The involved entities are:

- **Device**: Interacts with the RA to request and receive the certificates.
- **PCA**: Interacts with the RA to generate the device certificates.
- **RA**: Interacts with all the other entities to support the certificate provisioning process.
- **LAs**: Interact with the RA to provide the pre-linkage values.
- **Database**: Used by the PCA, the LAs, and the RA as data store.

The PCA, the RA, and the LAs are modelled as processes, the Device is modelled as an external interactor, and the Database as a data store. All communications between entities are modelled using HTTPS secure channels.

The tool reported 71 high priority threats belonging to the following STRIDE categories:

- **Spoofing**: An attacker may spoof the RA, thePCA, the LAs, the Device, or the destination database.
- **Tampering**: An attacker may corrupt data sored in the database.
- **Repudiation**: The RA, the PCA, the Device may repudiate sent data. The database may deny that data was written.
- **Denial of Service**: The RA, the PCA, or the data store may be targeted by DoS attacks, and the data flows between the entities may be interrupted.
- **Elevation of Privilege**: The RA and the PCA may be targeted by attacks that obtain higher privileges via impersonation, remote code execution, or changing the execution flow.

The Elevation of Privilege, the Tampering and the Denial of Service threats cannot be modelled using Tamarin. The Spoofing and Repudiation threats can be modelled and can be mitigated by signing messages with the sender's private key.

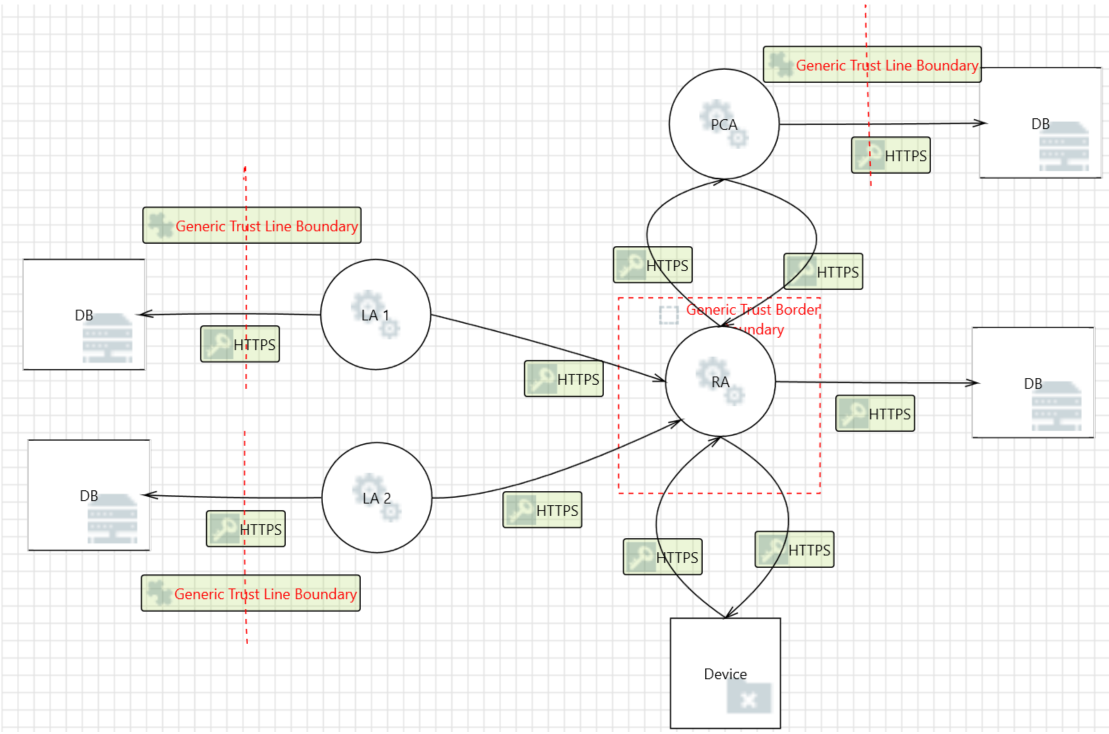


Figure 7.2: Certificate Provisioning Threat Model

7.2 Sanity Checks

Sanity checks are used to verify that the protocol model is constructed correctly and to detect potential modelling errors. The following sections show the sanity check results for the Device Bootstrapping, Certificate Provisioning, and Certificate Revocation phases. These checks are performed by proving a lemma that specifies a particular execution order of the protocol steps. If Tamarin manages to prove the lemma, the check is successful.

7.2.1 Device Bootstrapping and Certificate Provisioning

The lemma used for the sanity checks is the following:

```
lemma sanity_checks:
exists-trace
    "
        Ex Device Id Request Encrypted_Response
        #i1 #i2 #i3 #i4 #i5 #i6 #i7 #i8 .
        Device_Initialization(Device, Id) @#i1 &
```

```
        Bootstrapping_St_1(Device, Id) @#i2 &
        Bootstrapping_St_2(Device, Id) @#i3 &
        Bootstrapping_St_3(Device, Id) @#i4 &
        Pseudonym_Certificate_Provisioning_St_1(Device,Id,Request) @#i5 &
        Pseudonym_Certificate_Provisioning_St_2(Request) @#i6 &
        Pseudonym_Certificate_Provisioning_St_3(Encrypted_Response) @#i7 &
        Pseudonym_Certificate_Provisioning_St_4(Device, Id) @#i8 &
        #i1 < #i2 & #i2 < #i3 & #i3 < #i4 &
        #i4 < #i5 & #i5 < #i6 & #i6 < #i7 & #i7 < #i8
    "
```

The protocol steps for the Device Bootstrapping and Certificate Provisioning are:

1. The `Device` completes the initialization process.
2. The `Device` sends an enrollment certificate request.
3. The The ECA issues the enrollment certificate and sends it to the `Device`.
4. The `Device` receives the enrollment certificate.
5. The `Device` sends the pseudonym certificate request.
6. The RA handles the device's pseudonym-certificate request.
7. The PCA issues the pseudonym certificate.
8. The `Device` receives the pseudonym certificate.

Tamarin correctly proved the sanity checks lemma, as shown in Fig. 7.3.

