POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Formal verification of a cryptographic protocol in Cooperative Intelligent Transport Systems communications

Supervisors

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

Candidate

Francesco Galazzo

March 2024

Summary

Cooperative Intelligent Transport Systems (C-ITS) and Vehicle-to-Everything (V2X) communication stand as leading forces in transforming transportation. C-ITS facilitates the exchange of information between vehicles and traffic control systems via wireless networks, promoting both cooperative behavior and automated driving experiences. V2X is a broader term that includes various forms of vehicle communication, such as vehicle-to-vehicle (V2V), vehicle-to-infrastructure (V2I), vehicle-to-network (V2N), and vehicle-to-pedestrian (V2P) communications. These technologies are integral to enhancing road safety, traffic efficiency, and driving comfort. SCOOP@F, or "Système COopératif", is a pioneering C-ITS project in France that aims to improve road safety and traffic management by enabling V2V and V2I communication. This collaborative effort between road managers and car manufacturers tackles real-life challenges such as privacy, cybersecurity, and interoperability. The thesis in question examines the security mechanisms and the protocols used in SCOOP@F. In particular, it focuses on message exchanges between vehicles and Public Key Infrastructure (PKI) entities, as well as among the PKI entities themselves. Given the complexity and potential vulnerabilities of cryptographic protocols, which are crucial for protecting digital communications, the security mechanisms of the protocol require meticulous inspection. This scrutiny ensures the protection of message exchanges, thereby safeguarding user privacy and system integrity. Formal verification of cryptographic protocols, also used in C-ITS, is highlighted as a vital process to validate their resilience against potential vulnerabilities and to verify their security properties and robustness against message manipulation by attackers on the network. Proverif, a state-of-the-art tool for formal verification, is utilized to symbolically model and analyze the protocol. Proverif is capable of handling a wide range of cryptographic primitives and can process an unlimited number of protocol sessions simultaneously, allowing for extensive analysis and verification of protocols under various conditions and scenarios. The results of the formal verification, which are described in the thesis, show the traces produced by the tool. This verification process is essential to instill trust in the C-ITS landscape, where highly reliable communications and secure protocols are required.

Contents

List of Figures 7

Acronyms 9

1 Introduction 11
- 1.1 Thesis introduction . . . 11
- 1.2 Thesis description . . . 12

2 Overview of V2X 13
- 2.1 Intelligent Transport Systems . . . 13
  - 2.1.1 ITS architecture and PKI . . . 14
- 2.2 Cooperative Intelligent Transport Systems . . . 14
  - 2.2.1 ETSI Security standards for C-ITS . . . 16
  - 2.2.2 Security mechanisms and data structures . . . 16
  - 2.2.3 CCM Mode . . . 17
  - 2.2.4 ECIES . . . 18

3 SCOOP@F 22
- 3.1 SCOOP@F Project . . . 22
- 3.2 SCOOP@F PKI . . . 23
  - 3.2.1 PKI Architecture . . . 23
  - 3.2.2 Security objectives and the higher layers of the protocol stack 25
  - 3.2.3 Data structures . . . 26
- 3.3 General Overview of the PKI Protocol . . . 31
  - 3.3.1 LTC Request and Response . . . 32
  - 3.3.2 PC Request and Response . . . 32
  - 3.3.3 Keys . . . 34
  - 3.3.4 Technical Specifications . . . 34
- 3.4 Encryption of a message . . . 49

4 Formal verification of cryptographic protocols 51
4.1 Formal Verification Challenges 53
4.2 Proverif 55
4.2.1 How ProVerif Works 56
4.2.2 Specifying Protocols 56
4.2.3 Security Properties 57
4.2.4 Limitations 57

5 Thesis objective 59
5.1 Security Properties 60
5.2 Attack scenarios 60

6 Modelling of analysed protocol 61
6.1 Initial Procedures and Presumptions 62
6.2 Security mechanisms 62
6.2.1 Encryption 62
6.2.2 Elliptic Curves 63
6.2.3 Digital Signature 63
6.2.4 Hash Function 64
6.2.5 Certificates 64
6.2.6 ECIES 64
6.3 Protocol Analysis 65
6.3.1 Initialization Phase 65
6.3.2 Requests to the LTCA 66
6.3.3 Receipt of the Long Term Certificate 66
6.3.4 Request of a Pseudonym Certificate 67
6.3.5 Receipt of a Pseudonym Certificate 68
6.4 Privacy 69
6.4.1 Anonimity 69
6.4.2 Unlinkability 70

7 Results 71
7.1 Sanity Checks 71
7.1.1 Executability 71
7.2 Authentication 72
7.3 Privacy 73
7.3.1 Anonimity 73
7.3.2 Unlinkability 73

8 Conclusions 77
8.1 Future work 77

Bibliography 79

A Proverif Code 82

A.1 LTC Request and Response 82

A.2 PC Request and Response 93

List of Figures

2.1 ETSI TS 102 940 . . . 15
2.2 European C-ITS Protocol Stack (taken from [2]) . . . 15
2.3 CCM encryption schema . . . 17
2.4 CCM decryption schema . . . 18
2.5 ECIES encryption functional diagram [9] . . . 19
2.6 ECIES decryption functional diagram [9] . . . 20
2.7 ECIES ETSI schema . . . 20

3.1 SCOOP PKI Architecture . . . 23
3.2 Higher-layer supported PKI protocols . . . 26
3.3 Format of messages designed for the secure transmission of LTC and PC requests. [13] . . . 27
3.4 LTC request and response [13, 16] . . . 33
3.5 PC request and response [13, 16] . . . 34
3.6 PC request and response [13] . . . 35
3.7 LTC request [13] . . . 36
3.8 Structure of a Pseudonym Certificate request . . . 42
3.9 Pseudonym Certificate request [13] . . . 43
3.10 The structure of PC validation request . . . 45

4.1 Protocol's formal verification procedure overview. [19] . . . 52
4.2 ProVerif internal [20] . . . 56

7.1 Long Term Certificate request performed by the attacker . . . 74
7.2 PC request performed by the attacker . . . 75
7.3 Validation PC request generated by the attacker . . . 76

Acronyms

ITS

Intelligent Transport System

ITS-S

ITS Station

C-ITS

Cooperative ITS

PKI

Public Key Infrastructure

RCA

Root Certificate Authority

PCA

Pseudonym Certification Authority

LTCA

Long Term Certificate Authority

LTC

Long Term Certificate

PC

Pseudonym Certificate

CRL

Certificate Revocation List

TSL

Trust-service Status List

Chapter 1

Introduction

1.1 Thesis introduction

In this thesis, we explore the fascinating world of Cooperative Intelligent Transport Systems (C-ITS), a rapidly evolving field that is reshaping the landscape of transportation and communication. As we navigate through the complexities of C-ITS, we focus on the security of communications, a critical aspect that underpins the reliability and trustworthiness of these systems. C-ITS represents a paradigm shift in how vehicles and infrastructure interact, promising to enhance safety, efficiency, and sustainability in our transportation networks. However, as these systems become increasingly interconnected, they also become more vulnerable to potential security threats. Therefore, ensuring the robustness and integrity of communication protocols within C-ITS is of paramount importance. Our focus in this thesis is the Public Key Infrastructure (PKI) protocol within C-ITS, a cornerstone for secure and trusted communication. We will conduct an in-depth analysis of this protocol, exploring its design, functionality, and potential vulnerabilities. To ensure the security and reliability of the PKI protocol, we employ formal verification, a rigorous method of analysis that allows us to systematically evaluate the protocol. Specifically, we focus on the SCOOP's PKI protocol, a pioneering implementation within the realm of C-ITS. Using *ProVerif*, a powerful tool for formal verification, we dissect the protocol, examining its every facet to identify potential vulnerabilities and areas for improvement. Through the course of this thesis, we aim to provide a comprehensive understanding of C-ITS, the security of its communications, and the importance of formal verification. Our goal is to contribute to the ongoing efforts to make our transportation systems safer, more efficient, and more secure.

1.2 Thesis description

The subsequent section of this thesis is organized as outlined below.

- **Chapter 2:** provides a comprehensive overview and context of Cooperative Intelligent Transport Systems (C-ITS) and Vehicle-to-Everything (V2X) communications, specifically highlighting the security mechanisms as defined in the relevant ETSI standards.
- **Chapter 3:** offers a thorough analysis of the protocol for formal verification, including all details related to the security mechanisms, data structures, and messages.
- **Chapter 4:** introduces formal verification techniques and *ProVerif*, the tool that will be utilized.
- **Chapter 5:** outlines the thesis's objectives, the security properties to be verified, and the attack scenarios.
- **Chapter 6:** offers a comprehensive analysis of the design choices and security mechanisms, as well as the resulting model, which is then examined using *ProVerif*.
- **Chapter 7:** presents the outcomes of the automated formal verification performed on the models outlined in Chapter 6.
- **Chapter 8:** outlines the final conclusions drawn from this thesis and explores potential avenues for future research that could enhance and build upon the current work.

Chapter 2

Overview of V2X

2.1 Intelligent Transport Systems

As urban areas evolve into smart cities, there is a continuous exploration of innovative technologies to better manage these communities. Intelligent transportation systems have become an integral part of every burgeoning smart city due to their versatility and potential. These systems go beyond basic traffic management, enabling smarter transportation networks that offer improved efficiency, safety, and the ability to tackle persistent urban mobility challenges such as congestion. With endless possibilities for implementation, intelligent transportation systems provide smart cities with a more advanced approach to traditional transit infrastructure and management. Their intelligent design allows for real-time data collection, analysis, and response, facilitating dynamic and responsive transportation ecosystems. As smart cities continue to develop, intelligent transportation systems will likely serve as the backbone of next-generation urban mobility, leveraging connectivity, automation, and intelligence to revolutionize how people and goods traverse smarter built environments. Intelligent transportation systems encompass the integration of information and communication technologies within transportation infrastructure and vehicles, with the aim of enhancing the efficiency, safety, and environmental sustainability of transport networks. These systems are evolving swiftly, with the incorporation of Internet of Things (IoT) technologies playing a significant role in the advancement of intelligent transport solutions. ITS are increasingly becoming a cornerstone in the management of transportation and traffic, contributing to safer, more efficient, and sustainable travel. Intelligent transportation systems are instrumental in creating smarter journeys and improving access for emergency services, contributing to more sustainable transport. The development of intelligent transportation systems is crucial in the context of smart city evolution, aiming to significantly enhance transportation system efficiency, reduce energy consumption, lower transportation costs, and minimize environmental impact. Intelligent traffic control is key to addressing traffic congestion and pollution issues. Current intelligent transportation systems technologies primarily involve information technology, data communication technology, and sensors. Intelligent transportation systems are being implemented worldwide to increase the capacity of congested roads, reduce travel times, and gather information on road users. These systems are designed to integrate various functionalities, including electronic fare collection, road safety enhancement, and the provision of information to travelers. Intelligent Transportation Systems (STS) are pivotal in bolstering safety and tackling the growing challenges of emissions and traffic jams across Europe. By integrating various information and communication technologies into transportation modes for both people and goods, they enhance safety, increase operational efficiency, and foster environmental sustainability. Additionally, the fusion of existing technologies paves the way for novel service innovations. ITS are vital in stimulating job creation and economic expansion within the transport sector. Nonetheless, to maximize their effectiveness, the implementation of ITS must be systematic and well-synchronized across the European Union [1].

2.1.1 ITS architecture and PKI

The ETSI EN 302 665 outlines an architecture for Intelligent Transport Systems (ITS) stations, structured around four distinct processing layers: the Access Layer, Networking and Transport Layer, Facilities Layer, and Applications Layer. These layers are complemented by two vertical components: a Management entity and a Security entity, as depicted in Figure 2.1.

2.2 Cooperative Intelligent Transport Systems (C-ITS)

Cooperative Intelligent Transport Systems (C-ITS) are a specialized category within Intelligent Transport Systems that utilize wireless telecommunication technologies for real-time communication and information exchange between vehicles, and between vehicles and both central and roadside infrastructure. C-ITS focuses on vehicle-to-vehicle (V2V), vehicle-to-infrastructure (V2I), and vehicle-to-everything (V2X) communications. In Europe, C-ITS operations are regulated by a framework of standards ensuring interoperability, safety, and efficiency in vehicular communication systems. These standards are primarily developed by the European Telecommunications Standards Institute (ETSI), with contributions from other standardization bodies such as the International Organization for Standardization (ISO) and the European Committee for Standardization (CEN). The European

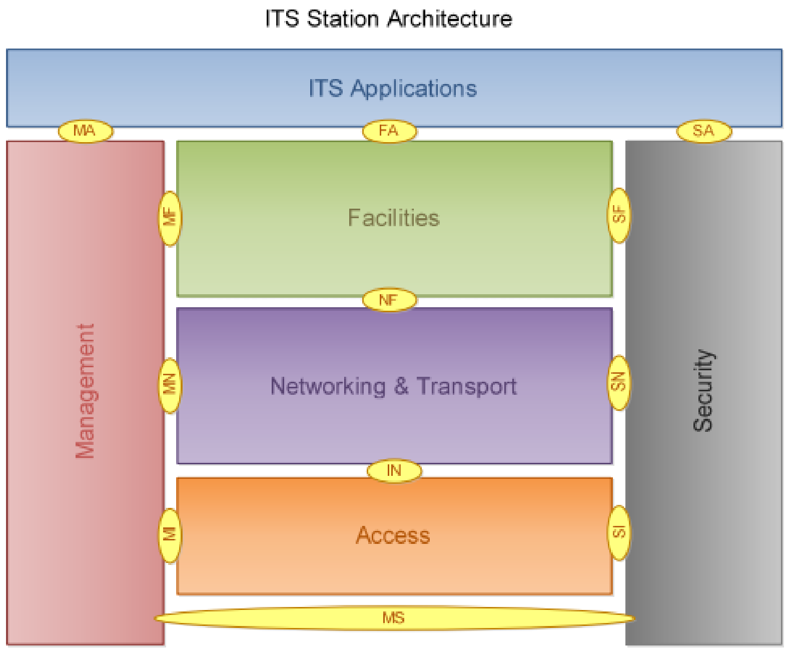


Figure 2.1: ETSI TS 102 940

C-ITS Protocol Stack is illustrated in the figure 2.2.

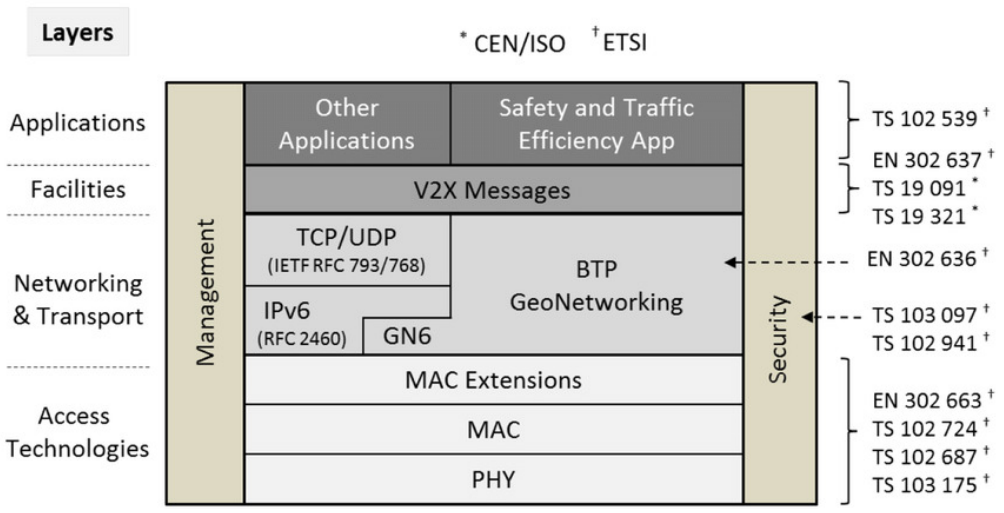


Figure 2.2: European C-ITS Protocol Stack (taken from [2])

2.2.1 ETSI Security standards for C-ITS

The European Telecommunications Standards Institute has developed a comprehensive series of standards dedicated to enhancing the security of Cooperative Intelligent Transport Systems. These standards encompass a wide range of security aspects, including encryption, authentication, privacy, and the integrity of communications. Key ETSI standards pertinent to C-ITS and Vehicle-to-Everything security include:

- **ETSI TS 102 731** [3]: details the security services and architecture for ITS communications.
- **ETSI TS 103 097** [4]: outlines the security architecture for ITS, including security headers, certificate formats, security profiles, and cryptographic algorithms.
- **ETSI TS 102 940** [5]: describes the security architecture and security management for V2X communications.
- **ETSI TS 102 941** [6]: focuses on trust and privacy management within ITS, detailing procedures for certificate and digital signature management.
- **ETSI TS 102 942** [7]: specifies the technical requirements for access control within ITS for V2X communications.
- **ETSI TS 102 943** [8]: addresses confidentiality services in ITS for V2X communications, emphasizing the protection of sensitive information to prevent unauthorized access and ensure data privacy.

2.2.2 Security mechanisms and data structures

The ETSI TS 103 097 [4] standard specifies various security mechanisms for ITS, leveraging elliptic curves and secure data structures. The cryptographic algorithms employed include:

- **Symmetric Encryption Algorithm**: AES-128-CCM with a 128-bit key size.
- **Signature Algorithm**: Based on ECDSA_nistP256_with_SHA256, a variant of the Digital Signature Algorithm (DSA) utilizing the NIST P-256 curve and SHA-256 for hashing.
- **Elliptic Curve Integrated Encryption Scheme (ECIES)**.

2.2.3 CCM Mode

The Counter with Cipher Block Chaining-Message Authentication Code (CCM) represents a cryptographic block cipher mode. This authenticated encryption technique is crafted to deliver authentication along with data privacy. Encryption is executed using the counter (CTR) mode, while authentication is achieved through Cipher Block Chaining-Message Authentication Code (CBC-MAC). As depicted in the Figure 2.3, the process begins by calculating the CBC-MAC on the message to generate a MAC (tag), which is then, along with the message, encrypted utilizing the counter mode. It has three inputs:

1. The message to be encrypted.
2. The encryption key K'.
3. A nonce (a random value used only once).

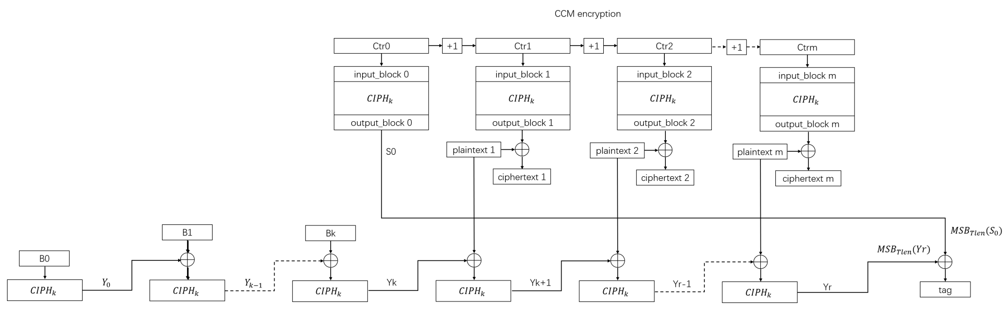


Figure 2.3: CCM encryption schema

The CCM encryption process can be summarized as follows:

1. The CBC-MAC is calculated using a structured input B, which is created from the nonce N, any associated data A, and the plaintext message M. This structured input B is composed of an initial block B0, succeeded by the blocks containing associated data and plaintext.
2. The final segment of the CBC-MAC output is cut down to the required tag size, resulting in the creation of the MAC tag T.
3. The message M along with the MAC tag T are merged and encrypted in counter (CTR) mode utilizing the encryption key K' and counter blocks A0, A1, ... which are generated from the nonce N. In detail:
   - The tag T undergoes encryption to become C0 $= lsbt(ENC_{K'}(A0)) \oplus T$
   - Every block of plaintext M_i is encrypted to Ci $= ENC_{K'}(Ci) \oplus M_i$

4. The resulting encrypted message C, comprising C0 || C1 || ... || Cm, along with the related associated data A, is dispatched to the recipient. The nonce N is also sent along if it's deemed necessary.

The recipient is then able to decrypt the message and check the Message Authentication Code (MAC) to confirm that the message remains unaltered and that it originated from the expected sender.

The CCM decryption mode shown in Figure 2.4 has the same inputs as the encryption mode. The ciphertext is first decrypted, and a plaintext is obtained. This plaintext is then used to compute a MAC with the CBC-MAC, which is compared with the MAC received. If the two MACs are equal, then the plaintext is authenticated.

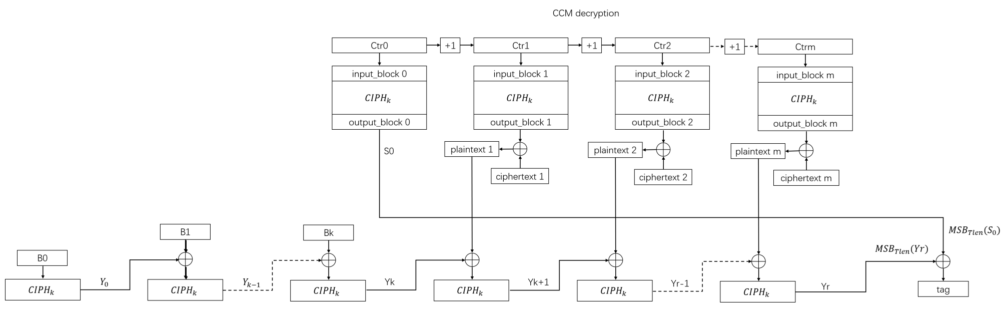


Figure 2.4: CCM decryption schema

2.2.4 ECIES

ECIES is an asymmetric encryption scheme defined in IEEE Std 1609.2 and in ETSI standard that is used to transport symmetric encryption keys. It is a hybrid encryption system that merges two distinct encapsulation mechanisms to provide a secure encryption process. Specifically, it integrates a Key Encapsulation Mechanism (KEM) with a Data Encapsulation Mechanism (DEM) to ensure both confidentiality and integrity of the data. Key Encapsulation Mechanism (KEM): this component of ECIES is responsible for generating a pair of keys (a public key and a private key that are ephemeral) based on elliptic curve cryptography. The public key may be freely distributed, whereas the private key is maintained in confidentiality. KEM is used to encapsulate or "wrap" a symmetric encryption key, which is then securely transmitted to the recipient with the public key generated by KEM. Data Encapsulation Mechanism (DEM): once the symmetric key is encapsulated and sent to the recipient, the DEM comes into play. It uses the symmetric key to encrypt the actual data or message. This mechanism typically employs a symmetric encryption algorithm like AES (Advanced Encryption Standard) to ensure the data is encrypted efficiently and securely. The combination of KEM and DEM in ECIES allows for the secure transmission of encrypted data. The symmetric key is encrypted using the private key provided by the Key Encapsulation Mechanism (KEM), and this symmetric key is subsequently utilized by the Data Encapsulation Mechanism (DEM) to encrypt the message. The recipient, who possesses the corresponding private key and the public key sent with the KEM, can decrypt the symmetric key and, subsequently, the message itself. Figures 2.5 and 2.6 depict the ECIES diagrams for encrypting and decrypting a plaintext message, denoted as 'm'. The ECIES ETSI schema is represented in Figure 2.7. The ECIES algorithms

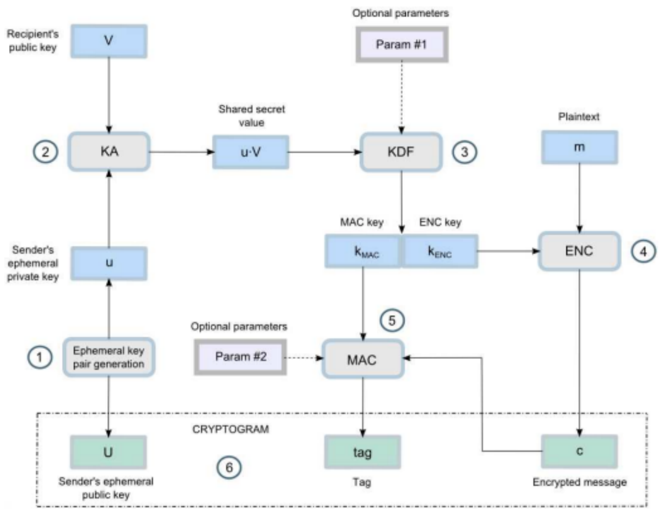


Figure 2.5: ECIES encryption functional diagram [9]

and parameters defined in the ETSI standard are as follows:

- **Key Agreement**: Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version with cofactor multiplication (ECSVDP-DHC). It is a cryptographic primitive used in key agreement protocols and is based on the Diffie-Hellman key exchange mechanism, but it is adapted for use with elliptic curve cryptography. In a key agreement protocol using ECSVDP-DHC, each party generates an ephemeral private key and computes the corresponding public key as a point on the elliptic curve. The parties then exchange their public keys and use their own private keys along with the received public key to compute the shared secret.
- **P1** and **P2**: are designated as empty strings.
- **Key Derivation Function**: a Key Derivation Function (KDF) is a cryptographic method that generates one or more secret keys from a private

