Project acronym:	EVITA
Project title:	E-safety vehicle intrusion protected applications
Project reference:	224275
Program:	Seventh Research Framework Program (2007–2013) of the European Community
Objective:	ICT-2007.6.2: ICT for cooperative systems
Contract type:	Collaborative project
Start date of project:	1 July 2008
Duration:	36 months

Deliverable D3.4.3: On-Board Architecture and Protocols Verification

Authors:	Andreas Fuchs, Sigrid Gürgens (Fraunhofer Institute SIT); Ludovic Apvrille, Gabriel Pedroza (Institut Télécom)
Reviewers:	Olaf Henniger (Fraunhofer Institute SIT)

Dissemination level:	Public
Deliverable type:	Report
Version:	1.0
Submission date:	30 December 2010

Abstract

The objective of the EVITA project is to design, verify, and prototype an architecture for automotive on-board networks where security-relevant components are protected against tampering, and sensitive data are protected against compromise. Thus, EVITA will provide a basis for the secure deployment of electronic safety aids based on vehicle-to-vehicle and vehicle-to-infrastructure communication. Security will cover various aspects such as dependability, integrity, authenticity, or even privacy. In order to introduce security issues into the product life cycle in an early stage of the process, model oriented approaches must be adjusted to take into account the security mechanisms. This document provides the results achieved when verifying the EVITA architecture and protocols with respect to the security requirements for automotive on-board networks.

Terms of use

This document was developed within the EVITA project (see http://evita-project.org), co-funded by the European Commission within the Seventh Framework Programme (FP7), by a consortium consisting of a car manufacturer, automotive suppliers, security experts, hardware and software experts as well as legal experts. The EVITA partners are

- BMW Research and Technology,
- Continental Teves AG & Co. oHG,
- escrypt GmbH,
- EURECOM,
- Fraunhofer Institute for Secure Information Technology,
- Fraunhofer Institute for Systems and Innovation Research,
- Fujitsu Services AB,
- Infineon Technologies AG,
- Institut Télécom,
- Katholieke Universiteit Leuven,
- MIRA Ltd.,
- Robert Bosch GmbH and
- TRIALOG.

Contents

List of figures

List of tables

List of abbreviations

APP	Application
BCU	Brake Control Unit
CADP	Construction and Analysis of Distributed Processes
CPU	Central Processing Unit
CSC	Chassis & Safety Controller
CTL	Computational tree logic
DIPLODOCUS	DesIgn sPace expLoration based on fOrmal Description teChniques, Uml and Systemc
ES	Environment Sensor
ECU	Electronic Control Unit
F-SeBB	Formal Security Building Block
M-SeBB	Mechanism Security Building Block
HMAC	Hash-Based Message Authentication Code
HSM	Hardware Security Module
LOTOS	Language Of Temporal Order Specification
MAC	Message Authentication Code
OMG	Object Management Group
RAM	Random Access Memory
RT-LOTOS	Real-Time Language Of Temporal Order Specification
SeBB	Security Building Block
SeMF	Security Modeling Framework
SysML	System Modeling Language
TIF	TURTLE Intermediate Format
ToV	Target of Verification
TTool	TURTLE Toolkit
TURTLE	Timed UML and RT-LOTOS Environment
UML	Unified Modeling Language
UPPAAL	Formal verification toolkit from UPPsala and AALborg universities

Document history

Version	Date	Changes
1.0	30 December 2010	First issue of deliverable

1 Introduction

1.1 Objectives of the EVITA Project

The objective of the EVITA project is to design, verify, and prototype an architecture for automotive on-board networks where security-relevant components are protected against tampering, and sensitive data are protected against compromise. Thus, EVITA will provide a basis for the secure deployment of electronic safety aids based on vehicle-to-vehicle and vehicle-to-infrastructure communication.

Security will cover various aspects such as dependability, integrity, authenticity, or even privacy. In order to introduce security issues into the product life cycle in an early stage of the process, model oriented approaches must be adjusted to take into account the security mechanisms. In EVITA we have first identified security requirements (see [17]) and have then designed an architecture (see [19]) and the security protocols to be used which are introduced in [18].

1.2 Scope and Outline of the Document

This document is concerned with the security analysis of architecture and protocols designed in EVITA with respect to the security requirements. This document gives a brief summary of our approach and presents the analysis results. Deliverable D3.4.4 will then formalize the attacks identified in [17] and analyze whether they can be prevented through the EVITA security architecture according to its analysis in this document.

In Section 2 we give a brief summary of our verification approaches. We further discuss limitations of these approaches and issues (such as attacks) mentioned in [19] which we do not take into account because of not being relevant for EVITA. Section 3 then provides the verification results concerning the security protocols to be used in EVITA, while Section 4 discusses the verification of the overall EVITA architecture, building on the protocol verification results.