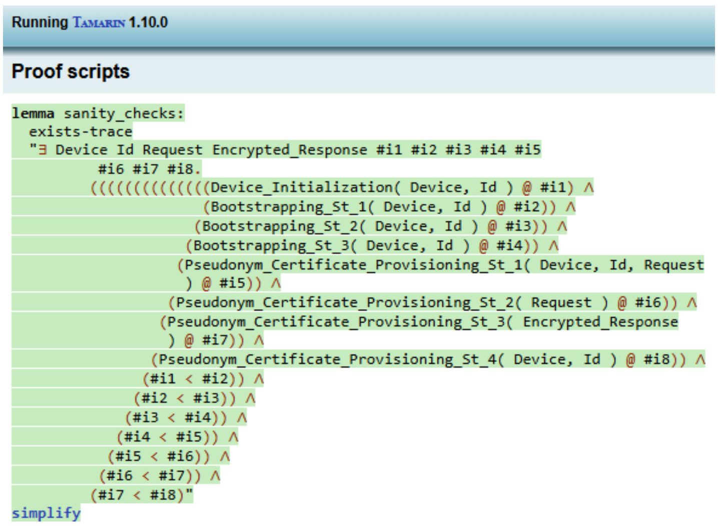


Figure 7.3: Device Bootstrapping and Certificate Provisioning sanity checks proof

7.2.2 Certificate Revocation

The lemma used for sanity checks is the following:

```
lemma sanity_checks:
exists-trace
    "
        Ex Device1 Device2 Cert1 Cert2 BSM Report Request Message
        #i1 #i2 #i3 #i4 #i5 #i6 #i7 #i8.
        Device_Initialization(Device1, Cert1) @ #i1 &
        Device_Initialization(Device2, Cert2) @ #i2 &
        Device_Sending_Message(Device1, BSM) @ #i3 &
        Certificate_Revocation_St_1(Device2, Cert2, Report) @ #i4 &
        Certificate_Revocation_St_2(Request) @ #i5 &
        Certificate_Revocation_St_3(Request) @ #i6 &
        Certificate_Revocation_St_4(Message) @ #i7 &
        Certificate_Revocation_St_5(Message) @ #i8 &
        #i1 < #i3 & #i1 < #i4 & #i3 < #i4 & #i4 < #i5 & #i5 < #i6 &
        #i6 < #i7 & #i7 < #i8
    "
```

The protocol steps are the following:

1. `Device1` and `Device2` complete the initialization process.
2. `Device1` sends a BSM via the public channel.
3. `Device2` sends the misbehavior report to the MA
4. The MA sent a requesto to the PCA, starting the revocation process.
5. The PCA handles the received request from the MA.
6. The MA sends the RA-to-PCA pseudonym certificate request hash to the RA.
7. The RA revokes the `Device1` certificate.

In this case, Tamarin was executed using the `--auto-sources` flag because of the presence of partial deconstructions. Tamarin generated the source lemma shown in Fig. 7.4, which Tamain proved successfully.

The sanity checks lemma described above was correctly proved by Tamarin, as shown in Fig. 7.5

```
lemma AUTO_typing [sources]:
  all-traces
  "((⊤) ∧
        (∀ x m #i.
          (AUTO_IN_TERM_3_0_0_0__ma_sends_hash___VARIANT_2( m, x ) @ #i) ⇒
          ((∃ #j. (!KU( x ) @ #j) ∧ (#j < #i)) ∨
           (∃ #j.
             (AUTO_OUT_TERM_3_0_0_0__ma_sends_hash___VARIANT_2( m ) @ #j) ∧
             (#j < #i))))) ∧
       (∀ x m #i.
         (AUTO_IN_TERM_3_0_0_1_0__ma_sends_hash___VARIANT_2( m, x ) @ #i) ⇒
         ((∃ #j. (!KU( x ) @ #j) ∧ (#j < #i)) ∨
          (∃ #j.
            (AUTO_OUT_TERM_3_0_0_1_0__ma_sends_hash___VARIANT_2( m ) @ #j) ∧
            (#j < #i))))"
by sorry
```

Figure 7.4: Certificate Revocation source lemmma

Running Tamarin 1.10.0

Proof scripts

```
lemma sanity_checks:
  exists-trace
  "∃ Device1 Device2 Cert1 Cert2 BSM Report Request Message #i1
          #i2 #i3 #i4 #i5 #i6 #i7 #i8.
         ((((((((((((Device_Initialization( Device1, Cert1 ) @ #i1) ∧
                    (Device_Initialization( Device2, Cert2 ) @ #i2)) ∧
                   (Device_Sending_Message( Device1, BSM ) @ #i3)) ∧
                  (Certificate_Revocation_St_1( Device2, Cert2, Report ) @ #i4)) ∧
                 (Certificate_Revocation_St_2( Request ) @ #i5)) ∧
                (Certificate_Revocation_St_3( Request ) @ #i6)) ∧
               (Certificate_Revocation_St_4( Message ) @ #i7)) ∧
              (Certificate_Revocation_St_5( Message ) @ #i8)) ∧
             (#i1 < #i3)) ∧
            (#i1 < #i4)) ∧
           (#i3 < #i4)) ∧
          (#i4 < #i5)) ∧
         (#i5 < #i6)) ∧
        (#i6 < #i7)) ∧
       (#i7 < #i8)"
simplify
```

Figure 7.5: Certificate Revocation sanity checks proof

7.3 Security Properties Verification

The following sections present the models for the secrecy, authentication, and pseudonymity security properties.

7.3.1 Secrecy

The secrecy property ensures that confidential data or information are known only to authorized parties and that an attacker cannot obtain this information. The lemma used to prove the secrecy property is:

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

This lemma states that if `x` is secret, then there is no action `K(x)` (representing the adversary's knowledge) in any trace.

The `secret(x)` action is used in the `register_pk` rule (`Secret(~ltk)`) for the device's long-term key, and in the `device_pseudonym_certificate_request` rule (`Secret(~aa)`, `Secret(~hh)`) for the signing and encryption butterfly key seeds. As shown in Fig. 7.6, Tamarin successfully proved the lemma.