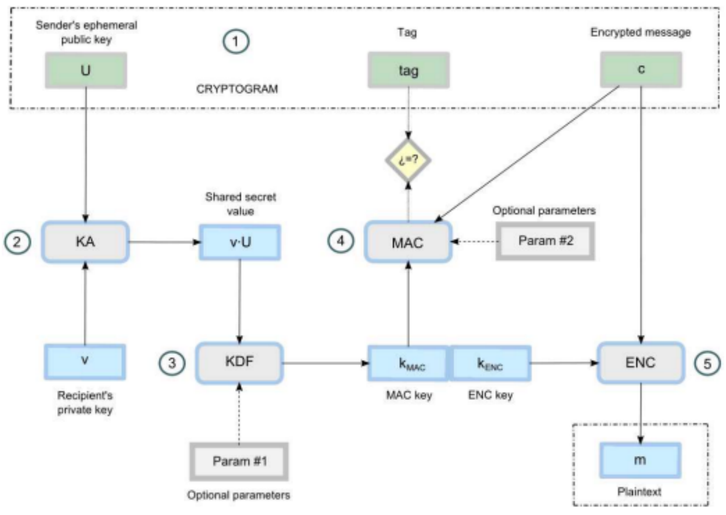


Figure 2.6: ECIES decryption functional diagram [9]

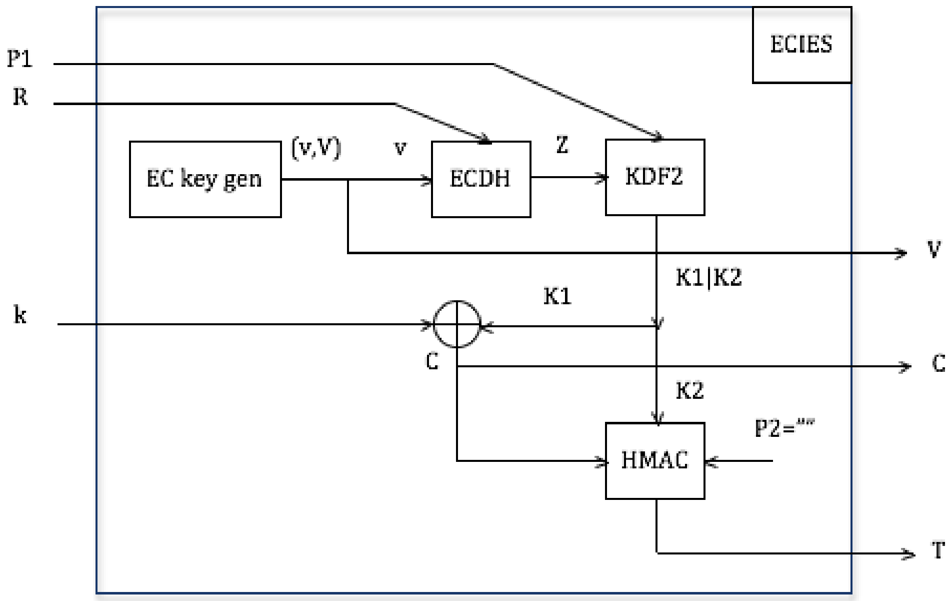


Figure 2.7: ECIES ETSI schema

value, such as a master key, password, or passphrase. KDFs find application in various scenarios, encompassing key derivation from secret passwords or passphrases, generating keys of varying lengths, and as integral components of multiparty key-agreement protocols. KDFs serve the purpose of extending keys into longer formats or obtaining keys with specific criteria. In this context, KDF2 is employed, specifically utilizing SHA-256. To elaborate, KDF2(Z, P1) = SHA-256(Z || Counter || P1), where the counter is 32-bit and Z represents a point on the elliptic curve.

- **Encryption**: XOR. The XOR operation, also known as exclusive OR, is a fundamental operation in cryptography, frequently employed in stream ciphers and various cryptographic protocols. The XOR operation is both associative and commutative, which means that the sequence and grouping of operands do not influence the end result. Mathematically, this property is expressed as:
  - $(a \oplus b)$ $\oplus$ c = a $\oplus$ $(b \oplus c)$ associativity
  - $a \oplus b = b \oplus a$ commutativity
- **MAC**: MAC1 with SHA-256. The MAC1 with SHA-256 is computed as HMAC(K2, C). HMAC or Hash-based Message Authentication Code, is a cryptographic technique used for data integrity and authentication. It combines the benefits of cryptographic hash functions and a shared secret key, making it more secure than many other authentication methods. HMAC is designed to be one-way, meaning it's easy to generate output from input but complex to do the reverse. It's also designed to be less affected by collisions than hash functions.

  HMAC(K2, C) = Hash($(K2 \oplus iPad)$ || Hash($(K2 \oplus oPad)$ || C))

  where:
  - Hash is the SHA-256 algorithm.
  - iPad and oPad are 256-bit (32-byte) blocks formed by repeating the byte 0x36 and 0x5C.
  - K2 is the key derived from KDF2.
  - C is the ciphertex.

Chapter 3

SCOOP@F

3.1 SCOOP@F Project

SCOOP project, also known as SCOOP@F, is a French pilot project focused on the deployment of Cooperative-Intelligent Transport Systems (C-ITS). The project utilizes information and communication technologies in the field of transport, with a cooperative approach based on the exchange of information between vehicles and between vehicles and infrastructure, also known as V2X communication. The project was divided into two parts: SCOOP@F Part 1 (2014-2015) and SCOOP@F Part 2 (2016-2018). The first part tested and validated the C-ITS use cases using ITS G5 communication technology, while the second part focused on a hybrid communication technology, ITS G5 and cellular, and the development of new C-ITS use cases. The project aims to enhance road safety, optimize traffic information, develop new services, and prepare the vehicles of tomorrow. It is considered a solution to make automated vehicles cope with critical situations they could not cope with otherwise, such as toll gates and road works, and to anticipate sensor detection for better driver comfort. SCOOP uses ITS G5 technology, a WiFi technology adapted to high-speed vehicles, operating in the 5.9 GHz band. This technology allows V2X exchanges with very low latency, which is crucial for road safety use cases. The communication with infrastructure is done through Road Side Units. The project does not involve any automation, and messages are received by the driver. The project involves large-scale deployment (3000 vehicles on 2000 km of roads), in real conditions, with real-life constraints. The vehicles are sold to real customers and are designed with the Commission Nationale de l'Informatique et des Libertés (CNIL) and the Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI). The project is funded 50% by the European Commission, and it includes ex ante and ex post evaluation. The project partners include the French Ministry of Transport, local authorities, TEN-T road operators, car manufacturers, universities and research centers, a telecommunication operator, and a provider of trust services [10].

3.2 SCOOP@F PKI

A primary objective of this project is the implementation of a dedicated Intelligent Transport Systems (ITS) Public Key Infrastructure (PKI). The proposed PKI is designed in accordance with the standards set by the European Telecommunications Standards Institute (ETSI).

3.2.1 PKI Architecture

The architecture of the SCOOP@F PKI is depicted in Figure 3.1.

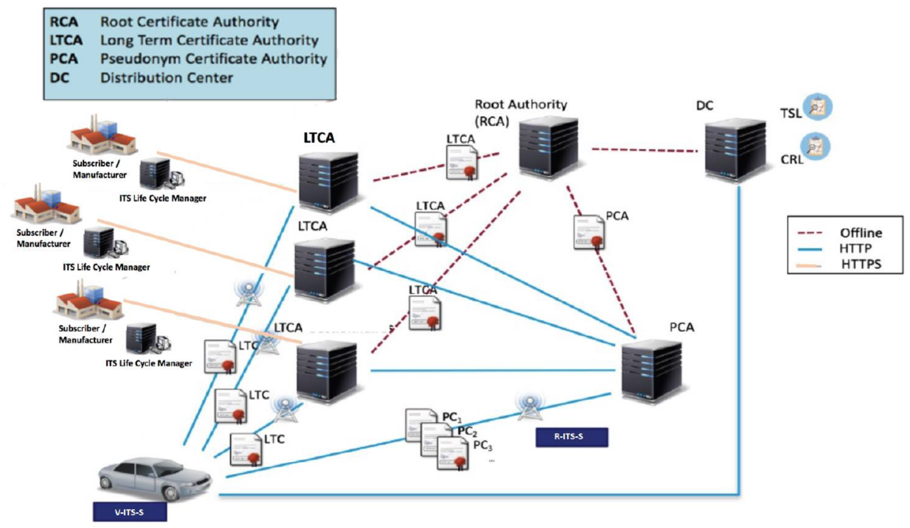

RCA Root Certificate Authority
LTCA Long Term Certificate Authority
PCA Pseudonym Certificate Authority
DC Distribution Center
Root Authority (RCA)
DC
TSL
CRL
LTCA
PCA
Subscriber / Manufacturer
ITS Life Cycle Manager
LTC
PC1
PC2
PC3
R-ITS-S
V-ITS-S
Offline
HTTP
HTTPS

Figure 3.1: SCOOP PKI Architecture

The entities involved in this PKI are:

- **Root Certificate Authority (RCA)**: an RCA, or Root Certification Authority, is distinguished by possessing a self-signed certificate, where the issuer and the signer are identical. Unlike other certificates, an RCA's certificate cannot be revoked through conventional means, such as inclusion in a Certificate Revocation List (CRL). RCA is always used offline and never connected to any network. The RCA certificate is self-signed. RCA never receives a certificate from another CA (never certified or cross-certified with another external CA). The RCA facilitates the provision of the following PKI functionalities:

  - Creation of a Root CA key pair and its own self-issued certificate.
  - Generation of CA key pairs and certificates.
  - Signature of TSL and CRL.
  - Revocation of CA certificates.
  - Update the CRL/TSL
  - Rekey of CA certificates.
  - Log trail generation.

- **Long Term Certification Authority (LTCA)**: serves as a security management entity tasked with issuing Long Term Certificates (LTCs) and validating Pseudonym Certificates (PCs). Additionally, it oversees the management of Intelligent Transport Systems Stations, encompassing registration, status updates, and permissions. The LTCA functions in an online capacity[11]. LTCA has the responsibility to:
  - Manage C-ITS-S status.
  - Sign LTC certificate.
  - Authenticate PCA using TSL signed by RCA that has signed LTCA.
  - Transmit right public keys to be certified by RCA.
  - Transmit right public keys to be certified by RCA.
  - Communicates only with PCA authenticated with RCA validation information communicated by DC.
  - Respond to request from Policy Management Authority (PMA).
  - Establishes contract with C-ITS-S Manufacturer.
  - Manage PCA validation request for PC certificate request.

  LTCA can only issue LTC and validate requests sent by a PCA for PC certificate request from a C-ITS-S produced by an C-ITS-S Manufacturer and for which the C-ITS-S is already registered by this LTCA.

- **Pseudonym Certificate Authority (PCA)**: is a security management entity tasked with the issuance, supervision, and utilization of Pseudonym Certificates (PCs). It functions in an online mode. PCA can only issue PCs for C-ITS-S which have been authorized.
  PCA is responsible for:
  - Managing C-ITS-S request.
  - Signing PC certificate.

  - Authenticating LTCA using TSL issued by RCA.
  - Managing communication with LTCA and DC.
  - Submit accurate and complete information to the LTCA.
  - Transmit right public keys to be certified by RCA.

- **Distribution Centre (DC)**: is responsible for supplying the Intelligent Transport Systems Stations with the latest trust information, including Trust Service Lists (TSL) and Certificate Revocation Lists (CRL). This information is crucial to ensure that the received data originates from legitimate and authorized ITS-Ss or a Public Key Infrastructure (PKI) certification authority. The DC provides the following PKI services:
  - Publication service.
  - Log trail generation.

3.2.2 Security objectives and the higher layers of the protocol stack

The protocol used in SCOOP@F aims to achieve the following security objectives [12]:

- **Authentication/Authorization Control**: Authentication involves verifying the identity of the data sender. Authorization control, on the other hand, is the process of checking an access policy based on trusted authentication. It is crucial to authenticate all entities participating in the protocol to prevent unauthorized individuals from accessing the system or certain restricted resources or services.
- **Data Integrity**: Ensuring the integrity of all transmitted data is vital to confirm that the content of the received data remains unaltered.
- **Confidentiality/Privacy**: Data should be accessible only to authorized entities. The real identity of the ITS Station must be safeguarded through cryptographic mechanisms, depending on the type of data transmitted.
- **Non-repudiation/Traceability**: Ensuring non-repudiation is crucial to stop the ITS Station or any involved parties from rejecting the sending or the content of their communications. Equally critical is traceability, which secures that an entity can not refute the sending or receiving of data.
- **Unlinkability**: This refers to the user's ability to use resources or services multiple times without others being able to associate these uses together.

- **Anonymity**: This is the user's ability to utilize a resource or service without revealing their identity.

The technical specification specifies [12], the higher layers of the protocol stack, Figure 3.2, and assumes either a fixed or cellular network with the ITS-S or an ITS G5 communication profile supporting IP connectivity.

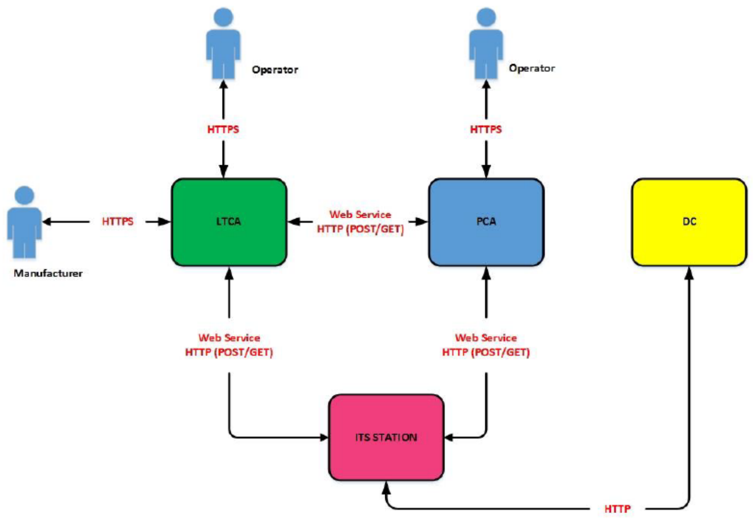


Figure 3.2: Higher-layer supported PKI protocols

Machine-to-machine communications with the LTCA, PCA, and DC components use HTTP/1.1 as a transport mechanism over TCP/IP. No supplementary cryptographic layer such as TLS is required. Messages are sent as HTTP GET or POST requests.

3.2.3 Data structures

In the SCOOP project, the protocol messages exchanged between Intelligent Transportation System Stations (ITS-S) and Public Key Infrastructure (PKI) entities, as well as among the PKI entities themselves, utilize specific data structures—*Data*, *SignedData*, and *EncryptedData* —along with associated algorithm identifiers. These are precisely defined to provide clear security properties and they have been meticulously defined using the Abstract Syntax Notation One (ASN.1) and are encoded by adhering to the Distinguished Encoding Rules (DER) scheme. The Certificate Revocation List structure enables the revocation of Long Term Certificates, that are employed by various actors and components within the PKI system. Figure 3.3 provides a detailed depiction of the standardized message format that is employed for the purpose of securing and transmitting requests for Long Term

Certificates and Pseudonym Certificates. This format is meticulously designed to ensure the integrity and confidentiality of the communication involved in the certificate issuance process, thereby safeguarding the exchange of these critical digital credentials within the system:

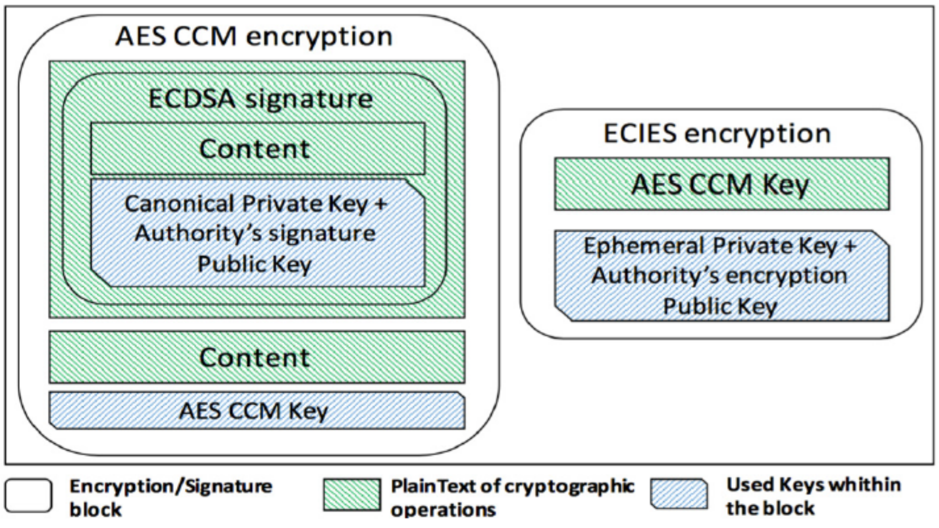

AES CCM encryption
ECDSA signature
Content
Canonical Private Key + Authority's signature Public Key
Content
AES CCM Key
ECIES encryption
AES CCM Key
Ephemeral Private Key + Authority's encryption Public Key
Encryption/Signature block
PlainText of cryptographic operations
Used Keys whithin the block

Figure 3.3: Format of messages designed for the secure transmission of LTC and PC requests. [13]

- **Data**: type is utilized as the most external container within the protocol. This design choice underscores its importance in serving as the outermost layer that holds critical information necessary for secure communications.

```
Data ::= SEQUENCE {
        version Version DEFAULT v1,
        contentType ContentType,
        content OCTET STRING OPTIONAL
 }

ContentType ::= OBJECT IDENTIFIER
```

- **EncryptedData**: is used to ensure confidentiality. It contains encrypted data along with information about the encryption algorithm used. This structure ensures that only authorized parties can decrypt and access the contents. It is designed to secure a message by encrypting data for a number of recipients following this process:

1. The sender selects an encryption algorithm and parameters (such as a nonce or IV) for encrypting the content and generates a symmetric key for this purpose.
2. This symmetric key is encrypted by the sender, and for each recipient, a specific *RecipientInfo* structure is created using the ECIES mechanism.
3. The chosen encryption algorithm, along with its associated parameters and the symmetric key for content encryption, are used to encrypt the content. For each recipient, these elements (the ciphertext, the algorithm, and its parameters) are saved into each *RecipientInfo* structure, and all these structures (in particular the array of *RecipientInfo*) are collectively formed into an EncryptedData structure.

In scenarios where the recipient's identity is verified using their public key instead of their certificate (for instance, when the recipient is in the process of requesting a certificate), the recipients field, which is of the HashedId8 type, should be computed by applying the SHA256 hash function to the compressed form of the encoded public key and then selecting the 8 least significant octets. In cases where the encrypted content is intended to be sent separately from the *EncryptedData* structure, the latter can be employed to convey the encrypted symmetric key and the encryption parameters. The inclusion of the *encryptedContent* field is not mandatory. It is defined in ASN.1 in [12]:

```
EncryptedData ::= SEQUENCE {
   version Version DEFAULT v1,
   recipients RecipientInfos,
   encryptedContentType ContentType,
   encryptionAlgorithm ContentEncryptionAlgorithmIdentifier,
   encryptedContent OCTET STRING OPTIONAL
 }

RecipientInfos ::= SEQUENCE SIZE (1..MAX) OF RecipientInfo

RecipientInfo ::= SEQUENCE {
        recipient HashedId8,
        kexalgid KeyEncryptionAlgorithmIdentifier
        DEFAULT { algorithm id-ecies-103097 },
        encryptedKeyMaterial OCTET STRING
        }
```

- **SignedData**: is used to provide data integrity and authentication and to verify the sender's identity through an ECDSA signature. This structure identifies the signer, linking the signature to a corresponding certificate. It is composed of four key elements:

- The actual content that requires signing, presented in an unencrypted form.
- A cryptographic hash (digest) of the content awaiting signature.
- The signer's identity information.
- The ECDSA signature that encompasses the aforementioned elements.

This data structure is designed with versatility in mind, accommodating both internally and externally authenticated content, the possibility of numerous signers and signatures and the capability for single-pass validation (streaming). The process of signing data involves the following steps [12]: The process of signing data involves the following steps [12]:

1. Create an empty *SignedData* structure:
   - Set the version to v1.
   - Assign the appropriate value to *signedContentType*.
2. Encapsulate the signed data:
   - Either enclose it in an OCTET STRING and include it in the *Signed-Data* structure, or
   - Keep it separate (detached or external signature).
3. Each signer performs the following actions:
   - Select their two favored hashing algorithms: the first for computing a hash of the signed content and another for hashing the attributes. And if desired, these hash algorithm identifiers can be added to the *hashAlgorithms* array to aid in the streamlined verification of the signature.
   - Process the signed content through the chosen hash algorithm and record the outcome in an *Attribute* structure labeled *attr-messageDigest*.
   - Construct a *SignerInfo* structure that includes:
     * The aforementioned *Attribute* structures within the *signedAttributes* array and a facultative *Attribute* labeled *attr-signingTime* within the *signedAttributes* array.
     * The *signerIdentifier* adjusted to the correct identifier and if desired, the sequence of certificates may be included to verify the identity of the signatory.
     * The *digestAlgorithm* matching the hash algorithm used on the signed content and the *signatureAlgorithm* corresponding to the cryptographic signature method employed by the signatory.
     * The actual signature, which is the product of the cryptographic operation performed on the serialized *signedAttributes* structure.

– Incorporate the fully assembled *SignerInfo* structure into the array of *signerInfos*.

It is important that the *attr-messageDigest* and *attr-contentType* attributes are included in the *signedAttributes*. Their presence is mandatory. The inclusion of the *attr-signingTime* attribute is optional and may be required depending on the context. [12]. The complete definition in ASN.1 taken from [12]:

```
SignedData ::= SEQUENCE {
        version Version DEFAULT v1,
        hashAlgorithms HashAlgorithmsIdentifiers,
        signedContentType ContentType,
        signedContent OCTET STRING OPTIONAL,
        signerInfos SignerInfos
        }

HashAlgorithmsIdentifiers ::= SEQUENCE OF HashAlgorithmIdentifier

SignerInfos ::= SEQUENCE OF SignerInfo

SignerInfo ::= SEQUENCE {
 version Version DEFAULT v1,
 signer [0] SignerIdentifier DEFAULT self:NULL,
 digestAlgorithm [1] HashAlgorithmIdentifier
 DEFAULT { algorithm id-sha256 },
 signatureAlgorithm [2] SignatureAlgorithmIdentifier
 DEFAULT { algorithm ecdsa-with-SHA256 },
 signedAttributes SignedAttributes,
 certificateChain SEQUENCE OF Certificate OPTIONAL,
 signature SignatureValue
}

SignerIdentifier ::= CHOICE {
        self NULL,
        certificateDigest CertificateDigest,
        certificate Certificate
        }

CertificateDigest ::= SEQUENCE {
        algorithm HashAlgorithmIdentifier
        DEFAULT { algorithm id-sha256 },
        digest HashedId8 }

SignedAttributes ::= SEQUENCE OF Attribute

Attribute ::= SEQUENCE {
  attrType ATTRIBUTE.&id({SupportedAttributes}),
  attrValue ATTRIBUTE.&Type({SupportedAttributes}{@attrType})
  OPTIONAL
}
```

```
SignatureValue OCTET STRING

/************ -- Attributes ************/

ATTRIBUTE ::= CLASS {
        &id OBJECT IDENTIFIER UNIQUE,
        &Type OPTIONAL } WITH SYNTAX { ID &id [VALUE &Type] }

        attr-messageDigest ATTRIBUTE ::= {
                ID id-messageDigest VALUE OCTET STRING
        }

        attr-contentType ATTRIBUTE ::= {
                ID id-contentType VALUE ContentType
        }

        attr-signingTime ATTRIBUTE ::= {
                ID id-signingTime VALUE Time32
        }

        SupportedAttributes ATTRIBUTE ::= {
                attr-messageDigest |
                attr-contentType |
                attr-signingTime ,
        ... }
```

Internally, an ECDSA signature contains the following structure [12]:

```
Ecdsa-Sig-Value ::= SEQUENCE { r INTEGER, s INTEGER }
```

3.3 General Overview of the PKI Protocol

Within the PKI system, various entities communicate not only among themselves but also with ITS Stations external to the system. This section outlines the different PKI communications (requests and responses) aimed at providing ITS-Ss with certificates (Long-Term Certificates or LTCs and Pseudonym Certificates or PCs). The protocol involves 4 phases [13]:

1. **Initialization Phase**
2. **LTC Request and Response**
3. **PC Request and Response**
4. **CRL/TSL Request and Response**

The initialization phase, conducted by the manufacturer, involves registering the Intelligent Transport Systems Stations (ITS-Ss) on the PKI. This process unfolds as follows:

1. The manufacturer generates a technical key pair, which includes a Technical Public Key (TPK) and a Technical Secret Key (TSK). The TSK is generated within the Hardware Security Module (HSM) [14] using the NIST P-256 curve[15].
2. A "Profile" is selected, and the TPK is associated with a unique canonical Identifier (ID), which serves as the permanent ID for the ITS-S. The canonical ID of the ITS-S must be unique for each LTCA.
3. The manufacturer specifies the associated Service Specific Permissions (SSP) relevant to the supported services.
4. A registration request is sent to the LTCA.
5. The LTCA responds to the manufacturer with a registration confirmation.