2 The Approach

2.1 Overview

One category of security requirements listed in [17] is of a type that cannot be met by a single security mechanism. An example is the driver of a car needing some message to be authentically originated by a sensor of another car (requirement *Authenticity_5* of [17]). The driver, a human being, cannot verify digital signatures or MACs. Another example is a device not being able to execute a mechanism (e.g. because of restricted resources) that is needed to meet a specific requirement. In [17] for example, we have identified a non-repudiation requirement, namely that the eTolling-Service Provider shall be able to prove the authenticity of the Billing-Information, based on the aggregated sensor data (requirement *Proof-of-Authenticity_1* of [17]). However, the sensor is not able to perform digital signatures, a mechanism usually applied for non-repudiation purposes. We call this type of security requirements *global composition requirements*.

An approach particularly suitable to verify that the EVITA architecture and protocols meet this type of requirements is to use the Security Building Blocks (SeBBs) introduced in [13]. In that we have used SeBBs to refine abstract requirements identified in [17] towards more concrete requirements that can be met by the EVITA architecture and protocols. In the current task however, we use the SeBBs in the reverse way. Based on the assumptions on both the EVITA Hardware Security Module (HSM) described in [19] and the protocols described in [18], the SeBBs either enable proofs that the global composition requirements listed in [17] indeed are met or point to assumptions that are needed to carry out a proof but that can not adequately be assumed to hold.

Another category of requirements listed in [17] is related to aspects of time and availability. Examples are *Freshness_103* and *Availability_108*. The first example refers to the case when the flashing of an ECU is being performed: an old flashing command should not be sent to a given ECU (replay attack). The second example refers to the maximum response time of the braking command whenever a driver or another vehicle trigger that command. Since the SeBBs available so far are not useful for modeling and proving such requirements, another more appropriate verification framework is used here. While this approach is not restricted to a particular level of abstraction and can therefore in principle also handle global composition requirements, once the need arises to include several different mechanisms and protocols, we are faced with the well-known state space explosion problem. Hence, when verifying that the EVITA architecture and protocols meet time and availability related requirements, we focus on particular parts of mechanisms and protocols, specifying them in a less abstract way. This approach is called *bounded magnifying view*.

Both approaches complement each other very well: Indeed, the *bounded magnifying view* focusses more on specific parts of the EVITA system, while its verification results are used by the *global composition approach* to ensure that the overall EVITA architecture – e.g. the HSM – and protocols meet the security requirements.

In Sections 2.2 and 2.3, we give a brief summary of and extensions to the bounded magnifying view approach and the global composition approach, respectively.

2.2 The Magnifying View Approach

2.2.1 Introduction

The magnifying view approach targets the verification of security properties over subsystems of the EVITA architecture. A subsystem might be limited to a subset of ECUs, and within ECUs, to a subset of hardware elements: CPUs, memories, HSM, sensors / actuators, etc. Also, for a given subset, only a limited number of functional elements might be considered, as well as a subset of cryptographic protocols: this is why this approach is named *bounded magnifying view*. Thus, cryptographic protocols are modeled independently from the use cases for which those protocols have been conceived. Additionally, some hardware constraints might be taken into account at this modeling level.

2.2.2 Problematics

For modeling and verifying embedded systems, we have already defined two UML profiles, TURTLE[1] [5] and DIPLODOCUS[2][6]. The former targets time-constrained systems. The latter is more focused on software / hardware partitioning, and thus explicitly takes into account hardware constraints. Both have been formally defined, and as a consequence, TURTLE or DIPLODOCUS models can be formally verified at the push of a button using the TTool environment [2]. Unfortunately, neither TURTLE nor DIPLODOCUS have been conceived for the formal verification of security properties, that is, those security properties cannot be directly entered in a model (for example, using a "confidentiality" keyword), but have to be explicitly modeled using specific attacker and observers made "by hand". However, whether we select TURTLE or DIPLODOCUS, our first task is to define a way to model system elements involved in the EVITA architecture and protocols with security requirements in mind [17], and to propose a formal verification framework based on those models. TURTLE has first been selected for formal verification purpose, and DIPLODOCUS has been selected for first performance and verification [17].

2.2.3 Verification methodology

Our verification process is based on a model-driven approach (see Figure 1). Inputs are considered to build up a model from which automatic code generation can be implemented. In the scope of formal verification, code generation targets the automatic generation of a formal specification from which formal proofs of security properties can be performed. Security properties are defined according to requirements previously identified in [17].

This overall formal verification methodology takes three different forms: *TURTLE*, *ProVerif*, and *AVATAR*[3].