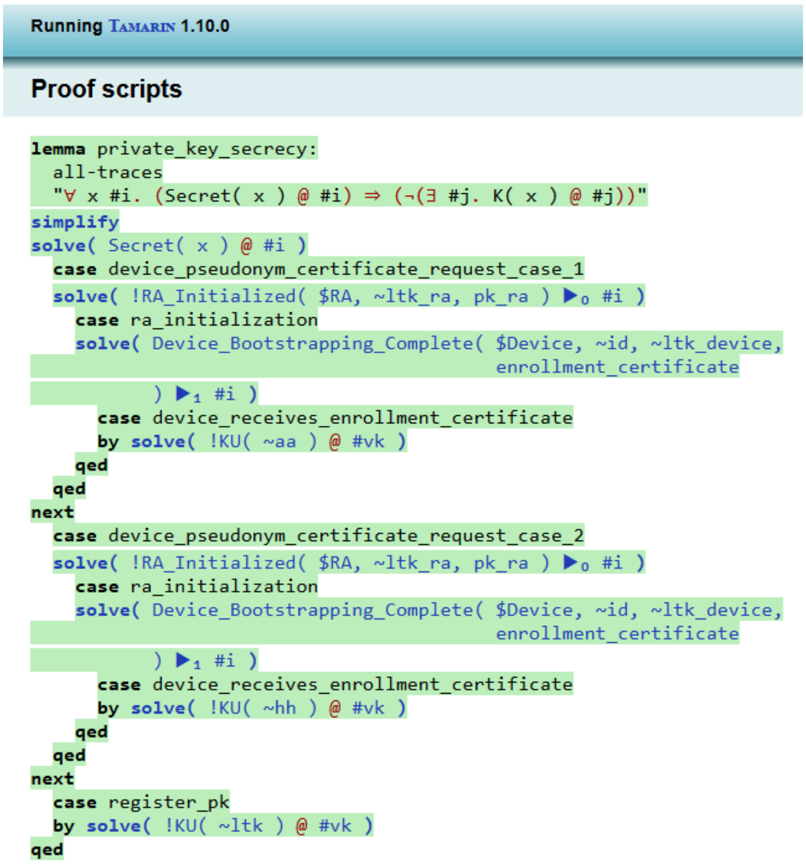


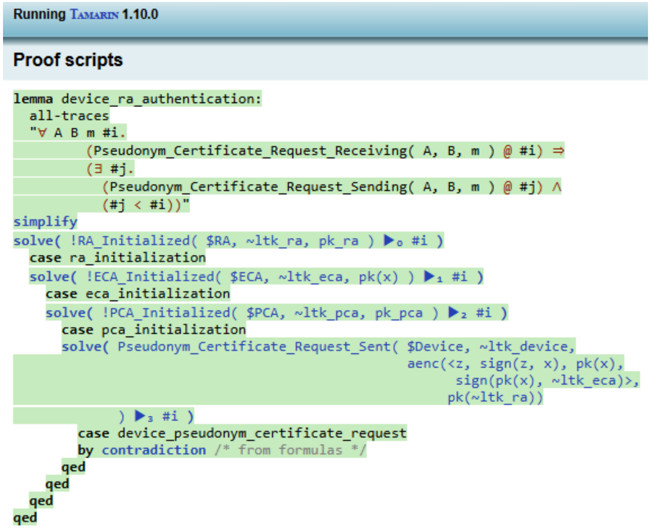
Figure 7.6: Secrecy property proof

7.3.2 Authentication

The authentication property ensures that messages are sent and received by the intended parties, preventing an attacker from forging a message that impersonates another entity. The lemmas used to prove this property are:

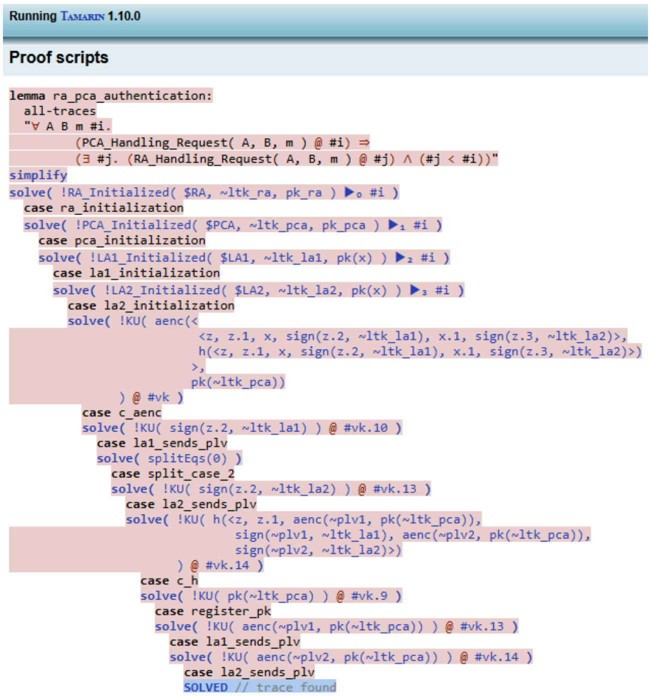
```
lemma device_ra_authentication:
"
  All A B m #i . Pseudonym_Certificate_Request_Receiving(A,B,m) @ #i ==>
    (Ex #j . Pseudonym_Certificate_Request_Sending(A,B,m) @ #j & #j < #i)
"

lemma ra_pca_authentication:
"
  All A B m #i . PCA_Handling_Request(A, B, m) @ #i ==>
    (Ex #j . RA_Handling_Request(A, B, m) @ #j & #j < #i)
"

lemma pca_device_authentication:
"
  All A m #i . Device_Receiving_Pseudonym_Certificate(A, m) @ #i ==>
    (Ex #j . PCA_Sending_Pseudonym_Certificate(A, m) @ #j & #j < #i)
"

lemma la1_pca_authentication:
"
  All A B m #i . Plv1_receiving(A, B, m) @ #i ==>
    (Ex #j . Plv1_Sending(A, B, m) @ #j & #j < #i)
"

lemma la2_pca_authentication:
"
  All A B m #i . Plv2_receiving(A, B, m) @ #i ==>
    (Ex #j . Plv2_Sending(A, B, m) @ #j & #j < #i)
"
lemma bsm_authentication:
"
  All A m #i1 . BSM_Receiving(A, m) @ #i1 ==>
    (Ex #j . BSM_Sending(A, m) @ #j & #j < #i1)
"
```

Each lemma targets the messages sent and received by the protocol parties, and requires that whenever a *Receiving* fact exists, there also exists a corresponding *Sending* fact with the same parameters, thereby ensuring message authenticity. Figures 7.7, 7.9, 7.10, 7.11, and 7.12 show that the authentication property is successfully proved. Figure 7.8 shows that the authentication property does not hold, because the messages between the RA and the PCA are not signed, allowing the attacker to send forged messages to the PCA, as illustrated in Fig. 7.13.