As a result, the ITS-S is registered in the LTCA's database, and the certificates of the RCA, LTCA, and PCA are stored in the ITS-S.

3.3.1 LTC Request and Response

An LTC, or Long-Term Certificate, serves as a digital credential in communications between an Intelligent Transportation System (ITS) Station and a security management entity, confirming that the legitimate possessor has the right to request a pseudonym certificate.

1. LTC Request: An ITS-S sends an LTC request to the LTCA.
2. LTC Response: If the ITS-S is registered in the LTCA's database, the LTCA responds with an LTC response containing the requested LTC.

3.3.2 PC Request and Response

A PC is a digital credential that verifies the bearer's authority to perform certain designated tasks.

1. PC Request: An ITS-S sends a PC request to the PCA.
2. Validation Request and Response: The PCA sends a validation request to the LTCA to verify the ITS-S's LTC. The LTCA responds with a validation response.

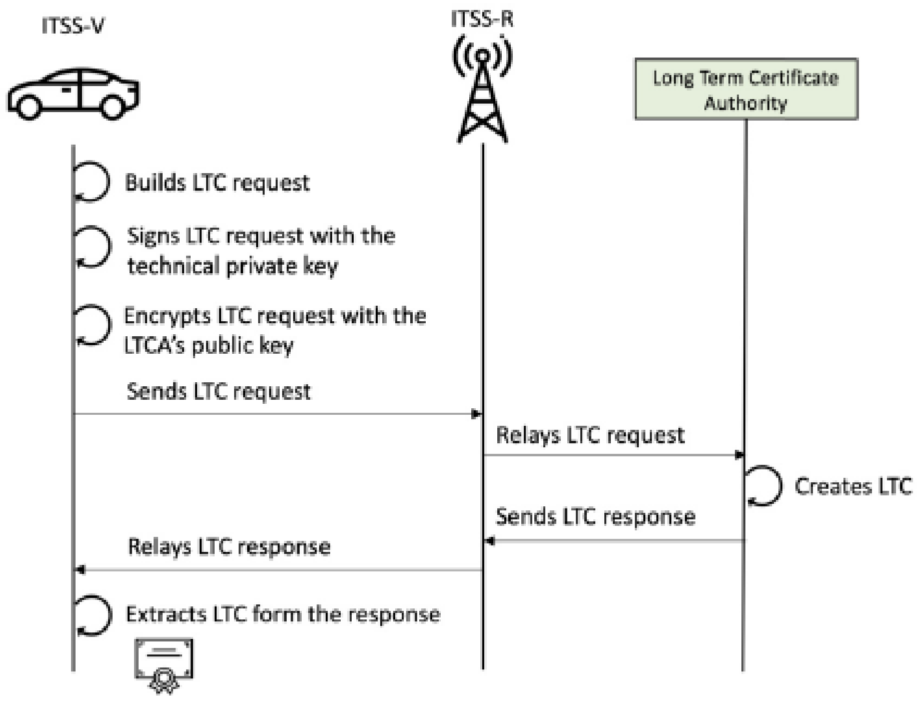

ITSS-V
ITSS-R
Long Term Certificate Authority
Builds LTC request
Signs LTC request with the technical private key
Encrypts LTC request with the LTCA's public key
Sends LTC request
Relays LTC request
Creates LTC
Sends LTC response
Relays LTC response
Extracts LTC form the response

Figure 3.4: LTC request and response [13, 16]

3. Verification Step: The LTCA performs a verification step to validate the ITS-S's LTC before the PCA can issue a PC.

4. PC Issuance: Upon successful validation, the PCA creates and sends a PC to the requesting ITS-S.

CRL/TSL Request and Response

A Certificate Revocation List is a digitally signed list issued by a Certificate Authority, detailing the identities of certificates that are no longer considered valid. A Trust-service Status List is a signed collection of newly issued RCA, LTCA, and PCA certificates, along with PKI service addresses (PCA and DC). The RCA signs this list, which can then be transmitted over the air.

1. CRL/TSL Requests: An ITS-S sends a CRL (Certificate Revocation List) or TSL (Trusted Service Status List) request to the Distribution Center (DC).

2. CRL/TSL Responses: The DC responds with a CRL or TSL response containing the requested list.

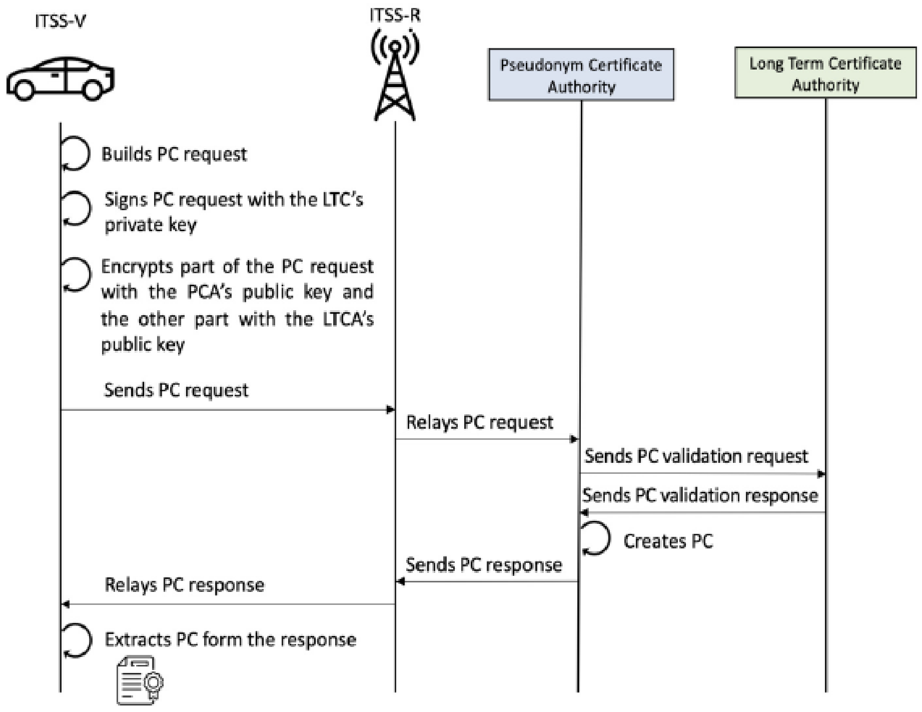

ITSS-V
ITSS-R
Pseudonym Certificate Authority
Long Term Certificate Authority
Builds PC request
Signs PC request with the LTC's private key
Encrypts part of the PC request with the PCA's public key and the other part with the LTCA's public key
Sends PC request
Relays PC request
Sends PC validation request
Sends PC validation response
Creates PC
Sends PC response
Relays PC response
Extracts PC form the response

Figure 3.5: PC request and response [13, 16]

3. List Extraction and Verification: The ITS-S extracts the list and verifies its authenticity by checking the signature of the authorized RCA.

3.3.3 Keys

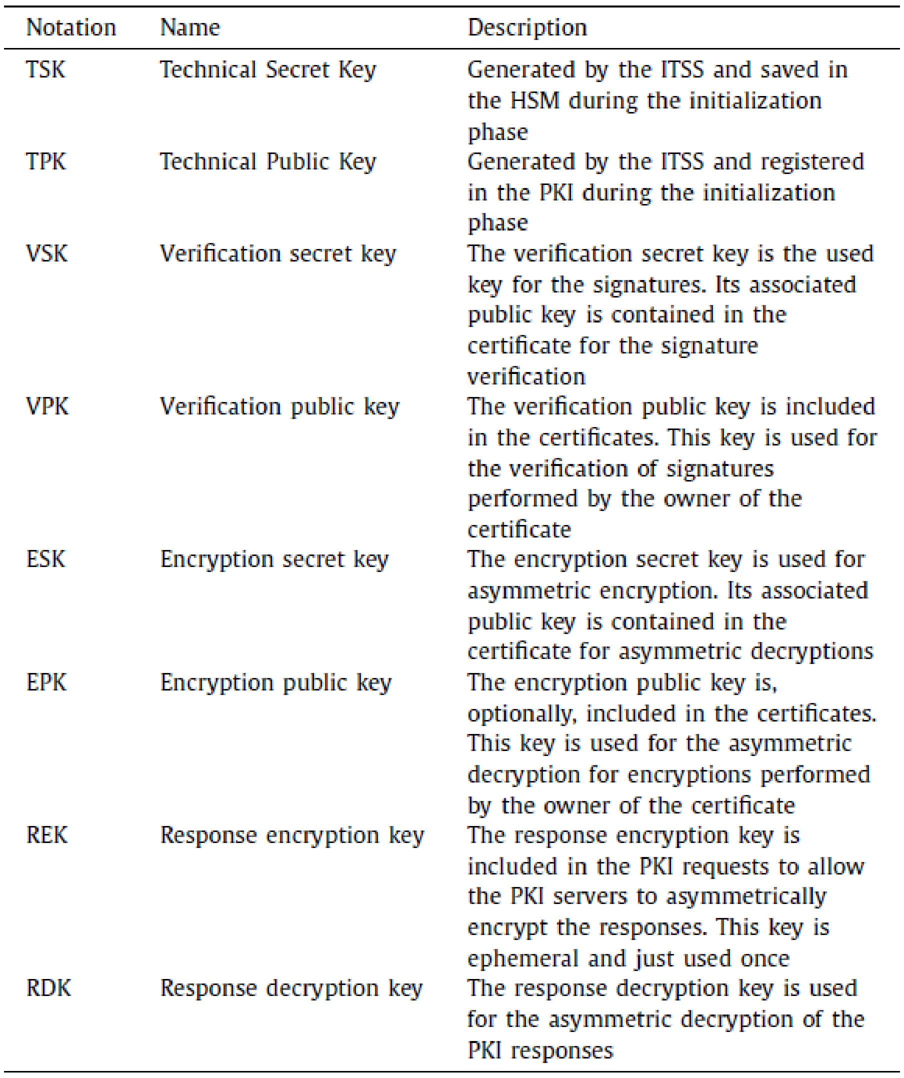
First and foremost, it is crucial to thoroughly delineate all the keys used in the complex communication protocol of the SCOOP Public Key Infrastructure, as accurately depicted in Figure 3.6.

3.3.4 Technical Specifications

LTC request

To provide further detail, the ITS-S must construct an LTC request. This is achieved by executing the steps outlined in Figure 3.7.

1. Generate the following elliptic keys:
   - (RSK, REK).



Notation	Name	Description
TSK	Technical Secret Key	Generated by the ITSS and saved in the HSM during the initialization phase
TPK	Technical Public Key	Generated by the ITSS and registered in the PKI during the initialization phase
VSK	Verification secret key	The verification secret key is the used key for the signatures. Its associated public key is contained in the certificate for the signature verification
VPK	Verification public key	The verification public key is included in the certificates. This key is used for the verification of signatures performed by the owner of the certificate
ESK	Encryption secret key	The encryption secret key is used for asymmetric encryption. Its associated public key is contained in the certificate for asymmetric decryptions
EPK	Encryption public key	The encryption public key is, optionally, included in the certificates. This key is used for the asymmetric decryption for encryptions performed by the owner of the certificate
REK	Response encryption key	The response encryption key is included in the PKI requests to allow the PKI servers to asymmetrically encrypt the responses. This key is ephemeral and just used once
RDK	Response decryption key	The response decryption key is used for the asymmetric decryption of the PKI responses

Figure 3.6: PC request and response [13]

- (LTC-VSK, LTC-VPK).
- optionally (LTC-ESK, LTC-EPK)

2. Construct an *InnerECRequest* structure containing the values:
   - A randomly generated *requestIdentifier*.
   - The canonical identifier of the ITS-S, the *itsId*.
   - The desired attributes.
   - Some optional restrictions.

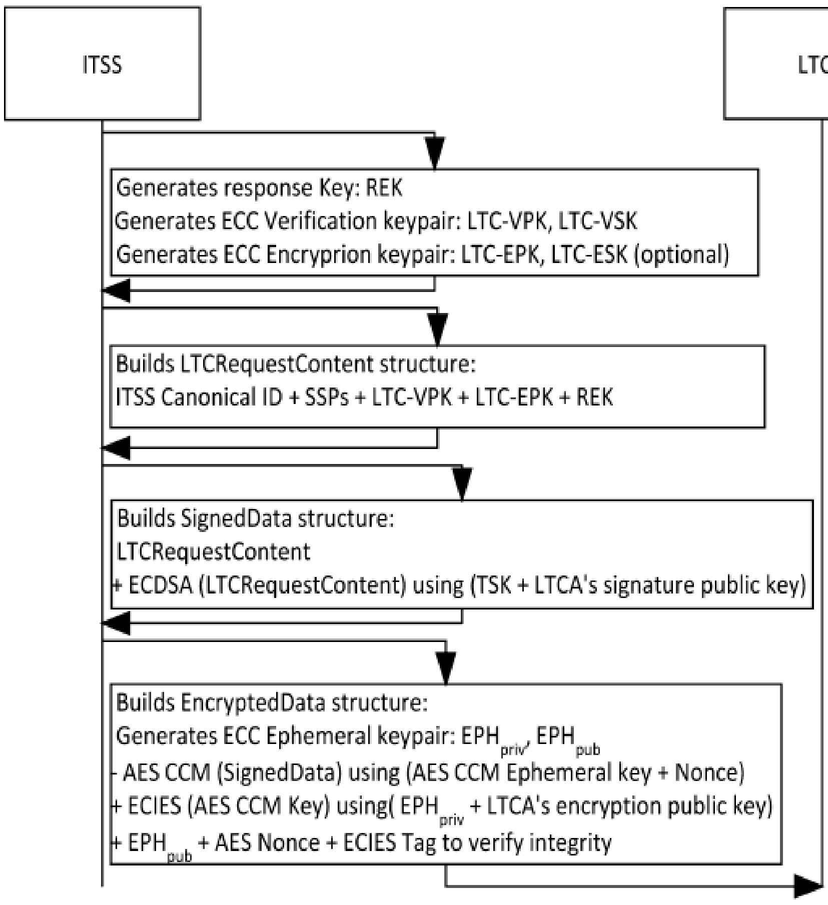

ITSS
LTCA
Generates response Key: REK
Generates ECC Verification keypair: LTC-VPK, LTC-VSK
Generates ECC Encryprion keypair: LTC-EPK, LTC-ESK (optional)
Builds LTCRequestContent structure:
ITSS Canonical ID + SSPs + LTC-VPK + LTC-EPK + REK
Builds SignedData structure:
LTCRequestContent
+ ECDSA (LTCRequestContent) using (TSK + LTCA's signature public key)
Builds EncryptedData structure:
Generates ECC Ephemeral keypair: EPH priv, EPH pub
+ AES CCM (SignedData) using (AES CCM Ephemeral key + Nonce)
+ ECIES (AES CCM Key) using(EPH priv + LTCA's encryption public key)
+ EPH pub + AES Nonce + ECIES Tag to verify integrity

Figure 3.7: LTC request [13]

- The *responseEncryptionKey (REK)*.

```
InnerECRequest ::= SEQUENCE {
    requestIdentifier OCTET STRING (SIZE(16)),
    itsId IA5String,
    wantedSubjectAttributes SubjectAttributes,
    wantedValidityRestrictions ValidityRestrictions OPTIONAL,
    responseEncryptionKey PublicKey
}
```

In this context, *wantedSubjectAttributes* represents the serialized form of the *subject_attributes* structure, as specified in the ETSI Standard [4]. It is mandatory that this structure includes precisely one occurrence of each of the subsequent elements:

- a verification_key,
- an *its_aid_ssp_list*.

wantedValidityRestrictions is the serialization of the subject *validity_restrictions* defined in ETSI Standard [4]; the inclusion of this field is not mandatory, as the Long-Term Certificate Authority (LTCA) possesses prior knowledge of the Intelligent Transport System Station (ITS-S) and has the capability to independently impose limitations on both the temporal validity and geographical scope of the certificate.

In the event of a network disruption during the transaction, the ITS-S has the option to utilize the same *requestIdentifier* when resending the request. Under such circumstances, it is anticipated that the ITS-S will transmit an identical request to the original.

3. A *SignedData* structure is then constructed, which:
   - Sets the *signedContentType* to indicate an Enrolment Request.
   - Contains the *InnerECRequest* within the signedContent.
   - Includes a collection of signedAttributes, featuring an attribute for the signing time.
   - Specifies the signer as the ITS-S itself.
   - Has a signature that is generated using the ITS-S's canonical private key, the *TSK*.
4. Following this, an *EncryptedData* structure is formed, where:
   - The intended recipient is the Long Term Certificate Authority, and the public key to be used for encryption is the LTCA's encryption key, the*LTCA-EPK*.
   - The *encryptedContentType* is set to indicate that it contains *SignedData*.
   - The *encryptedContent* holds the encrypted form of the *SignedData* structure.
5. Finally, a *Data* structure is put together, which:
   - Specifies the *contentType* as containing *EncryptedData*.
   - Includes the *EncryptedData* structure within the content.

If the ITS-S experiences a loss of network connectivity during the transaction, it can reuse the *requestIdentifier* to resend the exact same request.

LTC response

The ITS is designed to handle a specific data exchange format structured to ensure secure communication. This format involves a hierarchical arrangement of data structures, primarily focusing on the encapsulation of encrypted and signed data, contingent upon the successful validation of a response encryption key. Upon receipt of an LTC request, the LTCA performs the following steps:

1. **When the LTCA Successfully Validates the Response Encryption Key**:
    - The primary layer is a *Data* structure, identified by its content type as *id-ITS-ISE-ct-EncryptedData.*
    - Within this *Data* structure, there is an *EncryptedData* structure. This structure is notable for several key features:
        - It references the *responseEncryptionKey* specified in the initial request. The recipient's identifier is calculated as outlined in the *EncryptedData* section.
        - The type of encrypted content is designated as *id-ITS-ISE-ct-SignedData*.
        - Upon decryption, the encrypted content reveals a *SignedData* structure.
2. **When the LTCA Fails to Validate the Response Encryption Key**:
    - In scenarios where the LTCA cannot process the *responseEncryptionKey*, the format adjusts accordingly:
        - The data structure remains the outermost layer but is now identified by the content type *id-ITS-ISE-ct-SignedData.*
        - This structure directly contains a *SignedData* structure, bypassing the need for the *EncryptedData* layer.

Regardless of the LTCA's ability to validate the *responseEncryptionKey*, the *SignedData* structure adheres to a consistent format:

- It is identified by the *signedContentType id-ITS-ISE-ct-EnrolmentResponse.*
- It encapsulates an InnerECResponse.
- The signer field is populated with the *certificateDigest*, which includes the HashedId8 of the LTCA certificate.
- The signature is generated using the LTCA's private verification key, corresponding to the public verification key found in the LTCA certificate, the *LTCA-VSK.*

The InnerECResponse Structure:

```
InnerECResponse ::= SEQUENCE {
        requestHash OCTET STRING (SIZE(16)),
        responseCode EnrolmentResponseCode,
        certificate OCTET STRING OPTIONAL,

        cAContributionValue INTEGER OPTIONAL
        }

-- requestHash is a truncated SHA256 of the whole
        Data structure received
```

- Begins with the *requestHash*, which is derived from the left-most 16 octets of the SHA-256 digest of the *Data* structure received in the request.
- Includes a *responseCode* that signifies the outcome of the request.
  - A *responseCode* of 0 indicates a positive outcome, leading to the issuance of a certificate. Optionally, a CA contribution value may also be provided, allowing the ITS to compute its private key for the LTC certificate using ECQV (Elliptic Curve Qu-Vanstone) for implicit certificates.
  - A *responseCode* other than 0 indicates a negative outcome, resulting in the non-issuance of both a certificate and a CA contribution value.

Pseudonym Certificate request

After receiving the Long Term Certificate, the ITS-S can proceed to request Pseudonym Certificates, following the scheme described in Figure 3.8 and Figure 3.9. The procedure is as follows:

1. **Generation of Cryptographic Keys**
   - An elliptic private key, referred to as the *response-decryption-key*, is generated at random.
   - The corresponding public key, known as the *response-encryption-key*, is then computed.
   - Additionally, a secret key of 32 octets in length, termed the *hmac-key*, is randomly produced.
2. **Computation of the Key Tag**
   - Utilizing the HMAC-SHA256 function, a tag is calculated with the *hmac-key* against the combined serialization of *verificationKey* and *encryptionKey* elements (note that the *encryptionKey* is optional).

- This tag is truncated to 128 bits and is designated as the *keyTag*.

3. **Construction of the SharedATRequest Structure**

   - A *SharedATRequest* structure is assembled, incorporating:
     - A randomly generated *requestIdentifier*.
     - The *eaId* that identifies the LTCA intended for verification.
     - The previously computed *keyTag*.
     - The desired attributes for the request.
     - Any optional restrictions that may apply.
     - A specified start date and time for the request.
     - The *response-encryption-key*.

```
SharedATRequest ::= SEQUENCE {
        requestIdentifier OCTET STRING (SIZE(16)),
        eaId HashedId8, keyTag OCTET STRING (SIZE(16)),
        wantedSubjectAttributes SubjectAttributes,
        wantedValidityRestrictions ValidityRestrictions OPTIONAL,
        wantedStart Time32,
        responseEncryptionKey PublicKey
 }
```

4. **Creation of the SignedData Structure**:

   - A *SignedData* structure is formulated, featuring:
     - The *signedContentType* set to *id-ITS-ISE-ct-SharedATRequest*.
     - A collection of *signedAttributes*, including an *attr-signingTime* attribute.
     - An absent *signedContent* to indicate an external signature.
     - The signer identified by a *certificateDigest* that references the LTC.
     - A signature generated with the private verification key of the LTC certificate.

5. **Building of the EncryptedData Structure**:

   - An *EncryptedData* structure is constructed, in which:
     - The recipient is the LTCA, and the public key used is the LTCA's encryption key, the *LTCA-EPK*.
     - The *encryptedContentType* is set to *id-ITS-ISE-ct-SignedData*.
     - The *encryptedContent* includes the encrypted form of the aforementioned *SignedData* structure.

6. **Assembly of the InnerATRequest Structure**:
   - An *InnerATRequest* structure is created, containing:
     - The *verificationKey* that is being requested for certification.
     - An optional *encryptionKey* to be included in the same certificate.
     - The generated *hmac-key*.
     - The *signedByEC* which encompasses the *SharedATRequest* structure.
     - The *detachedEncryptedSignature* holding the previously mentioned *EncryptedData* structure.

```
InnerATRequest ::= SEQUENCE {
        verificationKey PublicKey,
        encryptionKey PublicKey OPTIONAL,
        hmacKey OCTET STRING (SIZE(32)),
        signedByEC SharedATRequest,
        detachedEncryptedSignature EncryptedData
        }
```

7. **Finalization with an EncryptedData Structure**:
   - An additional *EncryptedData* structure is created, featuring:
     - The PCA as the *recipient*, using the PCA's encryption_key, the *PCA-EPK*.
     - The *encryptedContentType* set to *id-ITS-ISE-ct-AuthorizationRequest*.
     - The *encryptedContent* containing the encrypted version of the *InnerATRequest* structure.

8. **Compilation into a Data Structure**:
   - A *Data* structure is assembled, characterized by:
     - The *contentType* set to *id-ITS-ISE-ct-EncryptedData*.
     - The *content* encapsulating the aforementioned *EncryptedData* structure.