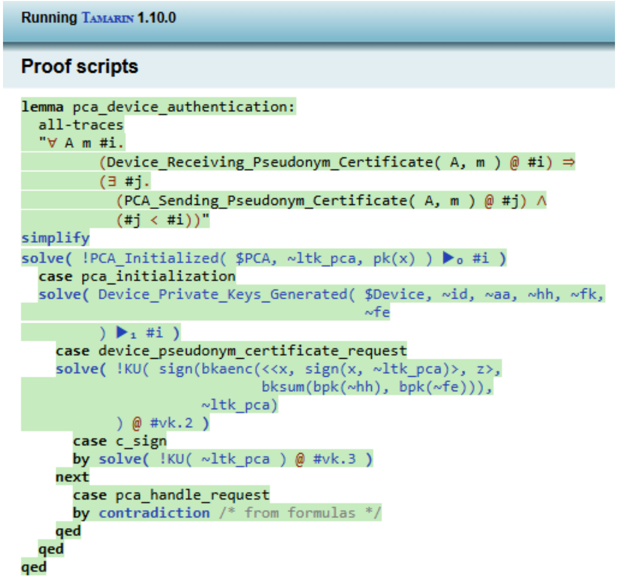
```
Running Tamarin 1.10.0
Proof scripts

lemma device_ra_authentication:
  all-traces
  "∀ A B m #i.
         (Pseudonym_Certificate_Request_Receiving( A, B, m ) @ #i) ⇒
         (∃ #j.
           (Pseudonym_Certificate_Request_Sending( A, B, m ) @ #j) ∧
           (#j < #i))"
simplify
solve( !RA_Initialized( $RA, ~ltk_ra, pk_ra ) ▶0 #i )
  case ra_initialization
  solve( !ECA_Initialized( $ECA, ~ltk_eca, pk(x) ) ▶1 #i )
    case eca_initialization
    solve( !PCA_Initialized( $PCA, ~ltk_pca, pk_pca ) ▶2 #i )
      case pca_initialization
      solve( Pseudonym_Certificate_Request_Sent( $Device, ~ltk_device,
                                                 aenc(<z, sign(z, x), pk(x),
                                                       sign(pk(x), ~ltk_eca)>,
                                                      pk(~ltk_ra))
             ) ▶3 #i )
        case device_pseudonym_certificate_request
        by contradiction /* from formulas */
      qed
    qed
  qed
qed
```

Figure 7.7: Device-RA authentication proof



```
Running Tamarin 1.10.0
Proof scripts

lemma ra_pca_authentication:
  all-traces
  "∀ A B m #i.
         (PCA_Handling_Request( A, B, m ) @ #i) ⇒
         (∃ #j. (RA_Handling_Request( A, B, m ) @ #j) ∧ (#j < #i))"
simplify
solve( !RA_Initialized( $RA, ~ltk_ra, pk_ra ) ▶0 #i )
  case ra_initialization
  solve( !PCA_Initialized( $PCA, ~ltk_pca, pk_pca ) ▶1 #i )
    case pca_initialization
    solve( !LA1_Initialized( $LA1, ~ltk_la1, pk(x) ) ▶2 #i )
      case la1_initialization
      solve( !LA2_Initialized( $LA2, ~ltk_la2, pk(x) ) ▶3 #i )
        case la2_initialization
        solve( !KU( aenc(<
                          <z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>,
                          h(<z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>)
                         >,
                         pk(~ltk_pca))
               ) @ #vk )
          case c_aenc
          solve( !KU( sign(z.2, ~ltk_la1) ) @ #vk.10 )
            case la1_sends_plv
            solve( splitEqs(0) )
              case split_case_2
              solve( !KU( sign(z.2, ~ltk_la2) ) @ #vk.13 )
                case la2_sends_plv
                solve( !KU( h(<z, z.1, aenc(~plv1, pk(~ltk_pca)),
                               sign(~plv1, ~ltk_la1), aenc(~plv2, pk(~ltk_pca)),
                               sign(~plv2, ~ltk_la2)>)
                       ) @ #vk.14 )
                  case c_h
                  solve( !KU( pk(~ltk_pca) ) @ #vk.9 )
                    case register_pk
                    solve( !KU( aenc(~plv1, pk(~ltk_pca)) ) @ #vk.13 )
                      case la1_sends_plv
                      solve( !KU( aenc(~plv2, pk(~ltk_pca)) ) @ #vk.14 )
                        case la2_sends_plv
                        SOLVED // trace found
```

Figure 7.8: RA-PCA authentication proof



Running Tamarin 1.10.0

Proof scripts

```
lemma pca_device_authentication:
  all-traces
  "∀ A m #i.
         (Device_Receiving_Pseudonym_Certificate( A, m ) @ #i) ⇒
         (∃ #j.
           (PCA_Sending_Pseudonym_Certificate( A, m ) @ #j) ∧
           (#j < #i))"
simplify
solve( !PCA_Initialized( $PCA, ~ltk_pca, pk(x) ) ▶₀ #i )
  case pca_initialization
  solve( Device_Private_Keys_Generated( $Device, ~id, ~aa, ~hh, ~fk,
                                        ~fe
         ) ▶₁ #i )
    case device_pseudonym_certificate_request
    solve( !KU( sign(bkaenc(<<x, sign(x, ~ltk_pca)>, z>,
                            bksum(bpk(~hh), bpk(~fe))),
                     ~ltk_pca)
           ) @ #vk.2 )
      case c_sign
      by solve( !KU( ~ltk_pca ) @ #vk.3 )
    next
      case pca_handle_request
      by contradiction /* from formulas */
    qed
  qed
qed
```

Figure 7.9: PCA-Device authentication proof

Running Tamarin 1.10.0

Proof scripts

```
lemma la1_pca_authentication:
  all-traces
  "∀ A B m #i.
         (Plv1_receiving( A, B, m ) @ #i) ⇒
         (∃ #j. (Plv1_Sending( A, B, m ) @ #j) ∧ (#j < #i))"
simplify
solve( !RA_Initialized( $RA, ~ltk_ra, pk_ra ) ▶₀ #i )
  case ra_initialization
  solve( !PCA_Initialized( $PCA, ~ltk_pca, pk_pca ) ▶₁ #i )
    case pca_initialization
    solve( !LA1_Initialized( $LA1, ~ltk_la1, pk(x) ) ▶₂ #i )
      case la1_initialization
      solve( !LA2_Initialized( $LA2, ~ltk_la2, pk(x) ) ▶₃ #i )
        case la2_initialization
        solve( !KU( aenc(<
                         <z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>,
                         h(<z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>)
                        >,
                        pk(~ltk_pca))
               ) @ #vk )
          case c_aenc
          solve( !KU( sign(z.2, ~ltk_la1) ) @ #vk.10 )
            case c_sign
            by solve( !KU( ~ltk_la1 ) @ #vk.16 )
          next
            case device_pseudonym_certificate_request
            by solve( !KU( ~ltk_ra ) @ #vk.15 )
          next
            case la1_sends_plv
            by contradiction /* from formulas */
          qed
        next
          case ra_handle_request
          solve( !KU( sign(z.2, ~ltk_la1) ) @ #vk.4 )
            case c_sign
            by solve( !KU( ~ltk_la1 ) @ #vk.9 )
          next
            case device_pseudonym_certificate_request
            by solve( !KU( ~ltk_ra ) @ #vk.8 )
          next
            case la1_sends_plv
            by contradiction /* from formulas */
          qed
        qed
      qed
    qed
  qed
qed
```

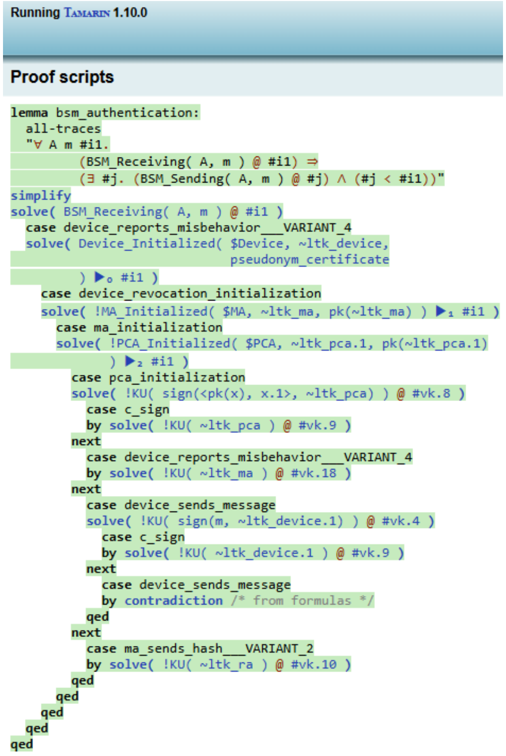
Figure 7.10: LA1-PCA authentication proof

Running Tamarin 1.10.0

Proof scripts

```
lemma la2_pca_authentication:
  all-traces
  "∀ A B m #i.
        (Plv2_receiving( A, B, m ) @ #i) ⇒
        (∃ #j. (Plv2_Sending( A, B, m ) @ #j) ∧ (#j < #i))"
simplify
solve( !RA_Initialized( $RA, ~ltk_ra, pk_ra ) ▶₀ #i )
  case ra_initialization
  solve( !PCA_Initialized( $PCA, ~ltk_pca, pk_pca ) ▶₁ #i )
    case pca_initialization
    solve( !LA1_Initialized( $LA1, ~ltk_la1, pk(x) ) ▶₂ #i )
      case la1_initialization
      solve( !LA2_Initialized( $LA2, ~ltk_la2, pk(x) ) ▶₃ #i )
        case la2_initialization
        solve( !KU( aenc(<
                          <z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>,
                          h(<z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>)
                         >,
                         pk(~ltk_pca))
               ) @ #vk )
          case c_aenc
          solve( !KU( sign(z.2, ~ltk_la1) ) @ #vk.10 )
            case c_sign
            by solve( !KU( ~ltk_la1 ) @ #vk.16 )
          next
            case device_pseudonym_certificate_request
            by solve( !KU( ~ltk_ra ) @ #vk.15 )
          next
            case la1_sends_plv
            solve( splitEqs(0) )
              case split_case_1
              solve( !KU( sign(adec(x, ~ltk_pca), ~ltk_la2) ) @ #vk.13 )
                case c_sign
                by solve( !KU( ~ltk_la2 ) @ #vk.16 )
              qed
            next
              case split_case_2
              solve( !KU( sign(z.2, ~ltk_la2) ) @ #vk.13 )
                case c_sign
                by solve( !KU( ~ltk_la2 ) @ #vk.16 )
              next
                case device_pseudonym_certificate_request
                by solve( !KU( ~ltk_ra ) @ #vk.15 )
              next
                case la2_sends_plv
                by contradiction /* from formulas */
              qed
            qed
          qed
        next
          case ra_handle_request
          solve( !KU( sign(z.2, ~ltk_la1) ) @ #vk.4 )
            case c_sign
            by solve( !KU( ~ltk_la1 ) @ #vk.9 )
          next
            case device_pseudonym_certificate_request
            by solve( !KU( ~ltk_ra ) @ #vk.8 )
          next
            case la1_sends_plv
            solve( splitEqs(0) )
              case split_case_1
              solve( !KU( sign(adec(x, ~ltk_pca), ~ltk_la2) ) @ #vk.7 )
                case c_sign
                by solve( !KU( ~ltk_la2 ) @ #vk.9 )
              qed
            next
              case split_case_2
              solve( !KU( sign(z.2, ~ltk_la2) ) @ #vk.7 )
                case c_sign
                by solve( !KU( ~ltk_la2 ) @ #vk.9 )
              next
                case device_pseudonym_certificate_request
                by solve( !KU( ~ltk_ra ) @ #vk.8 )
              next
                case la2_sends_plv
                by contradiction /* from formulas */
```

Figure 7.11: LA2-PCA authentication proof



Running Tamarin **1.10.0**

Proof scripts

```
lemma bsm_authentication:
  all-traces
  "∀ A m #i1.
       (BSM_Receiving( A, m ) @ #i1) ⇒
       (∃ #j. (BSM_Sending( A, m ) @ #j) ∧ (#j < #i1))"
simplify
solve( BSM_Receiving( A, m ) @ #i1 )
  case device_reports_misbehavior___VARIANT_4
  solve( Device_Initialized( $Device, ~ltk_device,
                             pseudonym_certificate
       ) ▶₀ #i1 )
    case device_revocation_initialization
    solve( !MA_Initialized( $MA, ~ltk_ma, pk(~ltk_ma) ) ▶₁ #i1 )
      case ma_initialization
      solve( !PCA_Initialized( $PCA, ~ltk_pca.1, pk(~ltk_pca.1)
             ) ▶₂ #i1 )
        case pca_initialization
        solve( !KU( sign(<pk(x), x.1>, ~ltk_pca) ) @ #vk.8 )
          case c_sign
          by solve( !KU( ~ltk_pca ) @ #vk.9 )
        next
          case device_reports_misbehavior___VARIANT_4
          by solve( !KU( ~ltk_ma ) @ #vk.18 )
        next
          case device_sends_message
          solve( !KU( sign(m, ~ltk_device.1) ) @ #vk.4 )
            case c_sign
            by solve( !KU( ~ltk_device.1 ) @ #vk.9 )
          next
            case device_sends_message
            by contradiction /* from formulas */
          qed
        next
          case ma_sends_hash___VARIANT_2
          by solve( !KU( ~ltk_ra ) @ #vk.10 )
        qed
      qed
    qed
  qed
qed
```