Validate Pseudonym Certificate request

The Pseudonym Certificate Authority is required to meticulously construct a permissions verification request by adhering to a specific sequence of steps shown in Figure 3.10, ensuring the secure validation of permissions. The detailed procedure is as follows:

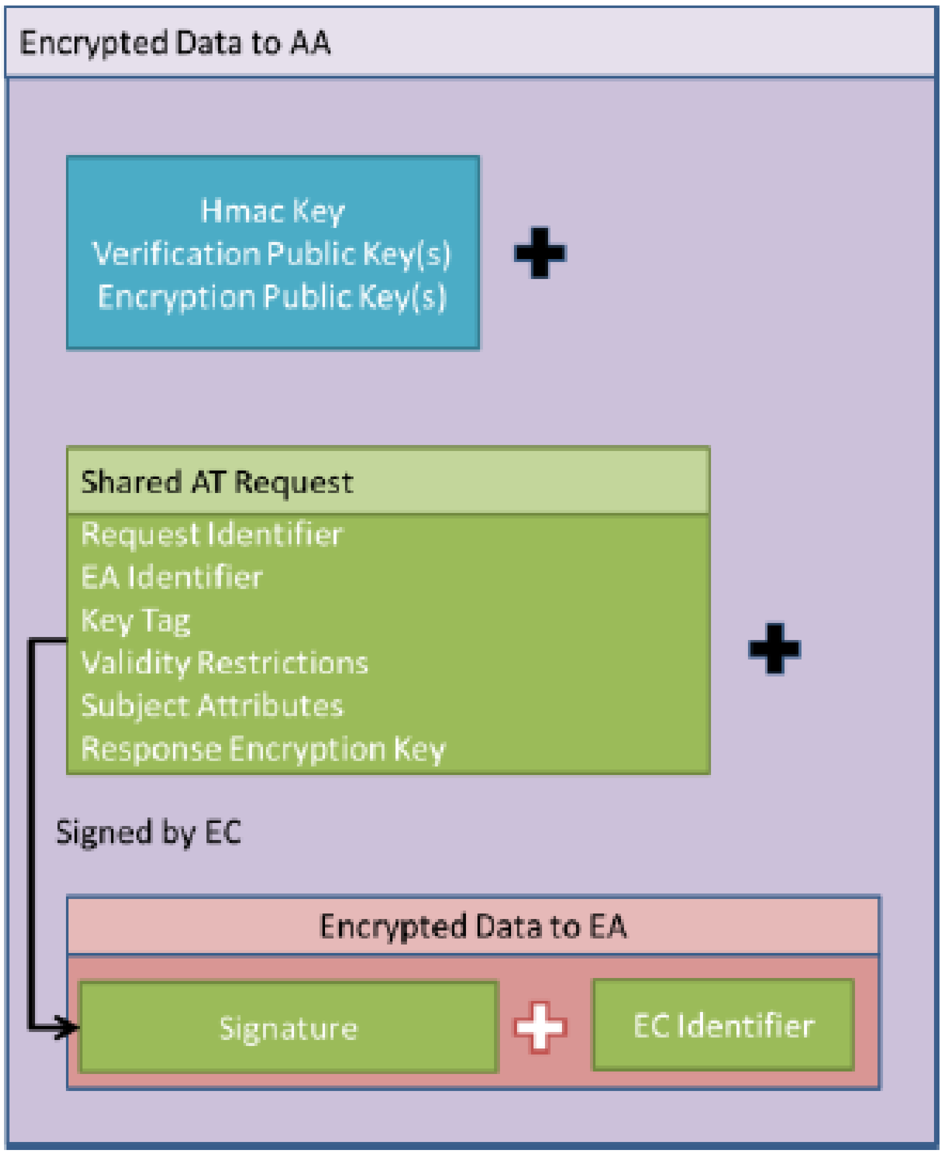


Figure 3.8: Structure of a Pseudonym Certificate request

1. **Key Generation**:
   - A new elliptic private key, known as the *response-decryption-key (REK)*, is generated through a random process.
   - Subsequently, the corresponding public key, referred to as the *response-encryption-key*, is derived from the private key.

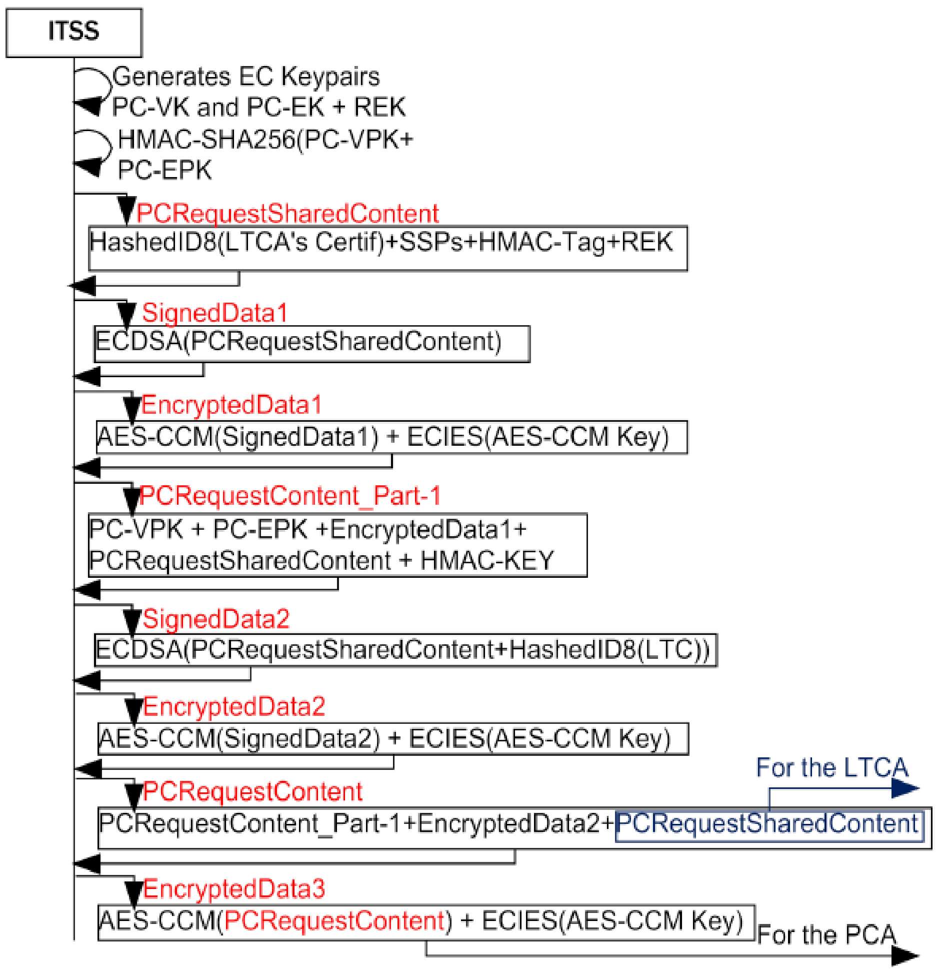

ITSS
Generates EC Keypairs
PC-VK and PC-EK + REK
HMAC-SHA256(PC-VPK+
PC-EPK
PCRequestSharedContent
HashedID8(LTCA's Certif)+SSPs+HMAC-Tag+REK
SignedData1
ECDSA(PCRequestSharedContent)
EncryptedData1
AES-CCM(SignedData1) + ECIES(AES-CCM Key)
PCRequestContent_Part-1
PC-VPK + PC-EPK +EncryptedData1+
PCRequestSharedContent + HMAC-KEY
SignedData2
ECDSA(PCRequestSharedContent+HashedID8(LTC))
EncryptedData2
AES-CCM(SignedData2) + ECIES(AES-CCM Key)
For the LTCA
PCRequestContent
PCRequestContent_Part-1+EncryptedData2+PCRequestSharedContent
EncryptedData3
AES-CCM(PCRequestContent) + ECIES(AES-CCM Key)
For the PCA

Figure 3.9: Pseudonym Certificate request [13]

2. **AuthorizationValidationRequest Structure**:

   - The PCA proceeds to create an *AuthorizationValidationRequest* structure, which includes:
     - A *requestIdentifier* that is generated randomly to uniquely identify the request.
     - The *sharedATRequest* that encompasses the *signedByEC* element, which was previously submitted as part of the pseudonym certificate request.
     - The *detachedEncryptedSignature*, also submitted in the pseudonym certificate request, ensuring the integrity and non-repudiation of the

request.
  - The *responseEncryptionKey* to facilitate secure communication with the recipient.

```
AuthorizationValidationRequest ::= SEQUENCE {
        requestIdentifier OCTET STRING (SIZE(16)),
        sharedATRequest SharedATRequest,
        detachedEncryptedSignature EncryptedData,
        responseEncryptionKey PublicKey
}
```

3. **SignedData Structure**:

   - A *SignedData* structure is then formulated, comprising:
     - The *signedContentType* set to *AuthorizationValidationRequest*, indicating the specific type of request being made.
     - The *signedContent*, that contains the actual *AuthorizationValidation-Request*.
     - A collection of *signedAttributes*, which includes an *attr-signingTime* attribute, providing a timestamp for the signature.
     - The signer field, which is populated with the PCA's certificate, authenticating the source of the request.
     - A signature that is generated using the PCA's private signature key, the *PCA-VSK*, ensuring the authenticity of the request.

4. **EncryptedData Structure**:

   - Following the creation of the *SignedData* structure, an *EncryptedData* structure is constructed, where:
     - The *recipient* is identified as the LTCA, and the public key utilized for encryption is the LTCA's encryption key, the *LTCA-EPK*.
     - The *encryptedContentType* is designated as *id-ITS-ISE-ct-SignedData*, specifying the content type within the encrypted package.
     - The *encryptedContent* includes the encrypted form of the *SignedData* structure, protecting the data from unauthorized access.

5. **Data Structure**:

   - Finally, a *Data* structure is assembled, characterized by:
     - The *contentType* set to *id-ITS-ISE-ct-EncryptedData*, indicating the nature of the content within.
     - The content that encapsulates the previously mentioned *Encrypted-Data* structure, completing the secure packaging of the request.

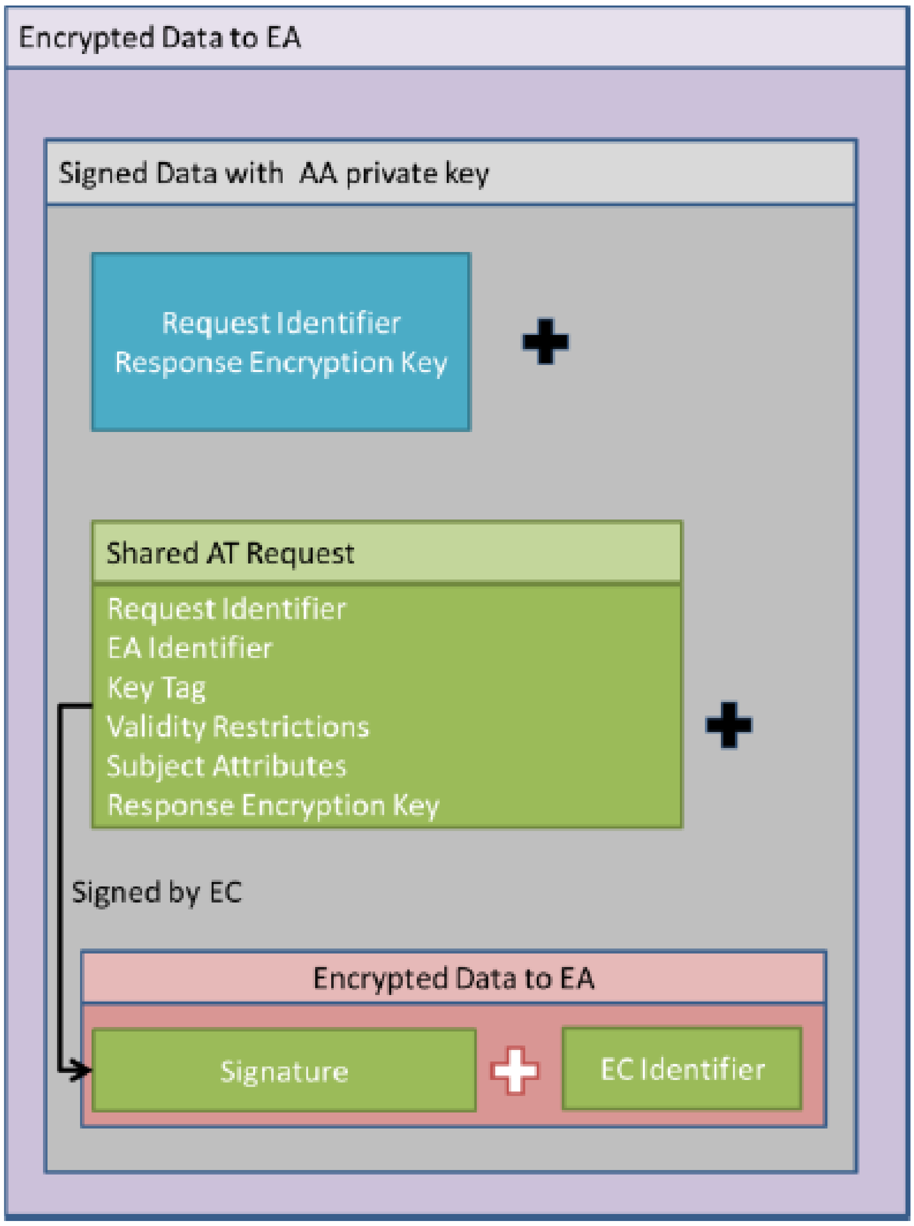


Figure 3.10: The structure of PC validation request

Validate Pseudonym Certificate response

The Pseudonym Certificate Authority is tasked with processing a *Data* structure that encapsulates an *EncryptedData* structure. This *EncryptedData* contains a *SignedData* structure, which in turn includes an *AuthorizationValidationResponse* structure. There are instances where the *EncryptedData* structure may be absent, especially if the Long Term Certification Authority has difficulty reading or validating the *responseEncryptionKey* included in the request. The process unfolds as follows:

1. **Successful Validation by LTCA**:
   - If the LTCA successfully reads and validates the responseEncryptionKey:
     - The primary layer encountered is a *Data* structure, marked by its *contentType* as *id-ITS-ISE-ct-EncryptedData*.
     - This Data structure includes an *EncryptedData* structure, which:
       * References the *responseEncryptionKey (REK)* from the request, identifying the recipient as outlined in the *EncryptedData* section.
       * Sets the *encryptedContentType* to *id-ITS-ISE-ct-SignedData*.
       * Reveals a *SignedData* structure upon decryption of the *encryptedContent*.
2. **Unsuccessful Validation by LTCA**:
   - In scenarios where the LTCA fails to read or validate the *responseEncryptionKey*:
     - The *Data* structure remains the outermost layer but is identified by the *contentType id-ITS-ISE-ct-SignedData*.
     - Directly houses a *SignedData* structure, bypassing the *EncryptedData* layer.

For both scenarios, the *SignedData* structure is expected to:

- Should be identified by the *signedContentType* set to *id-ITS-ISE-ct-AuthorizationValidationResponse*.
- Contain the *AuthorizationValidationResponse*.

```
AuthorizationValidationResponse ::= SEQUENCE {
        requestHash OCTET STRING (SIZE (16)),
        responseCode AuthorizationValidationResponseCode,
        subjectAssurance SubjectAssurance OPTIONAL,
        startDate [0] Time32 OPTIONAL,
        endDate [1] Time32 OPTIONAL
}
```

- Include a signer field that contains the *certificateDigest*, incorporating the HashedId8 of the LTCA certificate.

- Include a signature generated using the LTCA's private key, the *LTCA-VSK*, which aligns with the public verification key found in the LTCA certificate.

The InnerATResponse Structure:

- Starts with the *requestHash*, derived from the left-most 16 octets of the SHA-256 digest of the *Data* structure received in the request.
- Includes a *responseCode* that signifies the outcome of the request.

```
InnerATResponse ::= SEQUENCE {
        requestHash OCTET STRING (SIZE(16)),
        responseCode AuthorizationResponseCode,
        certificate Certificate OPTIONAL,
        cAContributionValue INTEGER OPTIONAL
        }

        (

        WITH COMPONENTS {
                responseCode (ok),
                 certificate PRESENT
        }
                |
        WITH COMPONENTS {
                responseCode ALL EXCEPT (ok),
                certificate ABSENT,
                cAContributionValue ABSENT
                }
        )
```

This structured approach to handling *Data* structures, whether through successful or unsuccessful validation of the *responseEncryptionKey*, ensures that the PCA can effectively process authorization validation responses, maintaining the integrity and security of the ITS-S framework.

Pseudonym Certificate response

The ITS-S is designed to handle a specific data structure for the issuance of Pseudonym Certificates. This structure typically includes an encrypted segment containing a signed portion, which in turn holds an *InnerATResponse*. However, in certain error scenarios, the encrypted part may be absent, if the Pseudonym Certificate Authority is unable to process the *responseEncryptionKey* from the request.

1. **Successful Validation by PCA**:

- Upon successful reading and validation of the *responseEncryptionKey (REK)* by the PCA:
    - The initial layer encountered is a *Data* structure, identified by its *contentType* as *id-ITS-ISE-ct-EncryptedData.*
    - Within this *Data* structure lies an *EncryptedData* structure, which:
        - References the *responseEncryptionKey* from the request, identifying the recipient as detailed in the *EncryptedData* section.
        - Sets the *encryptedContentType* to *id-ITS-ISE-ct-SignedData.*
        - Reveals a *SignedData* structure upon decryption of the *encrypted-Content.*

2. **Unsuccessful Validation by PCA**:

    - If the PCA fails to read or validate the *responseEncryptionKey*:
        - The *Data* structure remains the outermost layer but is identified by the *contentType id-ITS-ISE-ct-SignedData.*
        - Directly contains a *SignedData* structure, eliminating the need for the *EncryptedData* layer.

In both scenarios, the *SignedData* structure is expected to:

- Be identified by the *signedContentType id-ITS-ISE-ct-AuthorizationResponse.*
- Contain the *InnerATResponse.*
- Include a signer field filled with the *certificateDigest*, incorporating the *HashedId8* of the PCA certificate.
- Feature a signature generated using the PCA's private key, the *PCA-VSK*, which aligns with the public verification key found in the PCA certificate.

The InnerATResponse Structure:

- Begins with the *requestHash*, derived from the left-most 16 octets of the SHA-256 digest of the *Data* structure received in the request.
- Includes a *responseCode* that signifies the outcome of the request.
    - A *responseCode* of 0, indicating a positive response, results in the return of *subjectAssurance*, *startDate*, and *endDate* to be set in the corresponding Pseudonym Certificate.
    - A *responseCode* other than 0, indicating a negative response, leads to no *subjectAssurance*, no *startDate*, and no *endDate* being returned.

```
InnerATResponse ::= SEQUENCE {
        requestHash OCTET STRING (SIZE(16)),
        responseCode AuthorizationResponseCode,
        certificate Certificate OPTIONAL,
        cAContributionValue INTEGER OPTIONAL
}
```

This structured approach to handling *Data* structures, whether through successful or unsuccessful validation of the *responseEncryptionKey*, ensures that the ITS-S can effectively process authorization responses, maintaining the integrity and security of the communication with the PKI. This methodical process is crucial for the secure and efficient operation of the ITS-S, facilitating the validation and authorization of entities within the system.

3.4 Encryption of a message

It might be interesting to display the cryptographic operations implemented in SCOOP@F [12] for encrypting any message in accordance with the ETSI standard [4].

Transform a message m (N octets) from a sender into an encrypted form for a receiver.

1. Assuming an elliptic curve:
   (a) p: curve prime.
   (b) G: base point.
   (c) q: base point order.
2. The sender is solely in possession of the receiver's authenticated public key for encryption, denoted as "Kb".

- The sender initiates the process by creating a random AES key, labeled A, which is 128 bits or 16 bytes in size.
- The sender selects a nonce, labeled n, that is 12 bytes long.
- Using the AES-CCM encryption mode, the sender encrypts the plaintext message m with the *AES key A* and the nonce n, resulting in the ciphertext message M, which includes an authentication tag, expanding the total size by 16 bytes.
- The sender generates a temporary private key, denoted as r, within the range of 1 to q-1, and computes the corresponding public key v by multiplying r with the base point G on the elliptic curve, which can be represented in 33 bytes if compressed.

- The sender then computes a shared secret S by using the receiver's public encryption key, Kb. The shared secret S is the x-coordinate Px of the elliptic curve point obtained by multiplying r with Kb. The sender must ensure that the resulting point is not the identity element (0); if it is, the sender must return to the previous step.
- the sender derives two keys, ke and km, using a key derivation function applied to the shared secret S. The output is concatenated such that *ke* is 16 bytes and *km* is 32 bytes in length.
- The sender then encrypts the *AES key A* using the derived key *ke*, resulting in a ciphertext c ($c = ke \oplus A$) that is 16 bytes long.
- The sender computes a message authentication code (MAC) over the ciphertext c using the key *km*, producing a tag t that is also 16 bytes long.
- The sender assembles a message C for transmission to the receiver, which includes the following components:
  - The recipient's certificate identifier (cert_id), which is 8 bytes long.
  - The encrypted message M.
  - The encryption parameters, which include the algorithm identifier for aes_128_ccm and the nonce n, totaling 13 bytes.
  - The ephemeral public key v.
  - The encrypted *AES key c*, accompanied by its authentication tag t.

This comprehensive message C contains all the necessary elements for the receiver to authenticate and decrypt the message securely.

Chapter 4

Formal verification of cryptographic protocols

Cryptographic protocols are designed to meet specific security requirements. However, if not properly designed, they can fail to serve their intended purpose, leaving the system vulnerable to interference from malicious actors. Manual review and analysis of a protocol are important but may not suffice to detect design flaws. This is exemplified by the Needham-Schroeder case, where flaws were discovered in both versions of the protocol, despite their widespread use and presumed security. The symmetric key version was found to allow the use of a compromised old session key, as revealed by Denning and Sacco [17], while the public key authentication version was susceptible to a man-in-the-middle attack, as described by Lowe[18]. These instances underscore the importance of rigorous protocol design and thorough security analysis to ensure the effectiveness of cryptographic protocols and the security of the systems they protect. To uncover potential issues that might otherwise remain hidden, employing formal methods is a viable strategy. By creating a formal mathematical model of a system, one can utilize formal verification techniques to determine if the system meets certain criteria. Formal methods provide a mathematical foundation for system development, enabling the creation of software that is correct by construction. This approach is bolstered by validation techniques such as proofs, which verify the development process against a precise description of the required system properties. These methods encompass a variety of specifications, including algebraic and equational, and are increasingly used as tools for verification, such as static code analysis, to ensure property adherence and proper management of complexities like floating-point operations. The formal verification process for protocols is illustrated in the Figure 4.1, which typically involves four key steps.

1. The process begins with a thorough review of the protocol's specifications.

2. The next step is building the model by hand according to the given specifications.
3. This model is then converted into the input language compatible with the model checker or theorem prover.
4. The final stage involves examining the results of the formal verification. If necessary, recommendations for amendments to the standard are made based on these results.

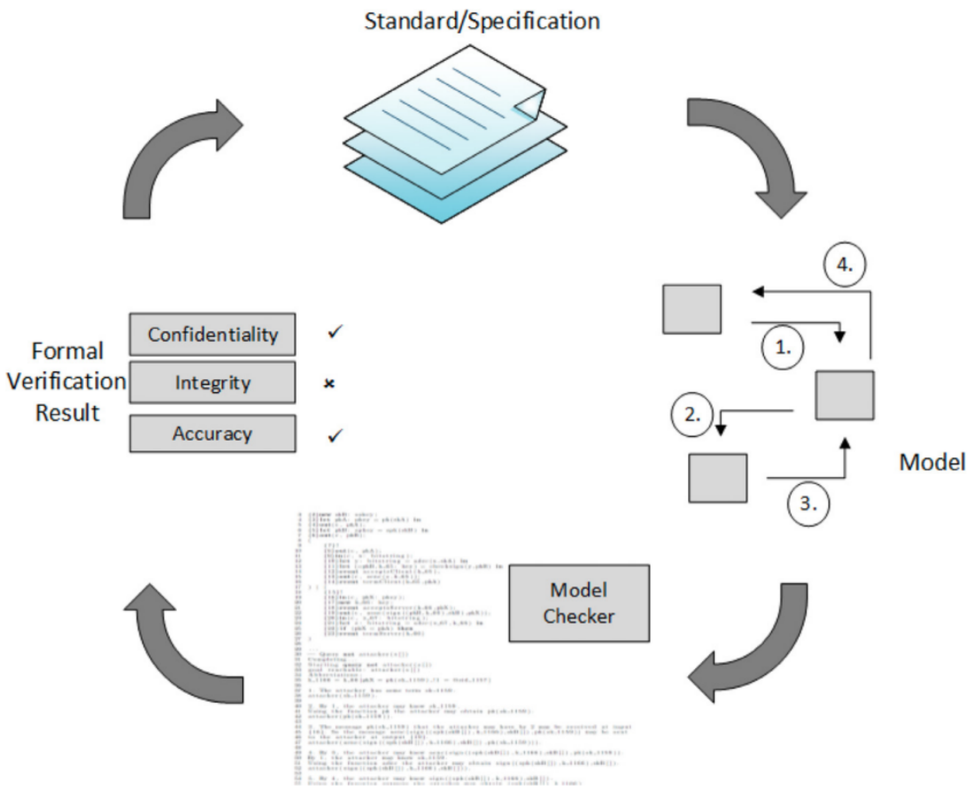


Figure 4.1: Protocol's formal verification procedure overview. [19]

Formal verification is the process of using mathematical methods to construct a proof that a system, such as a cryptographic protocol, aligns with its specified behavior. The objectives of formal verification can be broadly classified into several categories:

1. **System Behavior**: the main goal of formal verification is to mathematically ensure that a system's behavior, as described by a formal model, adheres to a specified property.

2. **Correctness**: the purpose of formal verification is to validate or refute a system's correctness in relation to a particular specification through the use of formal mathematical methods.

3. **Design Implementation**: formal verification inspects the design implementation of a system, providing a formal representation of the implementation at a higher level of abstraction that matches or is derived from the actual implementation.

4. **Error Detection**: formal verification is instrumental in identifying and pinpointing errors within designs. Should a property not hold, it can produce an error trace to aid in understanding and verifying the error through simulation.

5. **System Types**: formal verification techniques can be employed to analyze and validate a diverse array of systems, including cryptographic communication protocols, digital circuits with memory components, combinational logic circuits, and software systems represented by source code written in various programming languages.

6. **Quality Assurance**: formal verification enhances the quality of a system by offering a mathematically sound assurance of the system's correctness, irrespective of the input values.

7. **Specification and Implementation Alignment**: formal verification checks that the specification accurately reflects the designer's intentions and that the real-world implementation behaves in accordance with the model.

4.1 Formal Verification Challenges

Many questions about models are **undecidable**, which means that can always answer these questions correctly within finite time and using finite memory. However, this does not imply that an algorithm cannot correctly answer specific instances of the question using finite resources [20]. The *Halting Problem* is an example of an undecidable problem; it involves determining whether a given program will eventually halt or continue running indefinitely. It has been proven that no algorithm can accurately solve the *Halting Problem* for every possible program input [21, 22]. Even when a problem is decidable, it may be too complex to solve in a reasonable amount of time and space, as algorithms may not scale well with the problem size. Potential solutions to manage such complex problems include:

- **Semi-decision Procedures**: These are specific types of algorithms that may not always yield a conclusive answer for every scenario, but they can affirm the existence of a property if it is present. For instance, a semi-decision procedure could verify that a program ceases for certain inputs, but it might not conclude if the program continues indefinitely.

- **Abstractions**: In the realm of formal verification, abstraction refers to the process of formulating a more straightforward model of the system that maintains its key properties and is simpler to analyze. By diminishing the complexity, the system's verification becomes more manageable, albeit the results might be less exact.

- **Approximate/Non-exhaustive Modeling/Analysis**: This method entails scrutinizing a system in a manner that doesn't encompass all potential behaviors or states, yet it still offers valuable insights. It's a useful strategy when comprehensive verification is unfeasible due to limitations in resources or the system's inherent complexity.

An alternative approach to circumvent these issues is the **Correctness by Construction**: this approach deviates from the traditional method of verifying a system post-construction. Instead, it incorporates verification within the development process itself. The system is built in a manner that guarantees correctness at every stage, typically employing formally defined methods and tools.
Such strategies play a pivotal role in the development of critical systems where the margin for error is virtually non-existent, such as in the aerospace industry, autonomous car, nuclear power control systems, and medical devices, that are critical for patient safety.

For verifying the security properties of cryptographic protocols, two main verification methods are employed:

- **Symbolic**: This method uses high-level symbolic models to represent the system under consideration. A notable example is the Dolev-Yao model [23], which portrays the network as a group of trustworthy users exchanging messages composed of symbolic terms. It assumes flawless cryptographic operations, thereby excluding attacks stemming from weak cryptography. The attacker is capable of intercepting all network traffic, modifying or deleting messages, creating new ones, and using every cryptographic operation available to trustworthy users. However, terms identified as secret are initially beyond the attacker's reach. Tools such as *ProVerif* and *Tamarin* are utilized for model checking and automated theorem proving, which help to verify security properties including authentication, secrecy, and integrity.

- **Computational**: These models provide more realistic mathematical representations of security protocols, with data represented as bit string values and cryptographic primitives depicted as probabilistic polynomial-time algorithms.

These models include positive functional attributes and negative security assumptions about attackers' capabilities. An adversary is any polynomial-time algorithm with access to public communication channels and oracles representing additional information. A security property is considered computationally secure if the probability of a polynomial-time attacker violating it within one run of a probabilistic machine is negligible relative to the protocol's secret size [24].

Formal methods can be applied in two main ways: model checking and theorem proving.

- **Model checking** is an automatic verification technique that is used to verify finite state concurrent systems. It checks whether a given system model satisfies a specific property or specification, there is the model M (interpretation) and the property f (a well-formed formula, in the formal system). We check if the formula f is true under the interpretation M. If it is false, the model checker gives us a counterexample. Model Checking involves checking if a finite-state model of a system meets a given specification (typically temporal logic formulas).

- **Theorem Prover** (or proof assistants) involves using automated or interactive tools to construct proofs that a system meets its specification based on a set of axioms and inference rules. The input to a theorem prover typically includes a formal system theory and a formula that needs to be proven. The formal system theory is an abstract structure used for inferring theorems from axioms by a set of inference rules, while the formula is a mathematical statement that the system attempts to prove. The theorem proving process often starts with a state transition model (STM), which is transformed into a deductive system by generating a set of axioms and rules, these are combined with other axioms and rules, such as those of temporal logic, to form a new theory and this theory, along with the formula, is then given to the theorem prover.

4.2 ProVerif

ProVerif is a widely used security protocol verifier that employs theorem proving techniques to analyze the security properties of protocols. It has been applied to various scenarios, including simulating authentication protocols for client-server communication over insecure networks and proving unlinkability, a critical privacy property for systems like mobile phones and RFID chips [25]. *ProVerif* has also been used to analyze other protocols, such as TLS 1.3 Draft 18 [26].

4.2.1 How ProVerif Works

ProVerif employs a symbolic model of the protocol using Horn clauses and applies a resolution technique to these clauses with the aim of establishing the protocol's security attributes or identifying potential breaches [27].

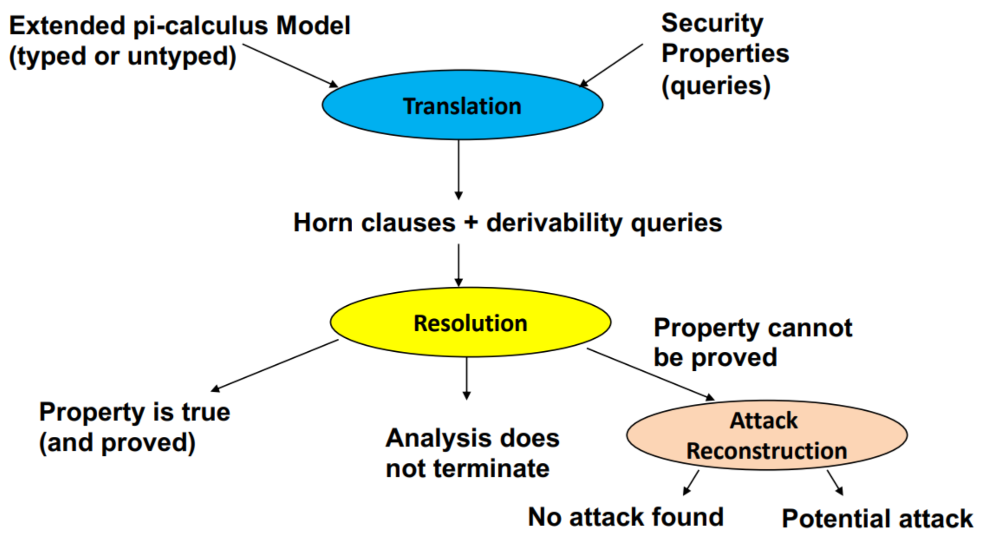

Extended pi-calculus Model (typed or untyped)
Security Properties (queries)
Translation
Horn clauses + derivability queries
Resolution
Property cannot be proved
Property is true (and proved)
Analysis does not terminate
Attack Reconstruction
No attack found
Potential attack

Figure 4.2: ProVerif internal [20]

ProVerif is based on applied pi calculus, a formal language for modeling concurrent systems where data can be exchanged on channels. Given a protocol and a security property, it may either prove that the property is satisfied or exhibit an attack. It may also return "cannot be proved," meaning that it cannot reach a conclusion. In some cases, it may not be efficient enough to conclude in a reasonable amount of time, or it may not terminate at all. It has been extended with several features to improve its precision, efficiency, and expressiveness. These features include lemmas, axioms, proofs by induction, natural numbers, and temporal queries. These not only extend the scope of *ProVerif* but can also be used to avoid false attacks and make it terminate more often. The algorithms used in *ProVerif*, such as generation of clauses, resolution, and subsumption, have been optimized to improve its speed on large examples [28].

4.2.2 Specifying Protocols

To specify protocols in *ProVerif*, users have two options:

- **Horn Clauses**: A low-level approach best suited for experts, Horn clauses offer a robust representation for modeling the symbolic manipulations of an attacker on terms. They allow for arbitrary applications across an unlimited number of protocol sessions, which is crucial for verifying protocols with an unbounded number of sessions.

- **Typed or Untyped Extended Pi-Calculus**: This higher-level method is internally converted into Horn clauses by *ProVerif*. The extended pi-calculus, which incorporates cryptographic operations into the pi calculus, enables more straightforward and legible protocol specifications. In this model, each pi-calculus process represents a protocol participant, including honest participants who adhere to the protocol specifications. Explicit modeling of the attacker is unnecessary because the attacker's capabilities are implicitly covered by the symbolic model.

4.2.3 Security Properties

ProVerif can check the following main security properties:

- **Secrecy**: it can prove that certain values or terms remain confidential and are not disclosed to the attacker by checking if the secret can be derived from the protocol's execution
- **Authentication**: it can verify that a participant is indeed the intended party in a protocol, ensuring mutual authentication between the initiator and the responder.
- **Reachability Properties**: it can demonstrate that certain states are unattainable, confirming that a protocol does not expose sensitive information or reach an insecure state.
- **Correspondence Assertions**: it can ensure that specific events occur in a particular sequence or under certain conditions, which is essential for proving non-repudiation and fair exchange.
- **Observational equivalence**: it can establish that two processes are indistinguishable to any adversary, a robust form of equivalence that is crucial for proving anonymity and privacy properties, as shown in [29].

4.2.4 Limitations

ProVerif has several limitations that affect its applicability to certain types of protocols and its ability to model and analyze them accurately. The main limitations include:

- **Handling of Algebraic Properties**: it can model any equational theory, but it may not terminate when dealing with certain algebraic properties, such as limited support for Exclusive-Or (XOR) and Diffie-Hellman exponentiation. While it can handle the commutativity of exponentiation, more complex properties can lead to non-termination.
- **Abstraction and Over-approximation**: it uses abstractions to represent protocols, which can sometimes lead to false attacks or hinder attack reconstruction due to over-approximation of the attacker's capabilities.
- **Global State**: it has limitations when dealing with protocols that maintain a global, mutable state, as its abstraction mechanisms may not accurately represent state changes, potentially resulting in false attacks or missed attacks.
- **Termination**: it implements a sound semi-decision procedure that may not terminate, meaning it may not provide a definitive answer on the satisfaction of a security property for some complex protocols.
- **Side-Channel Attacks**: it does not consider side-channel attacks such as timing, power consumption, or physical attacks against smart cards, which can provide additional information to an attacker

Chapter 5

Thesis objective

In the realm of automotive innovation, Cooperative Intelligent Transport Systems stand out as a pivotal solution in the ongoing quest for fully automated vehicles. The integration of C-ITS addresses a spectrum of challenges inherent in autonomous driving, particularly in instances where traditional automation systems might falter. Cooperative Intelligent Transport Systems (C-ITS) in Europe use Public Key Infrastructure (PKI) to ensure secure and efficient communication between vehicles and infrastructure. The main objective of this thesis is the formal analysis and verification of the communication protocol used within the Public Key Infrastructure (PKI) of the SCOOP project discussed earlier. Naturally, this necessitated an initial phase of comprehensive study of the protocol to determine what aspects to model and what to simplify. Formal verification of a protocol is a critical process that confirms the protocol's correctness, security, and adherence to specific properties. It enables a thorough analysis of the protocol's behavior, helping to pinpoint potential vulnerabilities, design flaws, and possible security breaches. In this context, we will utilize Proverif as described in che 4. The procedure with Proverif involves creating a model of the SCOOP protocol using typed pi calculus, a high-level language that is well-suited for describing communication protocols. Once the model is established, we specify the security properties we wish to examine. These properties could include aspects like confidentiality, integrity, and authentication. Proverif then rigorously analyzes the protocol against these properties, providing a robust and comprehensive assessment of the protocol's security posture. By using formal verification with Proverif, we can ensure that the SCOOP protocol is not only functionally correct but also secure against a wide range of potential threats. This process is crucial in maintaining the integrity and reliability of the protocol in real-world applications. The main phases of the analysis are as follows:

1. **Model in Proverif**:

    - Cryptographic mechanisms employed in the protocol.

- The protocol.
- The security properties.

2. **Conduct sanity checks**.
3. **Verify the security properties**.

5.1 Security Properties

ProVerif can verify several types of security properties, which are generally categorized into trace properties and equivalence properties. Trace properties are those that can be defined for each execution trace of the protocol, while equivalence properties involve comparing the behavior of two different protocols or two instances of the same protocol under different conditions. The properties we aim to validate using the tool include:

1. **Sanity checks**, specifically focusing on executability assessments.
2. **Secrecy**: This attribute guarantees that specific data within the protocol are kept confidential and inaccessible to unauthorized parties.
3. **Authentication and Correspondences**: This feature confirms the identities of entities communicating through the protocol, ensuring that the participants are genuinely who they purport to be.
4. **Integrity**
5. **Non-repudiation**
6. **Privacy**, with a particular emphasis on **Anonymity**.

5.2 Attack scenarios

In pursuit of more realistic use cases, the project underwent verification in two phases, each employing distinct scenarios:

- **Scenario 1**: The attacker does not have the certificates of the authorities (LTCA and PCA).
- **Scenario 2**: The attacker possesses the certificates of the authorities (LTCA and PCA).

Chapter 6

Modelling of analysed protocol

To analyze a protocol using *ProVerif*, the protocol needs to be represented using a modified version of the *applied pi calculus*, expands upon the standard *pi calculus* by incorporating features that facilitate the execution of cryptographic functions. This modelling includes specifying the protocol's behavior as processes and defining the cryptographic primitives used. In the subsequent chapter, we will delve into the process of modelling the cryptographic primitives and the security properties. We will describe the various decisions and assumptions made to ensure the model remains as faithful as possible to the original protocol.
As extensively outlined in Chapter 3, the protocol comprises four primary phases:

1. **Initialization Phase**: The manufacturer registers the ITSS with the corresponding LTCA to enable proper vehicle communication within the PKI.

2. **LTC Request and Response**: After successful registration, the vehicle can request a Long Term Certificate from the Long Term Certification Authority.

3. **PC Request and Response**: PC Request and Response: With an LTC, the ITSS can request Pseudonym Certificates from the Pseudonym Certification Authority.

4. **CRL/TSL Request and Response**: This phase involves the exchange of Certificate Revocation Lists (CRL) and Trusted Service Lists (TSL) between the vehicle and the relevant authorities.

6.1 Initial Procedures and Presumptions

Starting with the model depicted in figure 3.1, it was essential to make certain assumptions to optimally represent the protocol. This approach facilitated the reduction of testing complexity in *ProVerif*.
Specifically, the Distribution Centre, which is responsible for supplying the ITS-Ss with the TSL (Trust Service List) and CRL (Certificate Revocation List), has not been modelled.

6.2 Security mechanisms

ProVerif uses *applied pi-calculus* to model and verify these protocols. The tool supports a wide range of cryptographic primitives, which can be defined by rewrite rules or equations.

6.2.1 Encryption

As described in chapter 2 and in 3, the process of encryption is executed through the utilization of the *AES-128-CCM* algorithm, encompassing a series of procedural steps:

1. Initialization: A unique nonce and a secret key are used to initialize the encryption process. The nonce must be carefully chosen to never be reused with the same key, as this could compromise security.

2. Authentication: Before encrypting the data, AES-CCM generates an authentication tag (also known as a MAC or MIC) using CBC-MAC. This tag is computed over the plaintext and any additional associated data that needs to be authenticated but not encrypted.

3. Encryption: The actual encryption is performed using the CTR mode, where the AES algorithm encrypts a counter value, and the resulting keystream is XORed with the plaintext to produce the ciphertext.

4. Combination: The ciphertext and the authentication tag are combined and transmitted together. This ensures that any changes to the ciphertext or the associated data can be detected by the recipient.

5. Decryption and Verification: Upon receipt, the process is reversed. The recipient decrypts the ciphertext using the same nonce and secret key and then verifies the authentication tag to ensure the data has not been tampered with.

The implementation of AES-CCM in *ProVerif* first involves computing the ciphertext, which, in our case, is of the type aead_bitstring using the function aead_enc. Subsequently, the tag is computed using the compute_tag and and a reduction is performed for computing the decryption.

```
type nonce.
type tag_aead.
type aead_bitstring.
type result_aead.

fun aead_enc(bitstring, bitstring, nonce): aead_bitstring.
fun compute_tag(bitstring, aead_bitstring, bitstring, nonce): tag_aead.

reduc forall m:bitstring, k:bitstring, n:nonce;
aead_dec(aead_enc(m,k,n), k, n) = m.
```

6.2.2 Elliptic Curves

Representing elliptic curves in *ProVerif* involves abstracting the operations and properties of elliptic curve cryptography (ECC) into a form that can be analyzed by the tool. The tool, being an automatic proof verifying tool, does not directly support the complex mathematical structures of elliptic curves. Instead, it uses *applied pi calculus* to model cryptographic protocols, including those based on ECC. The key to modelling ECC in *ProVerif* is to abstract the elliptic curve operations into rewrite rules or equations that capture the essence of ECC operations within the limitations of *ProVerif*'s analysis capabilities.
The only operation on elliptic curves required for the analysis of the protocol is multiplication, which has been modelled using an equation.

```
type point.
const G: point [data].

type secretkey.

fun ec_mult(point, secretkey): point.

equation forall x:secretkey, y:secretkey;
ec_mult(ec_mult(G,x), y) = ec_mult(ec_mult(G,y),x)
```

6.2.3 Digital Signature

As described in chapter 2 , the signature algorithm employed in the protocol is ECDSA. The provided code is a representation of the ECDSA.

```
type result_signature.
```

```
fun ok_signature(): result_signature.
fun sign(bitstring, secretkey):bitstring .

reduc forall m:bitstring, y:secretkey; getmess(sign(m, y)) = m.
reduc forall m:bitstring, s:secretkey; checksign(sign(m,s),
ec_mult(G,s)) = ok_signature().
```

6.2.4 Hash Function

The hashing algorithm utilized to compute the digest of the request is SHA-256.

```
fun hash(bitstring): bitstring.
```

6.2.5 Certificates

In the modelling of the system's PKI, two types of certificates are established: self-signed root certificates issued by the RCA, which are private, and certificates issued by the PCA, LTCA, LTC, and PC, which are categorized as 'certificate'.

```
type root_certificate.

fun issue_rootcertificate(nat, nat, point, point, secretkey) :
root_certificate [private].

type certificate.

fun issue_certificate(nat, bitstring, nat, bitstring, point,
point, secretkey) : certificate.
```

6.2.6 ECIES

XOR

XOR is used to encrypt the AES key in ECIES. However, modelling it in *ProVerif* presents a challenge due to its unique algebraic properties, specifically associativity and cancellation, leading to an infinite number of rewrite rules. Consequently, *ProVerif* does not directly support XOR operations, as their associativity can prevent the program from terminating. The modelling of XOR adheres to a concise set of rules, involving the computation of the XOR of two inputs and the definition of XOR properties through four equations [30].

```
fun xor(bitstring,bitstring):bitstring.

equation forall x:bitstring, y:bitstring;
xor(xor(x,y),y)=x.
```

```
equation forall x:bitstring, y:bitstring;
xor(y,xor(x,x))=y.

equation forall x:bitstring, y:bitstring;
xor(xor(x,y),xor(x,x))=xor(x,y).

equation forall x:bitstring, y:bitstring;
xor(xor(x,y),xor(y,y))=xor(x,y).
```

Key Derivation Function

The Key Derivation Function, as outlined in Section 2.2.4, is implemented using SHA-256 and takes an elliptic curve point along with a counter to derive two keys. In the *ProVerif* modelling, it is expressed as:

```
const zero: bitstring.
const one: bitstring.

fun hkdf(point, bitstring): bitstring.
```

MAC

In the ECIES scheme, HMAC serves the purpose of generating a tag for the authentication of the ciphertext. Within the *ProVerif* model, it is intricately designed as a function, wherein the initial parameter corresponds to the key derived from the Key Derivation Function (KDF), while the second parameter pertains to the ciphertext. Specifically, the ciphertext represents the result of XORing the AES key with another key, also generated by the KDF.

```
fun hmac(bitstring, bitstring): bitstring.
```

6.3 Protocol Analysis

In this section, the queries and security properties required in the various phases of the protocol in both scenarios are illustrated.

6.3.1 Initialization Phase

The initial phase of the protocol entails the manufacturer registering the ITSSs by sending their respective canonical ID and TPK key to the corresponding LTCA. Subsequently, the LTCA responds by transmitting the certificates of the Certification Authorities (LTCA, RCA, PCA, DC), which will be installed in the ITSS.

The interaction between the ITSS manufacturer and the LTCA during the ITSS registration process necessitates a robust security measure. These communications are conducted through a secure channel or a dedicated, physically segregated network to mitigate the risk of eavesdropping. This initial phase has been modelled in *ProVerif* by defining a RCA that issues a self-signed certificate for use as the root certificate, an LTCA, and a PCA. The LTCA and PCA send their respective IDs and public keys to the RCA, which certifies these keys by issuing two certificates. Subsequently, the manufacturer generates a canonical ID and a public key, sending them to the LTCA. The LTCA registers this Canonical ID and the corresponding TPK in a table and forwards its certificate. In this initial phase, as per the specifications, all communications occur over private channels and so in this phase is not necessary to perform any query.

6.3.2 Requests to the LTCA

After obtaining the certificates of authorities, ITS-S can submit a request for a Long Term Certificate to the respective LTCA, as described in Section 3.3.4.
In this phase, it's crucial that the adversary is unable to acquire the Canonical ID, the ITS-S signature along with its TSK, and the elliptic keys LTC-VPK and LTC-EPK.

```
query attacker(canonical_id).

query secret ltc_vpk.

query secret ltc_epk.

query secret signedData.
```

It is important that the ITS-S request is received correctly by the LTCA, particularly ensuring adherence to the security mechanisms previously outlined, and that the LTCA can verify their proper application upon receiving a request.

```
query event(ltc_request_received()).

query inj-event(ecies_correct_ltc_request()) ==>
          inj-event(send_ltc_request()).
```

6.3.3 Receipt of the Long Term Certificate

After receiving the request, the LTCA verifies its ability to read the *responsEncryptionKey (REK)*, the Canonical ID, and validate the signature with the correct TPK associated with that Canonical ID. This part of the protocol requires that the issued LTC remains confidential, and the response sent by the Long Term Certification Authority is authenticated. Therefore, authentication is crucial in this context.

```
query secret ltc_cert.

query secret signedDataLTCA.

query event(ltc_request_received()) ==>
         event(send_ltc_request()).

query inj-event(ltc_request_received()) ==>
         inj-event(send_ltc_request()).

query event(get_ltc_cert()) ==> event(release_ltc()).

query inj-event(get_ltc_cert()) ==>
         inj-event(release_ltc()).
```

6.3.4 Request of a Pseudonym Certificate

This is the most relevant and complex part of the protocol, consisting of two sub-phases where the PCA communicates with the LTCA to authorize the issuance of Pseudonym Certificates. It is crucial that attackers cannot access the data structure created for the LTCA, known as *PCRequestSharedContent*. Additionally, it is necessary to ensure that the PCA correctly receives the request and that it conforms to the specified format.

```
query event(pc_request_received_from_its()) ==>
         event(pc_request_sent_from_its()).
```

Validate Pseudonym Certificate request

After receiving a PC request from the ITS-S, the PCA deduces from the request (in particular from its HashId8 certificate) which LTCA to contact and subsequently prepares a validation message that also includes the encrypted portion received in the request. In this phase, it is essential to prevent an adversary from conducting a Man-In-The-Middle (MITM) attack or replay attacks.

```
query inj-event(wrong_PCRequestreceived_from_its()) ==>
         inj-event(pc_request_sent_from_its()).

query inj-event(validation_pc_request_received_from_pca()) ==>
```

```
          inj-event(pc_request_received_from_its()).

query inj-event(validation_PCRequestSharedContent_from_its()) ==>
          inj-event(pc_request_received_from_its()).

query inj-event(validation_pc_request_received_from_pca()) ==>
          ( inj-event(pc_request_received_from_its())
            &&
                inj-event(pc_request_sent_from_its())
          ).
```

Validate Pseudonym Certificate response

In this phase, the LTCA receives the validation request from the PCA and must ensure two things:

1. That the request indeed originates from the PCA, making it essential to verify the signature and the certificate..
2. If the request comes from the PCA, the LTCA must validate that the request inserted by the ITS-S in its request is well-structured and that it can validate the signature of the request fields with the certified public key of the certificate (LTC-VPK), ensuring that the certificate was issued by the LTCA itself.

```
query inj-event(pc_response_validation()) ==>
      inj-event(pc_request_validation()).

query inj-event(pc_sent_to_its()) ==>
          inj-event(pc_response_validation()).

query inj-event(pc_response_validation()) ==>
          (inj-event(pc_request_received_from_its())
                &&
           inj-event(pc_request_sent_from_its())).

query inj-event(pc_response_validation()) ==>
          (inj-event(pc_request_received_from_its())
                &&
           inj-event(pc_request_sent_from_its())
          ).
```

6.3.5 Receipt of a Pseudonym Certificate

In this final phase, it is intended to verify that the ITS-S correctly receives the pseudonym certificate from the PCA; therefore, authenticating the PCA is essential.

```
query inj-event(pc_received_from_pca()) ==>
          inj-event(pc_sent_to_its()).

query inj-event(pc_received_from_pca()) ==>
          (inj-event(pc_request_received_from_its())
             &&
           inj-event(pc_request_validation())
             &&
           inj-event(pc_response_validation()
           ).

query inj-event(pc_received_from_pca()) ==>
          ( inj-event(pc_response_validation) ==>
            inj-event(pc_request_validation())
          ).
```

6.4 Privacy

The cryptographic protocol must also fulfill the properties of privacy, specifically Anonymity and Unlinkability.