Figure 7.12: BSM authentication

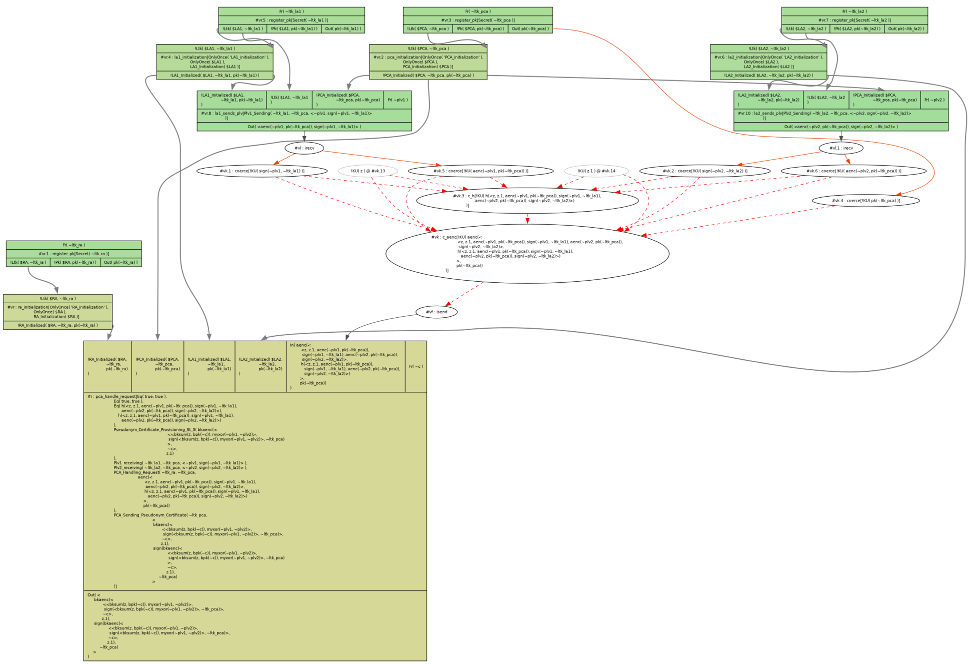


Figure 7.13: RA-PCA authentication attack

7.3.3 Compromise of Authorities

If an authority's long-term key is compromised, the authentication property may no longer hold because an attacker could forge a message and sign it with the compromised key.

If the PCA is compromised, the `pca_device_authentication` lemma cannot be proved, as shown in Fig. 7.14. Tamarin found an attack illustrated in Fig. 7.15.

If LA1 or LA2 is compromised, the `la1_pca_authentication` and the `la2_pca_authentication` lemmas cannot be proved as shown in Fig. 7.16 and Fig. 7.17. Tamarin found attacks illustrated in Fig. 7.18 and Fig. 7.19.

If the RA is compromised, there is no difference because the messages exchanged between the RA and the PCA are not signed by the RA.

Figure 7.14: PCA-Device authentication

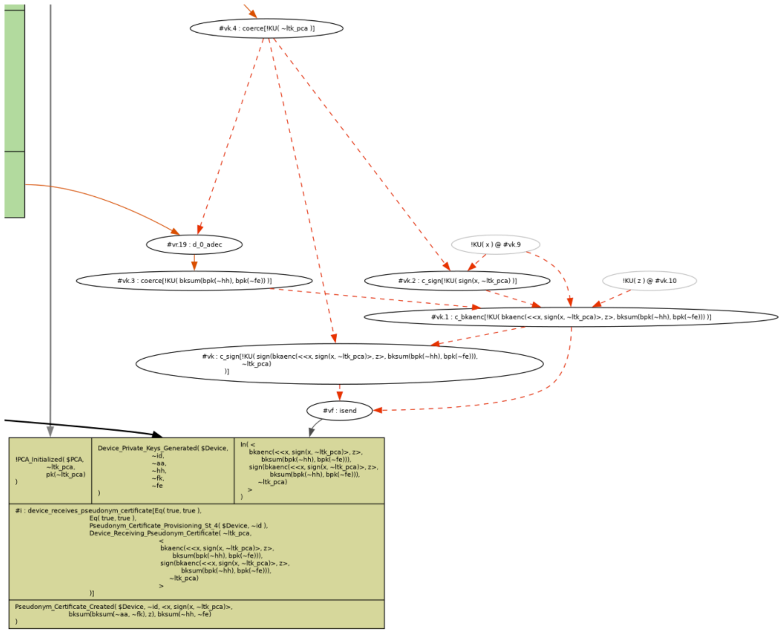


Figure 7.15: PCA-Device authentication attack

```
lemma la1_pca_authentication:
  all-traces
  "∀ A B m #i.
        (Plv1_receiving( A, B, m ) @ #i) ⇒
        (∃ #j. (Plv1_Sending( A, B, m ) @ #j) ∧ (#j < #i))"
simplify
solve( !RA_Initialized( $RA, ~ltk_ra, pk_ra ) ▶₀ #i )
  case ra_initialization
  solve( !PCA_Initialized( $PCA, ~ltk_pca, pk_pca ) ▶₁ #i )
    case pca_initialization
    solve( !LA1_Initialized( $LA1, ~ltk_la1, pk(x) ) ▶₂ #i )
      case la1_initialization
      solve( !LA2_Initialized( $LA2, ~ltk_la2, pk(x) ) ▶₃ #i )
        case la2_initialization
        solve( !KU( aenc(<
                         <z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>
                         h(<z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2
                        >,
                        pk(~ltk_pca))
                  ) @ #vk )
          case c_aenc
          solve( !KU( sign(z.2, ~ltk_la1) ) @ #vk.10 )
            case c_sign
            solve( !KU( sign(z.3, ~ltk_la2) ) @ #vk.13 )
              case la2_sends_plv
              solve( splitEqs(0) )
                case split_case_2
                solve( !KU( h(<z, z.1, aenc(z.2, pk(~ltk_pca)),
                               sign(z.2, ~ltk_la1), aenc(~plv2, pk(~ltk_pca)),
                               sign(~plv2, ~ltk_la2)>)
                      ) @ #vk.14 )
                  case c_h
                  solve( !KU( aenc(z.2, pk(~ltk_pca)) ) @ #vk.13 )
                    case c_aenc
                    solve( !KU( ~ltk_la1 ) @ #vk.16 )
                      case la1_initialization
                      solve( !KU( pk(~ltk_pca) ) @ #vk.12 )
                        case register_pk
                        solve( !KU( aenc(~plv2, pk(~ltk_pca)) ) @ #vk.15 )
                          case la2_sends_plv
                          SOLVED // trace found
                        qed
                      qed
                    qed
                  qed
                qed
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

Figure 7.16: LA1-PCA authentication