6.4.1 Anonimity

In this context, anonymity is defined as the capacity of a user to access a resource or service while maintaining the confidentiality of their identity. To formally represent this characteristic, the construct of *noninterference* was employed to ascertain the sufficiency of the ECIES security mechanism in ensuring the indetectability of the LTC request and response. This evaluation specifically focused on the interaction of the ITS signature with the LTC-VPK and the PC request and response, particularly in relation to the public pseudonym keys PC-VPK and PC-EPK.

```
noninterf canonical_id.

query secret signedData.

query secret signedDataLTCA.

query secret pc_vpk.

query secret pc_epk.
```

6.4.2 Unlinkability

This property refers to the user's ability to use resources or services multiple times without others being able to associate these uses together. This attribute pertains to the user's capacity to repeatedly utilize resources or services in such a manner that prevents external parties from correlating these instances of use. In order to conceptualize this property, the construct *choice* was employed with the objective of determining whether an adversary possesses the capability to associate two distinct requests of the pseudonym certificates (PCs).

```
        equivalence
           ...

            (PCA(ltca_cert,pca_cert, pca_vpk, pca_vsk, pca_epk, pca_esk, rca_c
            ITSSendPCRequest(ltc_esk, ltc_vsk, ltca_cert, pca_cert, rca_cert,
            LTCA(idltca, ltca_cert, ltca_esk, ltca_vsk, pca_cert, rca_cert, lt
)

            ...

            (PCA(ltca_cert,pca_cert, pca_vpk, pca_vsk, pca_epk, pca_esk, rca_c
            ITSSendPCRequest(ltc2_esk, ltc2_vsk, ltca_cert, pca_cert, rca_cert
            LTCA(idltca, ltca_cert, ltca_esk, ltca_vsk, pca_cert, rca_cert, lt
)
```

Chapter 7

Results

In this chapter, we will present the comprehensive findings derived from the automated examination performed using *ProVerif*, focusing on the security attributes outlined by the SCOOP@F project and detailed in the preceding chapter.

7.1 Sanity Checks

In *ProVerif*, a sanity check refers to a basic verification process to ensure that a cryptographic protocol operates as intended, particularly that all phases of the protocol are reachable and executable. This concept is crucial in the context of formal verification, where the goal is to mathematically prove the security properties of a protocol under certain assumptions.
Sanity checks in *ProVerif* can include various types of verifications, such as ensuring that the protocol can be executed from start to finish without logical inconsistencies, verifying the secrecy of critical information, and checking for mutual authentication between parties involved in the protocol.

7.1.1 Executability

The executability sanity check is a preliminary assessment designed to confirm that a protocol can successfully complete its intended process. Its primary goal is to ensure that the protocol is structured to execute without encountering any blocking conditions or incorrect pathways, and that it can properly conclude after achieving its intended goals.

First phase: LTC request and response

The sanity checks considered in the first phase of the protocol were performed in the following order:

- Long Term Certificate request by an ITS-S.
- Receipt of the request by the LTCA.
- Issuance of the Long Term Certificate.
- Receipt of the certificate.

Based on the conducted checks, it was determined that all the described steps were accurately executed in both scenarios.

Second phase: PC request and response

During this second phase, the following checks were conducted:

- An ITS-S requested a Pseudonym Certificate (PC).
- The PCA received the request.
- A Validate Pseudonym Certificate request was sent.
- A Validate Pseudonym Certificate response was received.
- The Pseudonym Certificate was issued.
- The certificate was received.

Upon review of the conducted checks, it was determined that all the described steps were accurately completed in each scenario.

7.2 Authentication

Within the realm of digital interactions, verifying the identity of communicating parties is an essential initial step that guarantees these entities can confirm each other's authenticity in a secure and dependable manner. This verification process is critical in safeguarding the messages shared between parties against threats like impersonation, message replay, and counterfeiting, thereby preserving the communication's integrity and security.

First phase: LTC request and response

In the initial phase, it is crucial to ensure that both the vehicle and the LTCA are authenticated. The results of the queries outlined in 6, particularly in Sections 6.3.2, which deal with the request to the LTCA and the receipt of LTC, are thoroughly examined, including the injective queries.

If the attacker possesses the certificates of the authorities (LTCA), they can create requests that appear formally valid; however, these requests will not pass validation because the signature will not be verified, as demonstrated by the following trace.

Second phase: PC request and response

In this phase of the protocol, formal verification has uncovered attacks in scenarios where the adversary acquires the certificates of the entities involved. If the adversary does not have the certificates, all authentication properties are maintained. However, if the adversary manages to obtain the certificates of the LTCA and PCA, they can initiate attacks such as making PC requests that appear formally valid from the perspective of ECIES verification, as illustrated in the trace referenced in Figure 7.2. Another attack that can be executed involves the attacker generating PC requests that are formally valid from the perspective of ECIES verification. These are then forwarded as a Validation PC request to the LTCA, as illustrated in Figure 7.3.

7.3 Privacy

Among the verified properties is privacy, specifically anonymity and unlinkability.

7.3.1 Anonimity

The formal verification of this property of the protocol was positive due to the strong security mechanisms employed by the protocol. In the absence of additional information provided to the adversary, the prover demonstrates that the ECIES mechanism is sufficiently robust to protect the symmetric encryption key used to encrypt the requests and responses. However, if the adversary is provided with a private key of an authority, for instance pca_vsk or ltca_esk, *ProVerif* is unable to provide a response and does not terminate.

7.3.2 Unlinkability

The prover does not provide a response for this property because it does not terminate.

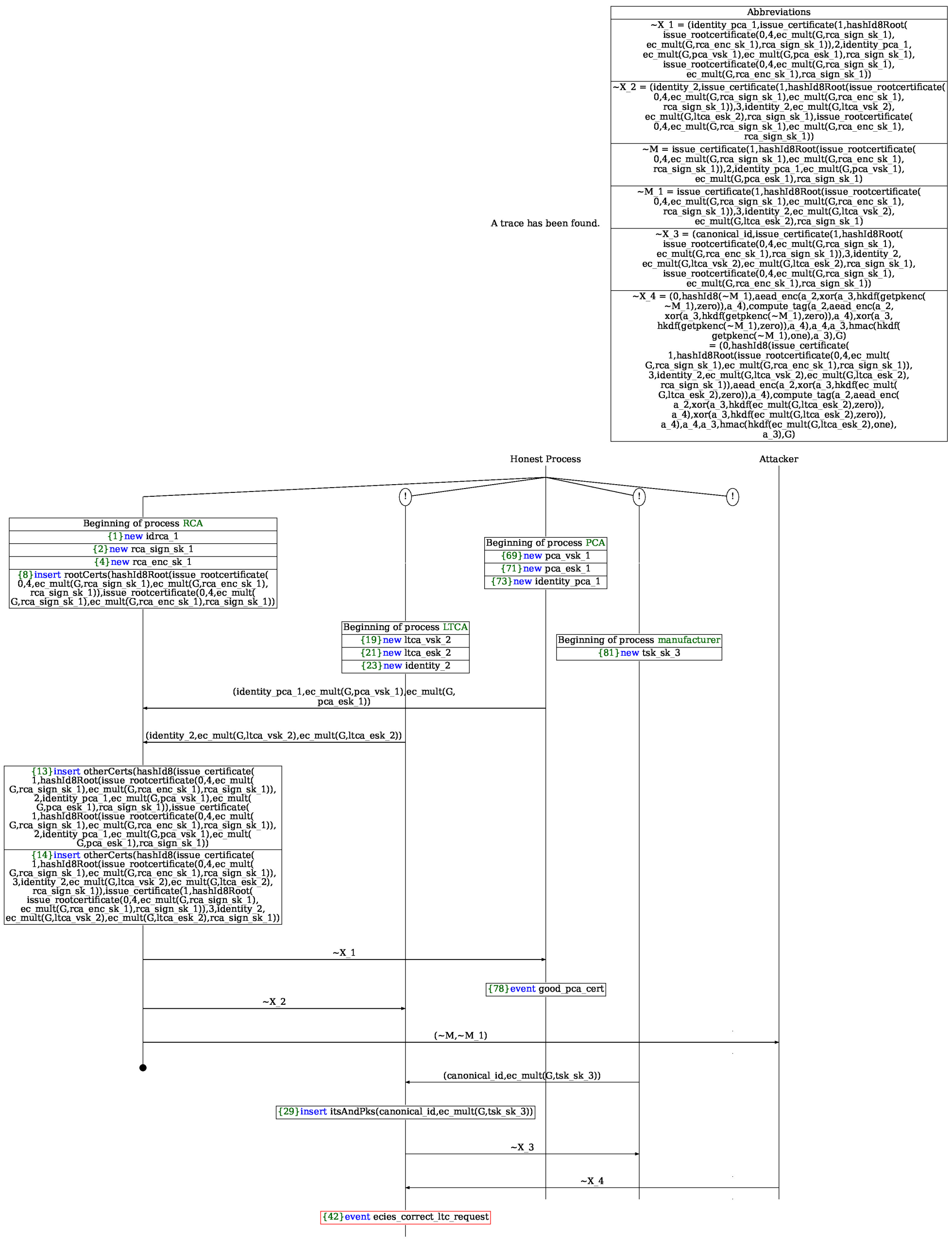
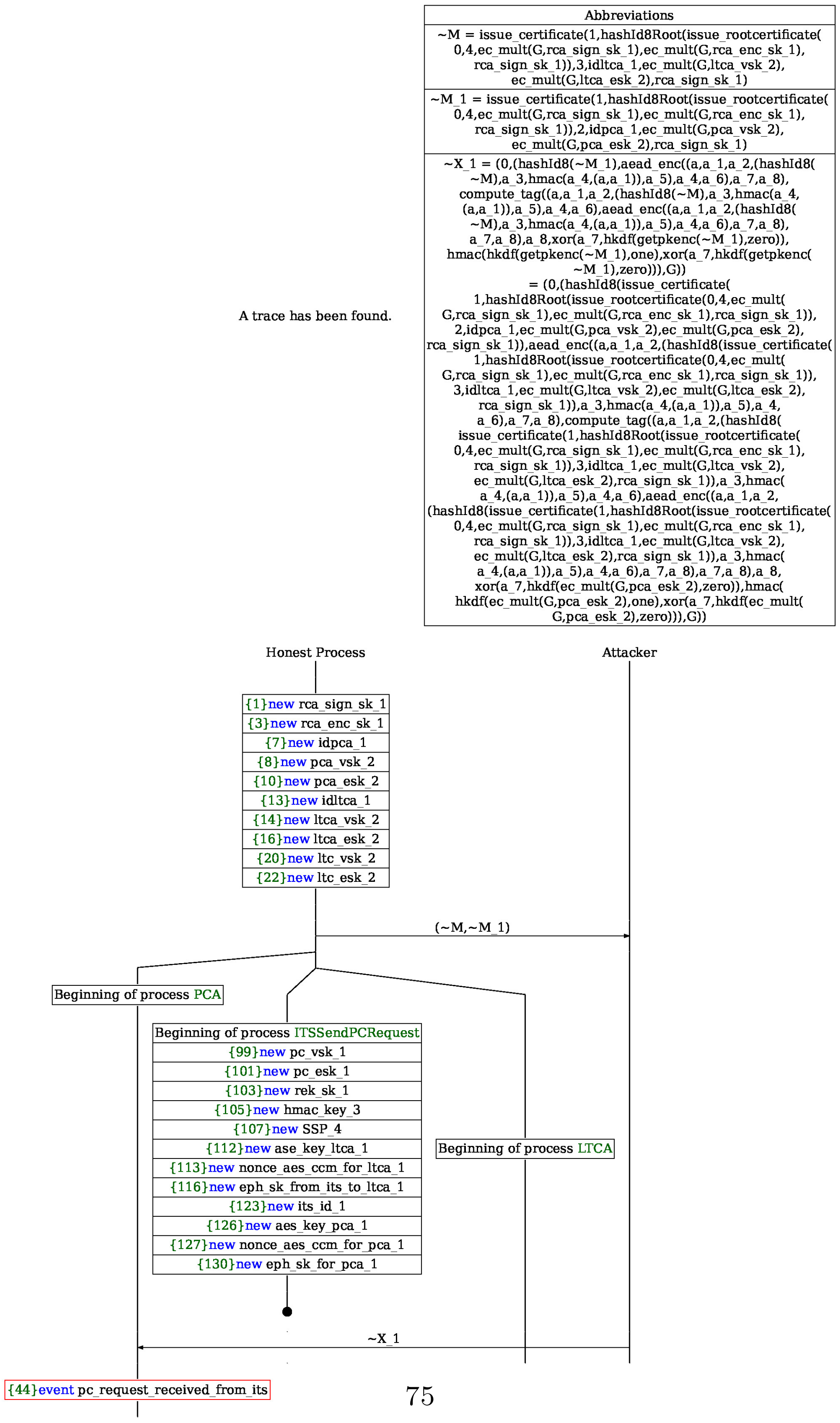

A trace has been found.
Abbreviations
~X_1 = (identity_pca_1,issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),2,identity_pca_1,ec_mult(G,pca_vsk_1),ec_mult(G,pca_esk_1),rca_sign_sk_1),issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1))
~X_2 = (identity_2,issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),3,identity_2,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2),rca_sign_sk_1),issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1))
~M = issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),2,identity_pca_1,ec_mult(G,pca_vsk_1),ec_mult(G,pca_esk_1),rca_sign_sk_1)
~M_1 = issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),3,identity_2,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2),rca_sign_sk_1)
~X_3 = (canonical_id,issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),3,identity_2,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2),rca_sign_sk_1),issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1))
~X_4 = (0,hashId8(~M_1),aead_enc(a_2,xor(a_3,hkdf(getpkenc(~M_1),zero)),a_4),compute_tag(a_2,aead_enc(a_2,xor(a_3,hkdf(getpkenc(~M_1),zero)),a_4),xor(a_3,hkdf(getpkenc(~M_1),zero)),a_4),a_4,a_3,hmac(hkdf(getpkenc(~M_1),one),a_3),G) = (0,hashId8(issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),3,identity_2,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2),rca_sign_sk_1)),aead_enc(a_2,xor(a_3,hkdf(ec_mult(G,ltca_esk_2),zero)),a_4),compute_tag(a_2,aead_enc(a_2,xor(a_3,hkdf(ec_mult(G,ltca_esk_2),zero)),a_4),xor(a_3,hkdf(ec_mult(G,ltca_esk_2),zero)),a_4),a_4,a_3,hmac(hkdf(ec_mult(G,ltca_esk_2),one),a_3),G)
Honest Process
Attacker
!
!
!
Beginning of process RCA
{1}new idrca_1
{2}new rca_sign_sk_1
{4}new rca_enc_sk_1
{8}insert rootCerts(hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1))
Beginning of process PCA
{69}new pca_vsk_1
{71}new pca_esk_1
{73}new identity_pca_1
Beginning of process LTCA
{19}new ltca_vsk_2
{21}new ltca_esk_2
{23}new identity_2
Beginning of process manufacturer
{81}new tsk_sk_3
(identity_pca_1,ec_mult(G,pca_vsk_1),ec_mult(G,pca_esk_1))
(identity_2,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2))
{13}insert otherCerts(hashId8(issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),2,identity_pca_1,ec_mult(G,pca_vsk_1),ec_mult(G,pca_esk_1),rca_sign_sk_1)),issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),2,identity_pca_1,ec_mult(G,pca_vsk_1),ec_mult(G,pca_esk_1),rca_sign_sk_1))
{14}insert otherCerts(hashId8(issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),3,identity_2,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2),rca_sign_sk_1)),issue_certificate(1,hashId8Root(issue_rootcertificate(0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),3,identity_2,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2),rca_sign_sk_1))
~X_1
{78}event good_pca_cert
~X_2
(~M,~M_1)
(canonical_id,ec_mult(G,tsk_sk_3))
{29}insert itsAndPks(canonical_id,ec_mult(G,tsk_sk_3))
~X_3
~X_4
{42}event ecies_correct_ltc_request

Figure 7.1: Long Term Certificate request performed by the attacker



```
~M = issue_certificate(1,hashId8Root(issue_rootcertificate(
0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),
rca_sign_sk_1)),3,idltca_1,ec_mult(G,ltca_vsk_2),
ec_mult(G,ltca_esk_2),rca_sign_sk_1)

~M_1 = issue_certificate(1,hashId8Root(issue_rootcertificate(
0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),
rca_sign_sk_1)),2,idpca_1,ec_mult(G,pca_vsk_2),
ec_mult(G,pca_esk_2),rca_sign_sk_1)

~X_1 = (0,(hashId8(~M_1),aead_enc((a,a_1,a_2,(hashId8(
~M),a_3,hmac(a_4,(a,a_1)),a_5),a_4,a_6),a_7,a_8),
compute_tag((a,a_1,a_2,(hashId8(~M),a_3,hmac(a_4,
(a,a_1)),a_5),a_4,a_6),aead_enc((a,a_1,a_2,(hashId8(
~M),a_3,hmac(a_4,(a,a_1)),a_5),a_4,a_6),a_7,a_8),
a_7,a_8),a_8,xor(a_7,hkdf(getpkenc(~M_1),zero)),
hmac(hkdf(getpkenc(~M_1),one),xor(a_7,hkdf(getpkenc(
~M_1),zero))),G))
= (0,(hashId8(issue_certificate(
1,hashId8Root(issue_rootcertificate(0,4,ec_mult(
G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),
2,idpca_1,ec_mult(G,pca_vsk_2),ec_mult(G,pca_esk_2),
rca_sign_sk_1)),aead_enc((a,a_1,a_2,(hashId8(issue_certificate(
1,hashId8Root(issue_rootcertificate(0,4,ec_mult(
G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),rca_sign_sk_1)),
3,idltca_1,ec_mult(G,ltca_vsk_2),ec_mult(G,ltca_esk_2),
rca_sign_sk_1)),a_3,hmac(a_4,(a,a_1)),a_5),a_4,
a_6),a_7,a_8),compute_tag((a,a_1,a_2,(hashId8(
issue_certificate(1,hashId8Root(issue_rootcertificate(
0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),
rca_sign_sk_1)),3,idltca_1,ec_mult(G,ltca_vsk_2),
ec_mult(G,ltca_esk_2),rca_sign_sk_1)),a_3,hmac(
a_4,(a,a_1)),a_5),a_4,a_6),aead_enc((a,a_1,a_2,
(hashId8(issue_certificate(1,hashId8Root(issue_rootcertificate(
0,4,ec_mult(G,rca_sign_sk_1),ec_mult(G,rca_enc_sk_1),
rca_sign_sk_1)),3,idltca_1,ec_mult(G,ltca_vsk_2),
ec_mult(G,ltca_esk_2),rca_sign_sk_1)),a_3,hmac(
a_4,(a,a_1)),a_5),a_4,a_6),a_7,a_8),a_7,a_8),a_8,
xor(a_7,hkdf(ec_mult(G,pca_esk_2),zero)),hmac(
hkdf(ec_mult(G,pca_esk_2),one),xor(a_7,hkdf(ec_mult(
G,pca_esk_2),zero))),G))
```

Figure 7.2: PC request performed by the attacker

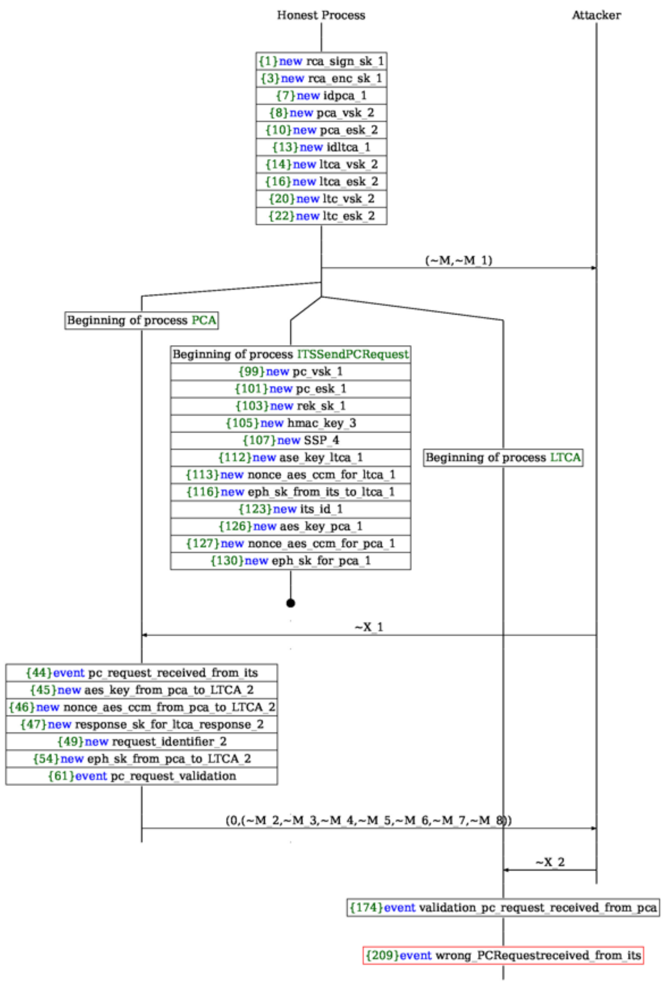

Honest Process
Attacker
{1}new rca_sign_sk_1
{3}new rca_enc_sk_1
{7}new idpca_1
{8}new pca_vsk_2
{10}new pca_esk_2
{13}new idltca_1
{14}new ltca_vsk_2
{16}new ltca_esk_2
{20}new ltc_vsk_2
{22}new ltc_esk_2
(~M,~M_1)
Beginning of process PCA
Beginning of process ITSSendPCRequest
{99}new pc_vsk_1
{101}new pc_esk_1
{103}new rek_sk_1
{105}new hmac_key_3
{107}new SSP_4
{112}new ase_key_ltca_1
{113}new nonce_aes_ccm_for_ltca_1
{116}new eph_sk_from_its_to_ltca_1
{123}new its_id_1
{126}new aes_key_pca_1
{127}new nonce_aes_ccm_for_pca_1
{130}new eph_sk_for_pca_1
Beginning of process LTCA
~X_1
{44}event pc_request_received_from_its
{45}new aes_key_from_pca_to_LTCA_2
{46}new nonce_aes_ccm_from_pca_to_LTCA_2
{47}new response_sk_for_ltca_response_2
{49}new request_identifier_2
{54}new eph_sk_from_pca_to_LTCA_2
{61}event pc_request_validation
(0,(~M_2,~M_3,~M_4,~M_5,~M_6,~M_7,~M_8))
~X_2
{174}event validation_pc_request_received_from_pca
{209}event wrong_PCRequestreceived_from_its

Figure 7.3: Validation PC request generated by the attacker

Chapter 8

Conclusions

The thesis has elucidated the SCOOP@F Public Key Infrastructure and the protocols employed for the exchange of information between vehicles, as well as between vehicles and road infrastructure. We discuss various techniques for the formal verification of cryptographic protocols. Specifically, we utilize the automated theorem prover *ProVerif*, detailing its capabilities and limitations. We then describe the SCOOP@F protocol and its associated security mechanisms. The protocol is formally modeled using *ProVerif*, and due to its complexity, it is divided into two parts and considers two attack scenarios. For each phase, we outline and verify the security properties of interest: **authentication**, **privacy**, and **secrecy**. We demonstrate that the security properties can largely be verified for the analyzed protocol. However, in scenarios where information loss occurs, attackers can potentially execute denial-of-service attacks by making requests to the involved entities. Additionally, some analyses regarding equivalent processes through observational equivalences with *ProVerif* were not completed, likely due to the complexity of the mechanisms and the tool's limitations in handling such approximations.