```
lemma la2_pca_authentication:
  all-traces
  "∀ A B m #i.
        (Plv2_receiving( A, B, m ) @ #i) ⇒
        (∃ #j. (Plv2_Sending( A, B, m ) @ #j) ∧ (#j < #i))"
simplify
solve( !RA_Initialized( $RA, ~ltk_ra, pk_ra ) ▶₀ #i )
  case ra_initialization
  solve( !PCA_Initialized( $PCA, ~ltk_pca, pk_pca ) ▶₁ #i )
    case pca_initialization
    solve( !LA1_Initialized( $LA1, ~ltk_la1, pk(x) ) ▶₂ #i )
      case la1_initialization
      solve( !LA2_Initialized( $LA2, ~ltk_la2, pk(x) ) ▶₃ #i )
        case la2_initialization
        solve( !KU( aenc(<
                         <z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>,
                         h(<z, z.1, x, sign(z.2, ~ltk_la1), x.1, sign(z.3, ~ltk_la2)>)
                        >,
                        pk(~ltk_pca))
                  ) @ #vk )
          case c_aenc
          solve( !KU( sign(z.2, ~ltk_la1) ) @ #vk.10 )
            case la1_sends_plv
            solve( splitEqs(0) )
              case split_case_2
              solve( !KU( sign(z.2, ~ltk_la2) ) @ #vk.13 )
                case c_sign
                solve( !KU( h(<z, z.1, aenc(~plv1, pk(~ltk_pca)),
                               sign(~plv1, ~ltk_la1), aenc(z.2, pk(~ltk_pca)), sign(z.2, ~ltk_la2)
                              >)
                      ) @ #vk.14 )
                  case c_h
                  solve( !KU( aenc(z.2, pk(~ltk_pca)) ) @ #vk.14 )
                    case c_aenc
                    solve( !KU( ~ltk_la2 ) @ #vk.16 )
                      case la2_initialization
                      solve( !KU( pk(~ltk_pca) ) @ #vk.12 )
                        case register_pk
                        solve( !KU( aenc(~plv1, pk(~ltk_pca)) ) @ #vk.15 )
                          case la1_sends_plv
                          SOLVED // trace found
                        qed
                      qed
                    qed
                  qed
                qed
              qed
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

Figure 7.17: LA2-PCA authentication

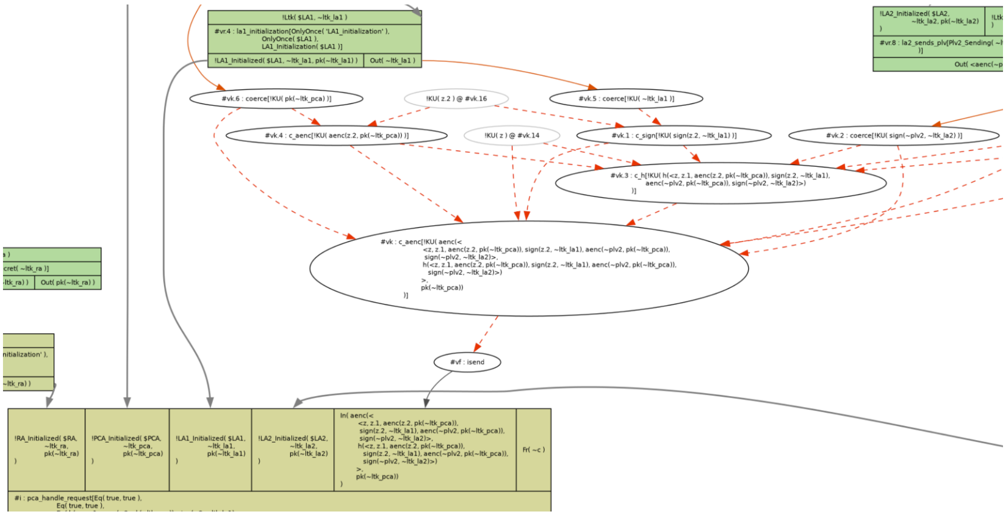


Figure 7.18: LA1-PCA authentication attack

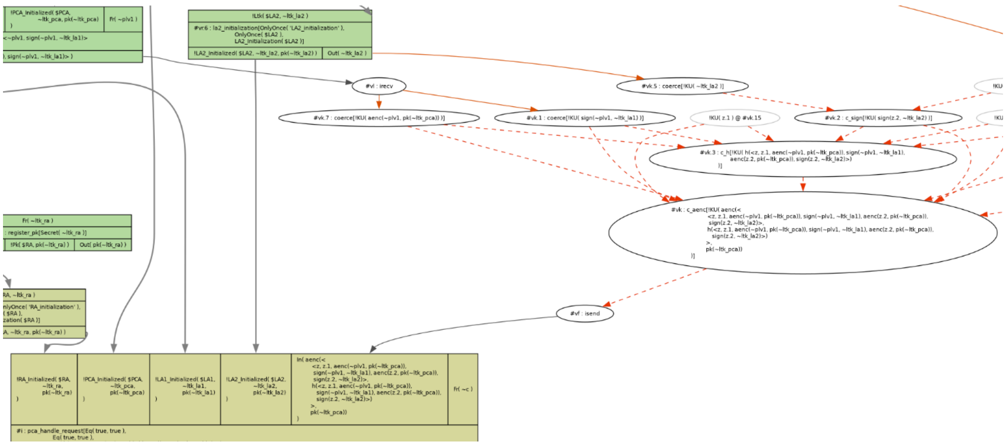


Figure 7.19: LA2-PCA authentication attack

7.3.4 Compromise of a Vehicle

If an attacker compromises a vehicle, security properties may be affected, therefore it is worthwhile to consider this case. This scenario is modelled with the following rule:

```
rule attacker_receives_enrollment_certificate:
    let
        enrollment_certificate = <pk(~ltk_device_attacker),
        sign(pk(~ltk_device_attacker), ~ltk_eca)>
    in
```

```
    [ !Ltk($Device_attacker, ~ltk_device_attacker),
      !ECA_Initialized($ECA, ~ltk_eca, pk_eca) ,
      Fr(~id)
    ]
  --[ ]->
    [ Device_Bootstrapping_Complete($Device, ~id, ~ltk_device_attacker,
      enrollment_certificate) ]
```

The rule models the state in which an attacker obtains control of a vehicle that possesses a valid enrollment certificate signed by the ECA. All authentication lemmas mentioned in the authentication section section have been successfully proved by Tamarin. Figure 7.20 shows that the attacker can request a pseudonym certificate as a legitimate user.