8.1 Future work

This project naturally paves the way for further development through the investigation of additional automated theorem proving tools. Among these tools is *Tamarin*, which utilizes multiset rewriting rules for modeling protocols and supports reasoning in the presence of equational theories. Such theories are necessary for cryptographic operations like the Diffie-Hellman key exchange and XOR. *Tamarin*'s approach allows for a more direct representation of cryptographic operations and their algebraic properties. Moreover, it offers two methods for proving security lemmas: an autoprover for automated solutions and a manual mode that provides step-by-step guidance. Exploring *Tamarin* could significantly enhance our understanding and application of automatic prover tools in complex scenarios.

Bibliography

[1] *Mobility and Transport*. URL: https://transport.ec.europa.eu/transport-themes/intelligent-transport-systems_en.

[2] Andreas Festag. «Standards for vehicular communication—from IEEE 802.11p to 5G». In: *e & i Elektrotechnik und Informationstechnik* 132.7 (Nov. 2015), pp. 409–416. DOI: 10.1007/s00502-015-0343-0.

[3] European Telecommunications Standards Institute (ETSI). *Intelligent Transport Systems (ITS); Security; Security Services and Architecture*. Technical Specification (TS) TS 102 731. Version 1.1.1. 2010.

[4] European Telecommunications Standard Institute (ETSI). *Intelligent Transport Systems (ITS); Security; Security header and certificate formats*. Technical Specification (TS) TS 103 097. Version 1.2.1. 2015.

[5] European Telecommunications Standard Institute (ETSI). *Intelligent Transport Systems (ITS); Security, ITS communications security architecture and security management*. Technical Specification (TS) TS 102 940. Version 1.1.1. 2015.

[6] European Telecommunications Standard Institute (ETSI). *Intelligent Transport Systems (ITS); Security; Trust and Privacy Management*. Technical Specification (TS) TS 102 941. Version 1.1.1. 2012.

[7] European Telecommunications Standard Institute (ETSI). *Intelligent Transport Systems (ITS); Security; Access Control*. Technical Specification (TS) TS 102 942. Version 1.1.1. 2012.

[8] European Telecommunications Standard Institute (ETSI). *Intelligent Transport Systems (ITS); Security; Confidentiality services*. Technical Specification (TS) TS 102 943. Version 1.1.1. 2012.

[9] Víctor Gayoso Martínez, Luis Hernandez Encinas, and Carmen Sánchez Ávila. «A Survey of the Elliptic Curve Integrated Encryption Scheme». In: *Journal of Computer Science and Engineering* 2 (Jan. 2010), pp. 7–13.

[10] *General presentation*. URL: https://www.scoop.developpement-durable.gouv.fr/en/general-presentation-a9.html.

[11] SCOOP Web Site. *Présentation PowerPoint.* URL: https://www.scoop.developpement-durable.gouv.fr/en/IMG/pdf/scoop_f_-_presentation_5_april_2018.pdf.

[12] SCOOP Project. *PKI architecture and technical specifications (v2).* Tech. rep. Deliverable 2.4.4.6. 2017.

[13] J. P. Monteuuis, Badis Hammi, Eduardo Salles, Houda Labiod, Remi Blancher, Erwan Abalea, and Brigitte Lonc. «Securing PKI Requests for C-ITS Systems». In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. 2017, pp. 1–8. DOI: 10.1109/ICCCN.2017.8038492.

[14] Marko Wolf and Timo Gendrullis. «Design, Implementation, and Evaluation of a Vehicular Hardware Security Module». In: Springer, 2011, pp. 302–318. DOI: 10.1007/978-3-642-31912-9_20.

[15] ETSI. *Intelligent Transport Systems (ITS), Security header and certificate formats.* Tech. rep. TS 103 097 V1.2.1. European Telecommunications Standards Institute (ETSI), June 2015.

[16] Badis Hammi, Jean-Philippe Monteuuis, and Jonathan Petit. «PKIs in C-ITS: Security functions, architectures and projects: A survey». In: *Vehicular Communications* 38 (2022), p. 100531. ISSN: 2214-2096. DOI: https://doi.org/10.1016/j.vehcom.2022.100531.

[17] Dorothy E Denning and Giovanni Maria Sacco. «Timestamps in key distribution protocols». In: *Communications of the ACM* 24.8 (1981), pp. 533–536. DOI: 10.1145/358722.358740.

[18] Gavin Lowe. «An attack on the Needham- Schroeder public- key authentication protocol». In: *Information processing letters* 56.3 (1995).

[19] Katharina Hofer-Schmitz and Branka Stojanović. «Towards formal verification of IoT protocols: A Review». In: *Computer Networks* 174 (2020), p. 107233. ISSN: 1389-1286. DOI: doi.org/10.1016/j.comnet.2020.107233.

[20] Riccardo Sisto. «Security Verification and Testing». Materiale didattico. Politecnico di Torino. 2022.

[21] Wikipedia. *Undecidable problem — Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/Undecidable_problem.

[22] *Undecidability.* URL: https://www.cs.rochester.edu/u/nelson/courses/csc_173/computability/undecidable.html.

[23] D. Dolev and A. Yao. «On the security of public key protocols». In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[24] M. Avalle, A. Pironti, and R. Sisto. «Formal verification of security protocol implementations: a survey». In: *Formal Aspects of Computing* 26 (2014), pp. 99–123. DOI: 10.1007/s00165-012-0269-9.

[25] D. Baelde, A. Debant, and S. Delaune. «Proving Unlinkability Using ProVerif Through Desynchronised Bi-Processes». In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2023, pp. 75–90. DOI: 10.1109/CSF57540.2023.00022.

[26] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. «Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate». In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 483–502. DOI: 10.1109/SP.2017.26.

[27] Bruno Blanchet. «The Security Protocol Verifier ProVerif and its Horn Clause Resolution Algorithm». In: *Electronic Proceedings in Theoretical Computer Science* 373 (Nov. 2022), pp. 14–22. ISSN: 2075-2180. DOI: 10.4204/eptcs.373.2. URL: http://dx.doi.org/10.4204/EPTCS.373.2.

[28] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. «ProVerif with Lemmas, Induction, Fast Subsumption, and Much More». In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 69–86. DOI: 10.1109/SP46214.2022.9833653.

[29] Simone Bussa, Riccardo Sisto, and Fulvio Valenza. «Formal Verification of a V2X Privacy Preserving Scheme Using Proverif». In: *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE. 2023, pp. 341–346.

[30] Cheng SHI and Kazuki YONEYAMA. «Verification of LINE Encryption Version 1.0 Using ProVerif». In: *IEICE Transactions on Information and Systems* E102.D (Aug. 2019), pp. 1439–1448. DOI: 10.1587/transinf.2018FOP0001.

Appendix A

Proverif Code

A.1 LTC Request and Response

```
type point.

const G: point [data].
type secretkey.

fun ec_mult(point, secretkey): point.
equation forall x:secretkey, y:secretkey; ec_mult(ec_mult(G,x),
y) = ec_mult(ec_mult(G,y),x).


type certificate.
type root_certificate.
type result.
fun ok(): result.

fun issue_rootcertificate(nat, nat, point, point, secretkey ):
 root_certificate [private].
reduc forall signerInfo: nat, typecert:nat, rcavpk:point,
 rcaepk:point, sk:secretkey;
 getrcatypecert(issue_rootcertificate(signerInfo, typecert,
 rcavpk, rcaepk, sk )) = typecert.

reduc forall signerInfo: nat, typecert:nat, rcavpk:point,
 rcaepk:point, sk:secretkey;
 getrcapksign(issue_rootcertificate(signerInfo, typecert, rcavpk,
 rcaepk, sk )) = rcavpk.

fun issue_certificate(nat, bitstring, nat, bitstring, point,
 point, secretkey ) : certificate.
```

```
fun hashId8(certificate): bitstring .
fun hashId8Root(root_certificate): bitstring .

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 checkcert(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk), ec_mult(G, sk)) = ok().

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getpksign(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = vpksign.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getpkenc(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = vpkenc.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getcid(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = cid.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getcahashid(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = hashId.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getsignerinfo(issue_certificate(signerInfo, hashId, typecert,
 cid, vpksign, vpkenc, sk)) = signerInfo.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 gettypecert(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = typecert.

type result_signature.

fun ok_signature(): result_signature.

fun sign(bitstring, secretkey):bitstring .

reduc forall m:bitstring, y:secretkey; getmess(sign(m, y)) = m.

reduc forall m:bitstring, s:secretkey; checksign(sign(m,s),
 ec_mult(G,s)) = ok_signature().
```

```
type nonce.
type tag_aead.
type aead_bitstring.
type result_aead.

fun compute_tag(bitstring, aead_bitstring, bitstring, nonce):
 tag_aead.
fun aead_enc(bitstring, bitstring, nonce): aead_bitstring.
reduc forall m:bitstring, k:bitstring, n:nonce;
 aead_dec(aead_enc(m,k,n), k, n) = m.

fun hash(bitstring): bitstring.

fun xor(bitstring,bitstring):bitstring.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),y)=x.
equation forall x:bitstring, y:bitstring; xor(y,xor(x,x))=y.
equation forall x:bitstring, y:bitstring;
 xor(xor(x,y),xor(x,x))=xor(x,y).
equation forall x:bitstring, y:bitstring;
 xor(xor(x,y),xor(y,y))=xor(x,y).

const zero: bitstring.
const one: bitstring.
fun hkdf(point, bitstring): bitstring.

fun hmac(bitstring,bitstring): bitstring.

free channel_its_ltca:channel .

free channel_its_pca : channel .

free chan_secret_request_ltca_rca: channel [private].
free chan_secret_response_ltca_rca : channel [private].

free chan_secret_request_pca_rca: channel [private].
free chan_secret_response_pca_rca : channel [private].

free chan_secret_manufacturer_pca : channel [private].
free chan_secret_manufacturer_receiver_pca : channel [private].

free chan_secret_manufacturer_ltca_request : channel [private].
free chan_secret_manufacturer_ltca_response : channel [private].

free channel_its_manufacturer:channel [private].
free channel_attack:channel.

(**)
```

```
not attacker(new ltca_vsk).
not attacker(new ltca_esk).
not attacker(new pca_vsk).
not attacker(new pca_esk).
not attacker(new tsk_sk).
not attacker(new ltc_vsk).
not attacker(new ltc_esk).
not attacker(new rek_sk).
not attacker(new rca_sign_sk).
not attacker(new rca_enc_sk).
not attacker(new eph_sk).
not attacker(new new_eph_sk).

(**)
table rootCerts(bitstring, root_certificate).
table otherCerts(bitstring, certificate).
table itsAndPks(bitstring, point).

free canonical_id:bitstring [private].


(**)
event release_ltc().
event not_phase_2().
event good_pca_cert().
event good_certs().
event get_ltc_cert().
event not_in_ltca_db().
event ltc_request_received().
event send_ltc_request().
event ecies_correct_ltc_request().


query event(get_ltc_cert()).
query event(good_certs()).
query attacker(canonical_id).

query secret signedData.


query secret ltc_cert.
query event(ltc_request_received).

query event (not_in_ltca_db()).

query event(ltc_request_received()) ==> event(send_ltc_request()).

query inj-event(ltc_request_received()) ==> inj-event(send_ltc_request()).
query event(get_ltc_cert()) ==> event(release_ltc()).

query inj-event(get_ltc_cert()) ==> inj-event(release_ltc()).

query secret signedDataLTCA.

noninterf canonical_id.

query attacker(canonical_id).

query inj-event(ecies_correct_ltc_request()) ==>
 inj-event(send_ltc_request()).

let RCA =
new idrca:bitstring;

new rca_sign_sk:secretkey;
let  rca_sign_pk = ec_mult(G, rca_sign_sk) in

new rca_enc_sk:secretkey;
let  rca_enc_pk = ec_mult(G, rca_enc_sk) in


let rca_cert =  issue_rootcertificate(0, 4, rca_sign_pk,
 rca_enc_pk, rca_sign_sk ) in

let rca_hashid8 = hashId8Root(rca_cert) in

insert rootCerts(rca_hashid8, rca_cert);


in(chan_secret_request_pca_rca,(idpca:bitstring, pcapksign:point,
 pcapkenc:point) );

in(chan_secret_request_ltca_rca,(idltca:bitstring,
 ltcapksign:point, ltcapkenc:point) );

let pca_cert = issue_certificate(1, rca_hashid8, 2, idpca,
 pcapksign, pcapkenc, rca_sign_sk) in
let ltca_cert = issue_certificate(1, rca_hashid8, 3, idltca,
 ltcapksign, ltcapkenc, rca_sign_sk) in

insert otherCerts(hashId8(pca_cert), pca_cert);
insert otherCerts(hashId8(ltca_cert), ltca_cert);
out(chan_secret_response_pca_rca, (idpca, pca_cert, rca_cert));

out(chan_secret_response_ltca_rca, (idltca, ltca_cert, rca_cert));
out(channel_attack, (pca_cert, ltca_cert));
0.

let LTCA =

new ltca_vsk:secretkey;
let ltca_vpk:point = ec_mult(G, ltca_vsk) in

new ltca_esk:secretkey;
let ltca_epk:point = ec_mult(G, ltca_esk) in

new identity:bitstring;

out (chan_secret_request_ltca_rca, (identity, ltca_vpk,
 ltca_epk));

in(chan_secret_response_ltca_rca, (=identity,
 ltca_cert:certificate, rca_cert:root_certificate));

if checkcert(ltca_cert, getrcapksign(rca_cert)) = ok() &&
 getpkenc(ltca_cert) = ltca_epk
 && getpksign(ltca_cert) = ltca_vpk && gettypecert(ltca_cert) = 3
 && getrcatypecert(rca_cert) = 4 then

let my_cert_id = hashId8(ltca_cert) in

in (chan_secret_manufacturer_ltca_request, (=canonical_id,
 its_station_vpk:point));

insert itsAndPks(canonical_id, its_station_vpk);

out(chan_secret_manufacturer_ltca_response, (canonical_id,
 ltca_cert, rca_cert));

in(channel_its_ltca, (typeRequest:nat, certId:bitstring,
 ciphertex_aesccm:aead_bitstring, tag_aesccm:tag_aead,
 nonce_aes_ccm:nonce,
 ciphertex_ecies:bitstring, tag_ecies:bitstring,
 eph_pk:point ));

if certId =  my_cert_id && typeRequest = 0 then

let shared = ec_mult(eph_pk, ltca_esk) in

let ke = hkdf(shared,zero) in
let km = hkdf(shared,one) in

let tag_ecies_computed = hmac(km, ciphertex_ecies) in
```

```
if tag_ecies = tag_ecies_computed then

let aes_key = xor(ciphertex_ecies, ke) in

let firma = aead_dec(ciphertex_aesccm, aes_key, nonce_aes_ccm) in

let tag_aesccm_computed = compute_tag(firma, ciphertex_aesccm,
 aes_key, nonce_aes_ccm) in

if tag_aesccm = tag_aesccm_computed then(

event ecies_correct_ltc_request();

let (canonical_id_its:bitstring, SSP:bitstring, LTC_VPK:point,
 LTC_EPK:point, REK:point)  = getmess(firma) in


get itsAndPks(=canonical_id, tsk_its_pk) in (


if checksign(firma, tsk_its_pk) = ok_signature() then

event ltc_request_received();

new new_nonce_aes:nonce;

let ltc_cert = issue_certificate(1, hashId8(ltca_cert), 0,
 canonical_id_its, LTC_VPK, LTC_EPK, ltca_vsk) in

insert otherCerts(hashId8(ltc_cert), ltc_cert);

let signedDataLTCA = sign(
(hash((certId, ciphertex_aesccm, tag_aesccm, nonce_aes_ccm,
 ciphertex_ecies,tag_ecies,eph_pk)),
0,
ltc_cert), ltca_vsk) in

new new_aes_key:bitstring; (*new aes_key*)

let new_ciphertex_aesccm = aead_enc(signedDataLTCA,new_aes_key,
 new_nonce_aes) in

let new_tag_aesccm =
 compute_tag(signedDataLTCA,new_ciphertex_aesccm, new_aes_key,
 new_nonce_aes) in

new new_eph_sk:secretkey;

let new_eph_pk = ec_mult(G, new_eph_sk) in
let new_shared = ec_mult(REK, new_eph_sk) in

let new_ke = hkdf(new_shared,zero) in

let new_km = hkdf(new_shared,one) in

let new_ciphertex_ecies = xor(new_aes_key, new_ke) in

let new_tag_ecies = hmac(new_km, new_ciphertex_ecies) in

event release_ltc();

let EncryptedData_For_ITS =  (new_ciphertex_aesccm,
 new_tag_aesccm, new_nonce_aes, new_ciphertex_ecies,
 new_tag_ecies, new_eph_pk) in

out(channel_its_ltca, (0,EncryptedData_For_ITS));

0

)else

let SignedData = sign((hash((certId, ciphertex_aesccm,
 tag_aesccm, nonce_aes_ccm,
 ciphertex_ecies,tag_ecies,eph_pk)),1), ltca_vsk) in

out(channel_its_ltca, (1,SignedData));

event not_in_ltca_db();0

)else

let SignedData = sign((hash((certId, ciphertex_aesccm,
 tag_aesccm, nonce_aes_ccm,
 ciphertex_ecies,tag_ecies,eph_pk)),1), ltca_vsk) in

out(channel_its_ltca, (1,SignedData));

0.

let PCA =

new pca_vsk:secretkey;
let pca_vpk:point = ec_mult(G, pca_vsk) in
new pca_esk:secretkey;
let pca_epk:point = ec_mult(G, pca_esk) in

new identity_pca:bitstring;
```

```
out (chan_secret_request_pca_rca, (identity_pca, pca_vpk,
 pca_epk));

in(chan_secret_response_pca_rca, (=identity_pca,
 pca_cert:certificate, rca_cert:root_certificate));

if checkcert(pca_cert, getrcapksign(rca_cert)) = ok() &&
 getpkenc(pca_cert) = pca_epk
 && getpksign(pca_cert) = pca_vpk
 && gettypecert(pca_cert) = 2 && getrcatypecert(rca_cert) = 4
 then(

let myID8 = hashId8(pca_cert) in

event good_pca_cert();

out(chan_secret_manufacturer_pca, pca_cert);

0

)else
0
.


let manufacturer =
new tsk_sk:secretkey;
let tsk_pk = ec_mult(G,tsk_sk) in
out (chan_secret_manufacturer_ltca_request, (canonical_id,
 tsk_pk));

in  (chan_secret_manufacturer_ltca_response,
 (=canonical_id,ltca_cert:certificate,
 rca_cert:root_certificate));

in  (chan_secret_manufacturer_pca, pca_cert:certificate);

if checkcert(ltca_cert,getrcapksign(rca_cert))=ok() &&
 checkcert(pca_cert,getrcapksign(rca_cert))=ok() then

event good_certs();

out(channel_its_manufacturer, (canonical_id, tsk_sk,
 tsk_pk,ltca_cert, pca_cert, rca_cert));

0.

let its_station =
in(channel_its_manufacturer, (=canonical_id, tsk_sk:secretkey,
 tsk_pk:point, ltca_cert:certificate, pca_cert:certificate,
 rca_cert:root_certificate));

new SSP:bitstring;

new ltc_vsk:secretkey;

let ltc_vpk = ec_mult(G, ltc_vsk) in

new ltc_esk:secretkey;
let ltc_epk = ec_mult(G, ltc_esk) in

new rek_sk:secretkey;
let rek_pk = ec_mult(G, rek_sk) in

let signedData = sign((canonical_id, SSP, ltc_vpk, ltc_epk,
 rek_pk), tsk_sk) in

new aes_key:bitstring;

new nonce_aes_ccm:nonce;

let ciphertex_aesccm = aead_enc(signedData, aes_key,
 nonce_aes_ccm ) in

let tag_aesccm = compute_tag(signedData, ciphertex_aesccm,
 aes_key, nonce_aes_ccm) in

new eph_sk:secretkey;

let eph_pk = ec_mult(G, eph_sk) in

let shared = ec_mult(getpkenc(ltca_cert), eph_sk) in

let ke = hkdf(shared,zero) in

let km = hkdf(shared,one) in

let ciphertex_ecies = xor(aes_key, ke) in

let tag_ecies = hmac(km, ciphertex_ecies) in

event send_ltc_request();

let EncryptedData =  (hashId8(ltca_cert), ciphertex_aesccm,
 tag_aesccm, nonce_aes_ccm, ciphertex_ecies,
 tag_ecies, eph_pk) in

out(channel_its_ltca, (0, EncryptedData));
```

```
let computed_hash = hash((hashId8(ltca_cert),ciphertex_aesccm,
 tag_aesccm, nonce_aes_ccm, ciphertex_ecies, tag_ecies, eph_pk))
 in

in(channel_its_ltca, (externaldata:nat, content:bitstring));

if externaldata = 0 then (

let (new_ciphertex_aesccm:aead_bitstring,
        new_tag_aesccm:tag_aead, new_nonce_aes:nonce,
        new_ciphertex_ecies:bitstring, new_tag_ecies:bitstring,
        new_eph_pk:point) = content in

let response_shared = ec_mult(new_eph_pk, rek_sk) in

let new_ke = hkdf(response_shared,zero) in

let new_km = hkdf(response_shared,one) in

let new_tag_ecies_computed = hmac(new_km, new_ciphertex_ecies) in

if new_tag_ecies_computed = new_tag_ecies then

let new_aes_key = xor(new_ciphertex_ecies, new_ke) in

let new_firma = aead_dec(new_ciphertex_aesccm, new_aes_key,
 new_nonce_aes) in

let new_tag_aesccm_computed =
 compute_tag(new_firma,new_ciphertex_aesccm, new_aes_key,
 new_nonce_aes) in

if new_tag_aesccm_computed = new_tag_aesccm then(

if checksign(new_firma, getpksign(ltca_cert)) = ok_signature()
 then

let new_message = getmess(new_firma) in

let (=computed_hash,response_code:nat, ltc_cert:certificate) =
 new_message in

if response_code = 0
 && checkcert(ltc_cert, getpksign(ltca_cert)) = ok()
 && canonical_id = getcid(ltc_cert) && gettypecert(ltc_cert) = 0
 then

event get_ltc_cert();
0

else
0

)else
0
else
0

)else

0.

process
(RCA() | !LTCA() | PCA() | !manufacturer() | !its_station())
```

A.2 PC Request and Response

```
type point.

const G: point [data].
type secretkey.

fun ec_mult(point, secretkey): point.
equation forall x:secretkey, y:secretkey; ec_mult(ec_mult(G,x),
 y) = ec_mult(ec_mult(G,y),x).

type certificate.
type root_certificate.
type result.
fun ok(): result.

fun issue_rootcertificate(nat, nat, point, point, secretkey ):
 root_certificate [private].

reduc forall signerInfo: nat, typecert:nat, rcavpk:point,
 rcaepk:point, sk:secretkey;
 getrcatypecert(issue_rootcertificate(signerInfo, typecert,
 rcavpk, rcaepk, sk )) = typecert.

reduc forall signerInfo: nat, typecert:nat, rcavpk:point,
 rcaepk:point, sk:secretkey;
 getrcapksign(issue_rootcertificate(signerInfo, typecert,
 rcavpk, rcaepk, sk )) = rcavpk.
```

```
fun issue_certificate(nat, bitstring, nat, bitstring, point,
 point, secretkey ) : certificate.

fun hashId8(certificate): bitstring .

fun hashId8Root(root_certificate): bitstring .

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 checkcert(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk), ec_mult(G, sk)) = ok().

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getpksign(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = vpksign.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getpkenc(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = vpkenc.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getcid(issue_certificate(signerInfo, hashId, typecert, cid,
 vpksign, vpkenc, sk)) = cid.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getcahashid(issue_certificate(signerInfo, hashId, typecert,
 cid, vpksign, vpkenc, sk)) = hashId.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 getsignerinfo(issue_certificate(signerInfo, hashId, typecert,
 cid, vpksign, vpkenc, sk)) = signerInfo.

reduc forall signerInfo: nat, hashId:bitstring, typecert:nat,
 cid:bitstring, vpksign:point, vpkenc:point, sk:secretkey;
 gettypecert(issue_certificate(signerInfo, hashId, typecert,
 cid, vpksign, vpkenc, sk)) = typecert.


type result_signature.

fun ok_signature(): result_signature.
fun sign(bitstring, secretkey):bitstring .
reduc forall m:bitstring, y:secretkey; getmess(sign(m, y)) = m.
reduc forall m:bitstring, s:secretkey; checksign(sign(m,s),
 ec_mult(G,s)) = ok_signature().


type nonce.
type tag_aead.
type aead_bitstring.
type result_aead.
fun compute_tag(bitstring, aead_bitstring, bitstring, nonce):
 tag_aead.

fun aead_enc(bitstring, bitstring, nonce): aead_bitstring.

reduc forall m:bitstring, k:bitstring, n:nonce;
 aead_dec(aead_enc(m,k,n), k, n) = m.

fun hash(bitstring): bitstring.

fun xor(bitstring,bitstring):bitstring.
equation forall x:bitstring, y:bitstring; xor(xor(x,y),y)=x.
equation forall x:bitstring, y:bitstring; xor(y,xor(x,x))=y.
equation forall x:bitstring, y:bitstring;
 xor(xor(x,y),xor(x,x))=xor(x,y).
equation forall x:bitstring, y:bitstring;
 xor(xor(x,y),xor(y,y))=xor(x,y).

const zero: bitstring.
const one: bitstring.
fun hkdf(point, bitstring): bitstring.


fun hmac(bitstring,bitstring): bitstring.

free channel_its_pca : channel.
free channel_ltca_pca : channel.

not attacker(new rca_sign_sk).
not attacker(new rca_enc_sk).


not attacker(new ltca_vsk).
not attacker(new ltca_esk).


not attacker(new pca_vsk).
(*not attacker(new pca_esk).*)

not attacker(new ltc_esk).
not attacker(new ltc_vsk).
not attacker(new new_eph_sk_for_its).

not attacker(new eph_sk_ltca_pca).

not attacker(new eph_sk_from_pca_to_LTCA).

not attacker(new eph_sk_for_pca).

not attacker( new pc_vsk).
not attacker( new pc_esk).

not attacker(new rek_sk).

not attacker(new response_sk_for_ltca_response).

not attacker(new eph_sk_from_its_to_ltca).

event  pc_sent_to_its().
event pc_received_from_pca().

event pc_request_sent_from_its().
event pc_response_validation_received_from_ltca().
event not_correct().
event hmac_keys_not_right().
event pc_response_validation_from_ltca().
event pc_request_validation().
event pc_response_validation().
event pc_request_received_from_its().


event validation_pc_request_received_from_pca().
event validation_PCRequestSharedContent_from_its().


event wrong_PCRequestreceived_from_its().

query event(pc_request_validation_received_from_pca()).
query event(pc_response_validation_from_ltca()).
query event(pc_response_validation_received_from_ltca()).
query event(pc_received_from_pca()).

query inj-event(pc_received_from_pca()) ==>
 inj-event(pc_sent_to_its()).

query inj-event(pc_response_validation_from_ltca()) ==>
 inj-event(pc_request_sent_from_pca()).

query inj-event(pc_request_received_from_its()) ==>
 inj-event(pc_request_sent_from_its()).
```

```
query secret PCRequestSharedContent.
query event(not_correct()).
query event(pcr_received()).
query event(hmac_keys_not_right()).
query secret request_pca.
query secret tag_mac_keys.

query secret pc_vpk.
query event(pc_response_validation_received_from_ltca()).
query event(pc_received_from_pca()).
query secret pc_cert.
query secret PCRequestSharedContent.
query secret PCRequestSharedContent_LTCA.

query inj-event(pc_request_received_from_its()) ==>
 inj-event(pc_request_sent_from_its()).

query inj-event(wrong_PCRequestreceived_from_its()) ==>
 inj-event(pc_request_sent_from_its()).

query event(validation_PCRequestSharedContent_from_its()).

query inj-event(validation_pc_request_received_from_pca()) ==>
 inj-event(pc_request_received_from_its()).

query inj-event(validation_PCRequestSharedContent_from_its())
 ==> inj-event(pc_request_received_from_its()).

query inj-event(pc_response_validation()) ==>
 inj-event(pc_request_validation()).

query inj-event(pc_sent_to_its()) ==>
 inj-event(pc_response_validation()).

query event(wrong_PCRequestreceived_from_its).

query event(pc_received_from_pca()).

query inj-event(pc_received_from_pca()) ==>
 inj-event(pc_sent_to_its()).

query inj-event(pc_received_from_pca()) ==>
 (inj-event(pc_request_received_from_its()) &&
  inj-event(pc_request_validation()) &&
   inj-event(pc_response_validation())).

query inj-event(pc_received_from_pca()) ==> (
 inj-event(pc_response_validation) ==>
```

```
  inj-event(pc_request_validation())).

query inj-event(pc_response_validation()) ==>
 (inj-event(pc_request_received_from_its()) &&
  inj-event(pc_request_sent_from_its())).

query inj-event(validation_pc_request_received_from_pca())
        ==> (inj-event(pc_request_received_from_its()) &&
    inj-event(pc_request_sent_from_its())).

let PCA(ltca_cert:certificate, pca_cert:certificate,
 pca_vpk:point, pca_vsk:secretkey, pca_epk:point,
 pca_esk:secretkey, rca_cert:root_certificate) =

in(channel_its_pca, (externaldata:nat, content:bitstring));

if externaldata = 0 then (

let (=hashId8(pca_cert), ciphertex_aesccm_its:aead_bitstring,
  tag_aesccm_its:tag_aead, nonce_aes_ccm_its:nonce,
  ciphertex_ecies_its:bitstring, tag_ecies_its:bitstring,
  eph_pub_its:point ) = content in

let compute_shared = ec_mult(eph_pub_its, pca_esk) in

let ke = hkdf(compute_shared, zero) in
let km = hkdf(compute_shared, one) in

let tag_ecies_computed = hmac(km, ciphertex_ecies_its) in

if tag_ecies_its = tag_ecies_computed then

let aes_key = xor(ciphertex_ecies_its, ke) in

let mess_from_its = aead_dec(ciphertex_aesccm_its, aes_key,
 nonce_aes_ccm_its ) in

let tg_aesccm_computed =
 compute_tag(mess_from_its,ciphertex_aesccm_its, aes_key,
 nonce_aes_ccm_its) in

if tg_aesccm_computed = tag_aesccm_its then

let (pc_vpk_its:point, pc_epk_its:point,
 encapsulate_ltca:bitstring, PCRequestSharedContent:bitstring,
 hmac_key:bitstring, idits:bitstring) = mess_from_its in

let (=hashId8(ltca_cert), SSP:bitstring,
 tag_mac_chiavi_from_its:bitstring, rek_pk:point ) =
 PCRequestSharedContent in

let compute_hmac_keys = hmac(hmac_key,
 (pc_vpk_its, pc_epk_its)) in

if compute_hmac_keys = tag_mac_chiavi_from_its then(

event pc_request_received_from_its();

new aes_key_from_pca_to_LTCA:bitstring;
new nonce_aes_ccm_from_pca_to_LTCA:nonce;

new response_sk_for_ltca_response:secretkey;

let response_pk_for_ltca_response = ec_mult(G,
 response_sk_for_ltca_response) in

new request_identifier:bitstring;
let dati_to_ltca = (request_identifier, encapsulate_ltca,
 PCRequestSharedContent, response_pk_for_ltca_response) in

let firma_dati_dall_its = sign(dati_to_ltca, pca_vsk) in

let cifra_firma_dati_aesccm = aead_enc(firma_dati_dall_its,
 aes_key_from_pca_to_LTCA, nonce_aes_ccm_from_pca_to_LTCA) in

let tag_firma_dati_aesccm =
 compute_tag(firma_dati_dall_its,cifra_firma_dati_aesccm,
 aes_key_from_pca_to_LTCA, nonce_aes_ccm_from_pca_to_LTCA) in

new eph_sk_from_pca_to_LTCA:secretkey;

let eph_pub_from_pca_to_LTCA = ec_mult(G,
 eph_sk_from_pca_to_LTCA) in

let shared_from_pca_to_LTCA = ec_mult(getpkenc(ltca_cert),
 eph_sk_from_pca_to_LTCA) in

let ke_pc_request = hkdf(shared_from_pca_to_LTCA, zero) in

let km_pc_request = hkdf(shared_from_pca_to_LTCA, one) in

let ciphertex_key_ecie_from_pca_to_LTCA =
 xor(aes_key_from_pca_to_LTCA, ke_pc_request) in

let tag_ecies_from_pca_to_LTCA = hmac(km_pc_request,
 ciphertex_key_ecie_from_pca_to_LTCA) in

event pc_request_validation();
```

```
let EncryptedData_PCA_LTCA = (hashId8(ltca_cert),
 cifra_firma_dati_aesccm,
 tag_firma_dati_aesccm, nonce_aes_ccm_from_pca_to_LTCA,
 ciphertex_key_ecie_from_pca_to_LTCA,
 tag_ecies_from_pca_to_LTCA, eph_pub_from_pca_to_LTCA  ) in

out(channel_ltca_pca, (0, EncryptedData_PCA_LTCA)  );

let compute_hash_request = hash((hashId8(ltca_cert),
 cifra_firma_dati_aesccm,
 tag_firma_dati_aesccm, nonce_aes_ccm_from_pca_to_LTCA,
 ciphertex_key_ecie_from_pca_to_LTCA,
 tag_ecies_from_pca_to_LTCA, eph_pub_from_pca_to_LTCA  )) in


in(channel_ltca_pca,
 (contenttype_from_ltca:nat,content_from_ltca:bitstring ));

if contenttype_from_ltca = 0 then(

let (=hashId8(pca_cert),
 ciphertex_ltca_aesccm_response:aead_bitstring,
 tag_ciphertex_dati_ltca_aesccm_response:tag_aead,
 nonce_aes_ccm_from_ltca_to_pca_response:nonce,
 ciphertex_key_ecie_from_ltca_to_pca_response:bitstring,
 tag_ecies_from_ltca_to_pca_response:bitstring,
 eph_pub_from_ltca_to_pca_response:point  ) = content_from_ltca
 in

let compute_shared_response =
 ec_mult(eph_pub_from_ltca_to_pca_response,
 response_sk_for_ltca_response) in

let ke_response = hkdf(compute_shared_response,zero) in
let km_response = hkdf(compute_shared_response,one) in

let tag_ecies_computed_response = hmac(km_response,
 ciphertex_key_ecie_from_ltca_to_pca_response) in

if tag_ecies_computed_response =
 tag_ecies_from_ltca_to_pca_response then

let aes_key_from_ltca =
 xor(ciphertex_key_ecie_from_ltca_to_pca_response, ke_response)
 in

let risposta_dalla_ltca =
 aead_dec(ciphertex_ltca_aesccm_response, aes_key_from_ltca,
 nonce_aes_ccm_from_ltca_to_pca_response ) in

let tag_firma_risposta_dalla_ltca_calcolato =
 compute_tag(risposta_dalla_ltca,
 ciphertex_ltca_aesccm_response,aes_key_from_ltca,
 nonce_aes_ccm_from_ltca_to_pca_response) in

if  tag_firma_risposta_dalla_ltca_calcolato =
 tag_ciphertex_dati_ltca_aesccm_response then

if checksign(risposta_dalla_ltca, getpksign(ltca_cert)) =
 ok_signature() then

let (=compute_hash_request, risultato:nat) =
 getmess(risposta_dalla_ltca) in

if risultato = 0 then

event pc_response_validation_received_from_ltca();

let pc_cert = issue_certificate(1, hashId8(pca_cert), 1,
 idits, pc_vpk_its, pc_epk_its, pca_vsk) in

let firma_risposta = sign((
 hash((hashId8(pca_cert), ciphertex_aesccm_its,
 tag_aesccm_its, nonce_aes_ccm_its,
 ciphertex_ecies_its,tag_ecies_its,eph_pub_its)),
0,
pc_cert
), pca_vsk) in

new new_aes_key:bitstring;
new new_nonce_aes:nonce;

let new_ciphertex_aesccm = aead_enc(firma_risposta,
 new_aes_key, new_nonce_aes) in

let new_tag_aesccm =
 compute_tag(firma_risposta,new_ciphertex_aesccm,
 new_aes_key, new_nonce_aes) in

new new_eph_sk_for_its:secretkey;

let new_eph_pk_for_its = ec_mult(G, new_eph_sk_for_its) in

let new_shared_for_its = ec_mult(rek_pk, new_eph_sk_for_its) in

let new_ke = hkdf(new_shared_for_its, zero) in
let new_km = hkdf(new_shared_for_its, one) in
let new_ciphertex_ecies = xor(new_aes_key, new_ke) in
let new_tag_ecies = hmac(new_km, new_ciphertex_ecies) in


let EncryptedData_PCA_ITS = (new_ciphertex_aesccm,
new_tag_aesccm, new_nonce_aes,
new_ciphertex_ecies, new_tag_ecies,
new_eph_pk_for_its ) in

event pc_sent_to_its();


out(channel_its_pca, (0, EncryptedData_PCA_ITS));
0
)else
0


)else
0

)else
0
.

let LTCA (id_cert: bitstring, ltca_cert:certificate,
 ltca_esk:secretkey, ltca_vsk:secretkey, pca_cert:certificate,
 rca_cert:root_certificate, ltc_cert:certificate) =

in(channel_ltca_pca, (externaldata:nat, content:bitstring));


if externaldata = 0 then (

let (=hashId8(ltca_cert), ciphertex_aesccm:aead_bitstring,
 tag_aesccm:tag_aead, nonce_aes_ccm:nonce,
 ciphertex_ecies:bitstring, tag_ecies:bitstring,
 eph_pk:point ) = content in

let compute_shared = ec_mult(eph_pk, ltca_esk) in

let ke = hkdf(compute_shared, zero) in
let km = hkdf(compute_shared, one) in

let tag_ecies_computed = hmac(km, ciphertex_ecies) in

if tag_ecies = tag_ecies_computed then
```

```
let aes_key = xor(ciphertex_ecies, ke) in
let firma_da_validare = aead_dec(ciphertex_aesccm, aes_key,
 nonce_aes_ccm ) in

let tag_firma_dati_ricevuti_aesccm =
 compute_tag(firma_da_validare, ciphertex_aesccm, aes_key,
 nonce_aes_ccm) in

if tag_firma_dati_ricevuti_aesccm = tag_aesccm then

if checksign(firma_da_validare, getpksign(pca_cert)) =
 ok_signature() then

event validation_pc_request_received_from_pca();


let ottieni_dati = getmess(firma_da_validare) in

let (requestId:bitstring, encapsulate_ltca:bitstring,
 PCRequestSharedContent_From_PCA:bitstring, rek_pca_key:point)
  = ottieni_dati in

let (=hashId8(ltca_cert), SSP:bitstring,
 tag_mac_chiavi:bitstring, rek_pk:point ) = PCRequestSharedContent_From_PCA i


let (ciphertex_aesccm_from_its:aead_bitstring,
 tag_aesccm_from_its:tag_aead, nonce_aes_ccm_from_its:nonce,
 ciphertex_key_ecies:bitstring, tag_its_ecies:bitstring, eph_pub_ltca:point) 

let new_compute_shared = ec_mult(eph_pub_ltca, ltca_esk) in

let ke_its = hkdf(new_compute_shared, zero) in

let km_its = hkdf(new_compute_shared, one) in

let tag_ecies_computed_req = hmac(km_its, ciphertex_key_ecies)
 in

if  tag_ecies_computed_req = tag_its_ecies then

let aes_key_its = xor(ciphertex_key_ecies, ke_its) in

let (firma_PCRequestSharedContent_LTCA:bitstring,
 firma_PCRequestSharedContent:bitstring) =
 aead_dec(ciphertex_aesccm_from_its, aes_key_its,
 nonce_aes_ccm_from_its) in

let tag_aesccm_computed_its =
 compute_tag((firma_PCRequestSharedContent_LTCA,
 firma_PCRequestSharedContent),ciphertex_aesccm_from_its,
 aes_key_its, nonce_aes_ccm_from_its) in

if tag_aesccm_computed_its = tag_aesccm_from_its then

let (PCRequestSharedContent:bitstring) =
 getmess(firma_PCRequestSharedContent) in

let (PCRequestSharedContent_for_LTCA:bitstring,
 =hashId8(ltc_cert)) =
 getmess(firma_PCRequestSharedContent_LTCA) in

if checksign(firma_PCRequestSharedContent_LTCA,
 getpksign(ltc_cert)) = ok_signature()
 && checksign(firma_PCRequestSharedContent, getpksign(ltc_cert))
 = ok_signature() then

event validation_PCRequestSharedContent_from_its();

let hash_request = hash((hashId8(ltca_cert), ciphertex_aesccm,
 tag_aesccm, nonce_aes_ccm, ciphertex_ecies, tag_ecies, eph_pk))
  in

let response = 0 in

let firma_ltca_pca = sign((hash_request, response), ltca_vsk) in

new eph_sk_ltca_pca:secretkey;

let eph_pub_ltca_pca = ec_mult(G, eph_sk_ltca_pca) in

let shared_ltca_pca = ec_mult(rek_pca_key, eph_sk_ltca_pca) in

let ke_ltca_pca = hkdf(shared_ltca_pca, zero) in

let km_ltca_pca = hkdf(shared_ltca_pca, one) in

new aes_ltca_pca:bitstring;

new nonce_aes_ccm_ltca_pca:nonce;

let ciphertex_key_ecies_ltca_pca = xor(aes_ltca_pca,
 ke_ltca_pca) in

let tag_ecies_ltca_pca = hmac(km_ltca_pca,
 ciphertex_key_ecies_ltca_pca) in

let ciphertex_aesccm_response = aead_enc(firma_ltca_pca,
 aes_ltca_pca , nonce_aes_ccm_ltca_pca ) in

let tag_aesccm_response =
 compute_tag(firma_ltca_pca,ciphertex_aesccm_response ,
 aes_ltca_pca , nonce_aes_ccm_ltca_pca) in

event pc_response_validation();

let EncryptedData_LTCA_PCA = (hashId8(pca_cert),
 ciphertex_aesccm_response , tag_aesccm_response ,
 nonce_aes_ccm_ltca_pca , ciphertex_key_ecies_ltca_pca
 ,tag_ecies_ltca_pca , eph_pub_ltca_pca) in

out(channel_ltca_pca , (0, EncryptedData_LTCA_PCA));

0
)else

event wrong_PCRequestreceived_from_its();
0


)else
0
.


let ITSSendPCRequest(ltc_esk:secretkey , ltc_vsk:secretkey ,
 ltca_cert:certificate , pca_cert:certificate , rca_cert:
  root_certificate , ltc_cert:certificate) =

new pc_vsk:secretkey;

let pc_vpk = ec_mult(G, pc_vsk) in

new pc_esk:secretkey;
let pc_epk = ec_mult(G, pc_esk) in

new rek_sk:secretkey;
let rek_pk = ec_mult(G, rek_sk) in

new hmac_key:bitstring;
let tag_mac_keys = hmac(hmac_key , (pc_vpk , pc_epk)) in

new SSP:bitstring;

let PCRequestSharedContent = (hashId8(ltca_cert), SSP,
 tag_mac_keys , rek_pk ) in
```

```
let PCRequestSharedContent_LTCA = (PCRequestSharedContent,
 hashId8(ltc_cert)) in

let PCRequestSharedContent_signed = sign(PCRequestSharedContent,
 ltc_vsk) in

let PCRequestSharedContent_LTCA_signed =
 sign(PCRequestSharedContent_LTCA, ltc_vsk) in

new ase_key_ltca:bitstring;
new nonce_aes_ccm_for_ltca:nonce;

let ciphertex_aesccm_for_ltca =
 aead_enc((PCRequestSharedContent_LTCA_signed,
 PCRequestSharedContent_signed), ase_key_ltca,
 nonce_aes_ccm_for_ltca ) in

let tag_aesccm_for_ltca =
 compute_tag((PCRequestSharedContent_LTCA_signed,
  PCRequestSharedContent_signed), ciphertex_aesccm_for_ltca,
  ase_key_ltca, nonce_aes_ccm_for_ltca) in


new eph_sk_from_its_to_ltca:secretkey;

let eph_pub_ltca = ec_mult(G, eph_sk_from_its_to_ltca) in

let shared = ec_mult(getpkenc(ltca_cert),
 eph_sk_from_its_to_ltca) in

let ke = hkdf(shared, zero) in

let km = hkdf(shared, one) in

let ciphertex_key_ecies = xor(ase_key_ltca, ke) in
let tag_key_ecies = hmac(km, ciphertex_key_ecies) in

new its_id:bitstring;
let encapsulate_ltca = (ciphertex_aesccm_for_ltca,
 tag_aesccm_for_ltca, nonce_aes_ccm_for_ltca, ciphertex_key_ecies, tag_key_eci

let request_pca = (pc_vpk, pc_epk, encapsulate_ltca,
 PCRequestSharedContent, hmac_key, its_id) in

new aes_key_pca:bitstring;
new nonce_aes_ccm_for_pca:nonce;

let ciph_aesccm_for_pca = aead_enc(request_pca, aes_key_pca,
 nonce_aes_ccm_for_pca ) in
```

```
let tg_aesccm_for_pca = compute_tag(request_pca,
 ciph_aesccm_for_pca, aes_key_pca, nonce_aes_ccm_for_pca) in

new eph_sk_for_pca:secretkey;

let eph_pk_for_pca = ec_mult(G, eph_sk_for_pca) in

let new_shared = ec_mult(getpkenc(pca_cert), eph_sk_for_pca) in

let ke_for_pca = hkdf(new_shared, zero) in

let km_for_pca = hkdf(new_shared, one) in

let ciph_ecies_for_pca = xor(aes_key_pca, ke_for_pca) in

let tg_ecies_for_pca = hmac(km_for_pca, ciph_ecies_for_pca) in

event pc_request_sent_from_its();

let EncryptedData =  (hashId8(pca_cert), ciph_aesccm_for_pca,
 tg_aesccm_for_pca, nonce_aes_ccm_for_pca, ciph_ecies_for_pca,
 tg_ecies_for_pca, eph_pk_for_pca) in

out(channel_its_pca, (0, EncryptedData) );

let computed_hash = hash((hashId8(pca_cert),
 ciph_aesccm_for_pca, tg_aesccm_for_pca, nonce_aes_ccm_for_pca,
 ciph_ecies_for_pca, tg_ecies_for_pca, eph_pk_for_pca)) in

in(channel_its_pca, (externaldata:nat, content:bitstring));

if externaldata = 0 then (

let (ciph_aesccm_from_pca:aead_bitstring,
 tg_aesccm_from_pca:tag_aead,
 nonce_aes_ccm_from_pca_to_its:nonce,
 ciph_ecies_from_pca:bitstring, tg_ecies_from_pca:bitstring,
 eph_pk_from_pca:point  ) = content in

let response_shared = ec_mult(eph_pk_from_pca, rek_sk) in

let new_ke = hkdf(response_shared, zero) in

let new_km = hkdf(response_shared, one) in

let new_tag_ecies_computed = hmac(new_km, ciph_ecies_from_pca)
 in
if new_tag_ecies_computed = tg_ecies_from_pca then

let aes_key_recovered = xor(ciph_ecies_from_pca, new_ke) in

let new_firma = aead_dec(ciph_aesccm_from_pca,
 aes_key_recovered, nonce_aes_ccm_from_pca_to_its) in

let new_tag_aesccm_computed =
 compute_tag(new_firma,ciph_aesccm_from_pca, aes_key_recovered,
 nonce_aes_ccm_from_pca_to_its) in

if new_tag_aesccm_computed = tg_aesccm_from_pca then

if checksign(new_firma, getpksign(pca_cert)) = ok_signature()
 then

let new_message = getmess(new_firma) in

let (=computed_hash,response_code:nat, pc_cert:certificate) =
 new_message in

if response_code = 0 && checkcert(pc_cert, getpksign(pca_cert))
 = ok() && gettypecert(pc_cert) = 1 then

event pc_received_from_pca();

0
)else
0.


process

(**)
new rca_sign_sk:secretkey;
let rca_sign_pk = ec_mult(G, rca_sign_sk) in

new rca_enc_sk:secretkey;
let rca_enc_pk = ec_mult(G, rca_enc_sk) in

let rca_cert = issue_rootcertificate(0, 4, rca_sign_pk,
 rca_enc_pk, rca_sign_sk ) in

let rca_hashid8 = hashId8Root(rca_cert) in


(**)
new idpca:bitstring;
new pca_vsk:secretkey;
let pca_vpk:point = ec_mult(G, pca_vsk) in

new pca_esk:secretkey;
let pca_epk:point = ec_mult(G, pca_esk) in


let pca_cert = issue_certificate(1, rca_hashid8, 2, idpca,
 pca_vpk, pca_epk, rca_sign_sk) in

(**)
new idltca:bitstring;

new ltca_vsk:secretkey;
let ltca_vpk:point = ec_mult(G, ltca_vsk) in

new ltca_esk:secretkey;
let ltca_epk:point = ec_mult(G, ltca_esk) in

let ltca_cert = issue_certificate(1, rca_hashid8, 3, idltca,
 ltca_vpk, ltca_epk, rca_sign_sk) in
 let ltca_cert_id = hashId8(ltca_cert) in


(**)
new ltc_vsk:secretkey;
let ltc_vpk = ec_mult(G, ltc_vsk) in

new ltc_esk:secretkey;
let ltc_epk = ec_mult(G, ltc_esk) in

let ltc_cert = issue_certificate(1, ltca_cert_id, 0, idltca,
 ltc_vpk, ltc_epk, ltca_vsk) in

(PCA(ltca_cert,pca_cert, pca_vpk, pca_vsk, pca_epk, pca_esk,
 rca_cert) |

ITSSendPCRequest(ltc_esk, ltc_vsk, ltca_cert, pca_cert,
 rca_cert, ltc_cert) |

LTCA(idltca, ltca_cert, ltca_esk, ltca_vsk, pca_cert, rca_cert,
 ltc_cert)  )
```