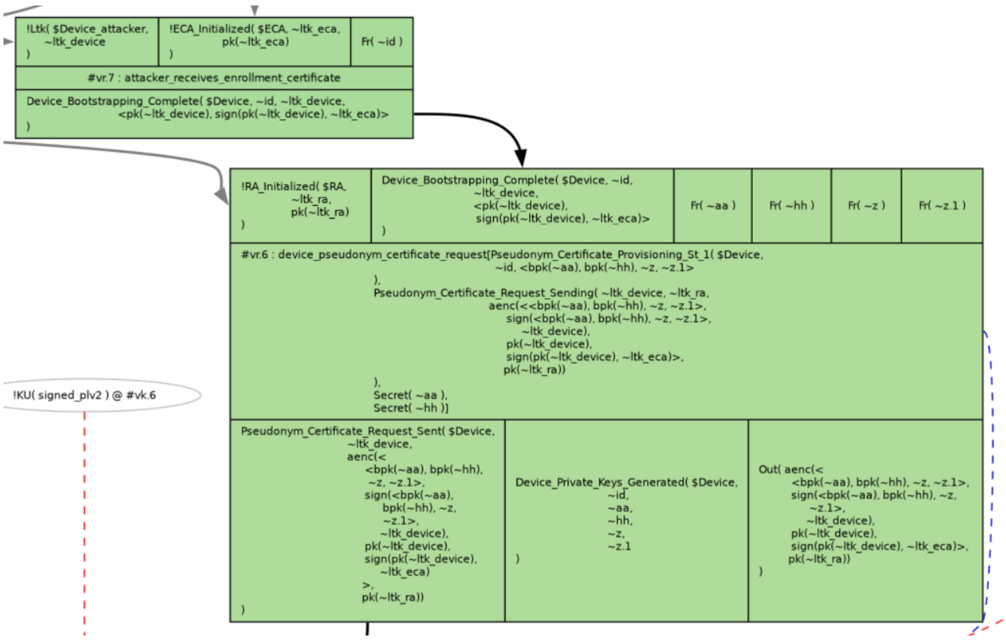


Figure 7.20: Attacker pseudonym certificate request

7.3.5 Pseudonymity

Another property that can be verified using the Tamarin prover is pseudonymity. This property ensures that the certificates used in the SCMS do not contain real-world identifiers, preventing users from bing tracked. To model this scenario, we used the observational equivalence in the following rule:

```
rule device_pseudonym_certificate_request:
    let
```

```
        AA = bpk(~aa)  // AA = aa * G
        HH = bpk(~hh)  // HH = hh * G
        request = <AA, HH, ~fk, ~fe>
        signed_request = sign(request, diff(~ltk_device, ~new_ltk))
        message = <request, signed_request, enrollment_certificate>
        encrypted_message = aenc(message, pk_ra)
    in
    [ !RA_Initialized($RA, ~ltk_ra, pk_ra),
      Device_Bootstrapping_Complete($Device, ~id, ~ltk_device,
      enrollment_certificate),
      Fr(~aa),  // signing private key seed
      Fr(~hh),  // encryption private key seed
      Fr(~fk),  // expansion function for signing keys
      Fr(~fe),   // expansion function for encryption keys
      Fr(~new_ltk)
    ]
  --[ Pseudonym_Certificate_Provisioning_St_1($Device, ~id, request),
      Pseudonym_Certificate_Request_Sending(~ltk_device, ~ltk_ra,
      encrypted_message),
      Secret(~aa),
      Secret(~hh)
    ]->
    [ Pseudonym_Certificate_Request_Sent($Device, ~ltk_device,
      encrypted_message),
      Device_Private_Keys_Generated($Device, ~id, ~aa, ~hh, ~fk, ~fe),
      Out(encrypted_message) ]
```

Unfortunately, the property could not be verified because Tamarin failed to execute the protocol due to insufficient memory on our available devices.

Chapter 8

Conclusions

This thesis examined the Security Credential Management System used in V2X communications through threat modelling and formal verification with the Tamarin prover.

We constructed detailed Tamarin models of the SCMS protocol steps and successfully proved secrecy and authentication properties for the exchanged messages. We could not complete a formal proof of pseudonymity, an essential property for protecting user privacy in V2X, because Tamarin exhausted the computational resources available to us as a result of the model's complexity.

The possible improvements to this work could be: reduce Tamarin model complexity by removing some details; allocate more computational resources to allow Tamarin to complete the verification of the pseudonymity property; explore alternative formal verification tool or techniques, that may be less resource intensive.

Bibliography

[1] Jonathan Petit, Florian Schaub, Michael Feiri, and Frank Kargl. «Pseudonym Schemes in Vehicular Networks: A Survey». In: *IEEE Communications Surveys & Tutorials* 17.1 (2015), pp. 228–255. DOI: 10.1109/COMST.2014.2345420 (cit. on p. 3).

[2] 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* 19.12 (2018), pp. 3850–3871. DOI: 10.1109/TITS.2018.2797529 (cit. on pp. 3, 14).

[3] David Basin. *Formal Methods for Security Knowledge Area Version 1.0.0*. 2021 (cit. on p. 6).

[4] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. *Modeling and Analyzing Security Protocols with Tamarin: A Comprehensive Guide*. 2025 (cit. on p. 8).

[5] *OWASP Threat Modeling Project*. https://owasp.org/www-community/Threat_Modeling. OWASP Foundation, Inc (cit. on p. 12).

[6] *Microsoft Threat Modeling Tool*. https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool. Microsoft (cit. on p. 12).

[7] *Tamarin-Prover Examples*. https://github.com/tamarin-prover/tamarin-prover/tree/develop/examples.