1. In the **TURTLE** approach, the subsystem of interest is modeled as a set of TURTLE class and activity diagrams, using the TTool toolkit (see Figure 2). EVITA Requirements were already modeled in TTool as a set of SysML Requirement Diagrams. Those requirements are meant to be integrated into the system and modeled

[1] Timed UMLand RT-LOTOS environment
[2] DesIgn sPace exLoration based on fOrmal Description teChniques, Uml and Systemc
[3] Automated Verification of reAl Time softwARe

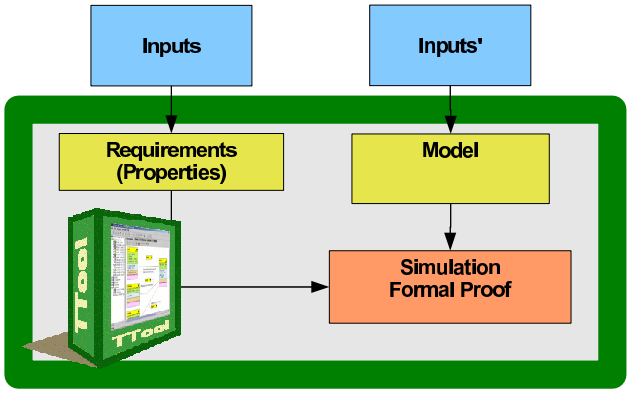


Figure 1 Model-driven approach

as generic attacker and observer models. More precisely, generic attackers can be modeled as specific class of the class diagrams. This class can intercept all messages on communications links, can alter those messages, and can inject resulting messages back to the communication links (Dolev-Yao approach). Other security properties (e.g., freshness) can also be modeled using TURTLE observers that can observe differences of time between two given actions in the model (e.g., the sending of a message and the receiving of that message). In the TURTLE approach, formal proofs are conducted on the system model using the translation capabilities of TTool (e.g., translation to LOTOS [16] and UPPAAL [7]).

Drawbacks of TURTLE concern the fact that the Dolev-Yao model considers all possible value modification in messages: but in TURTLE, data are modeled as numbers in a given range, and therefore, the exploration we can make on message alteration is limited. Also, TURTLE considers only a limited number of instances in systems (e.g., only a given number of ECUs), and so, proof results are limited to the number of instances we were able to model in the system. Finally, above mentioned drawbacks lead us to consider two other options (ProVerif, AVATAR) that are further described.

2. **ProVerif** is a toolkit based on spi-calculus processes that are further translated into Horn clauses for proof purpose. Only confidentiality properties can directly be expressed using a query language (and more precisely, the *secret* query), but other properties (e.g. authenticity) can be expressed using queries on relations between events. The strength of this approach relies in its capability to model data within an infinite value range, that is, all possible message alterations can be considered in the Dolev-Yao model. Additionally, proofs can be conducted for an infinite number of process instances. Last, but not least, Horn clauses resolution is automated, i.e., proofs are conducted automatically.

   The obvious drawback of that approach is that security proofs are not performed from a high-level model, and so, transformations must be made by hand (see Figure 3). Also, the spi-calculus model can not be reused for other methodological EVITA

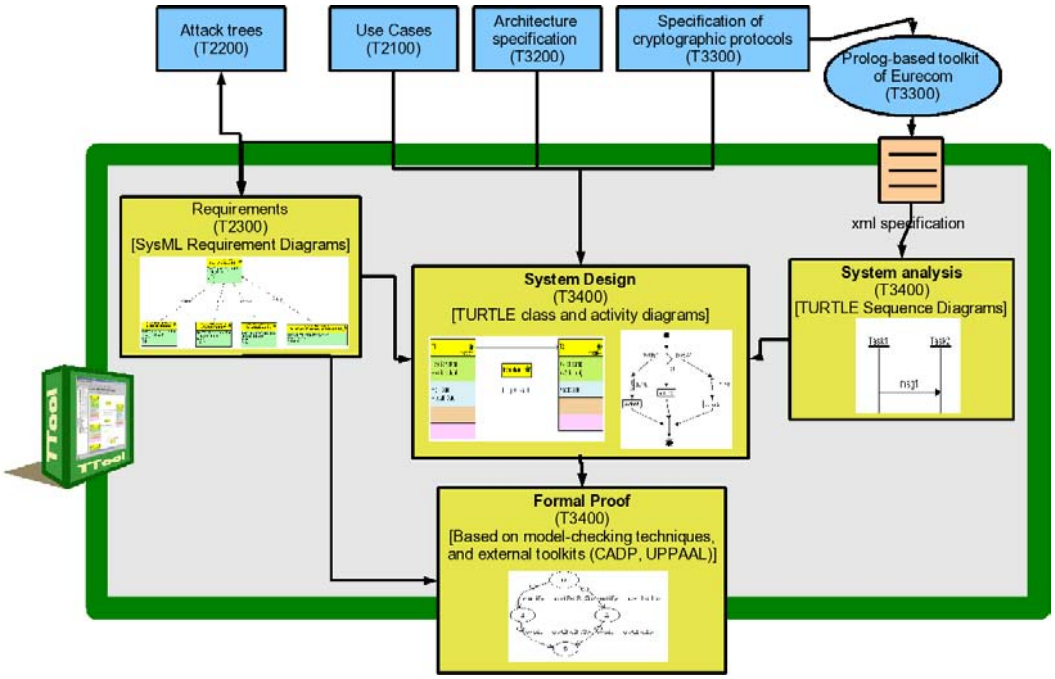


Figure 2 The TURTLE verification approach

steps, e.g., for performance evaluation, and for executable code generation. Those drawbacks lead us to consider a third solution: developing a new UML profile able to capture embedded systems in a way that formal security proofs can be achieved directly from the model – with little or no modification – and from which performance evaluation and executable code generation can be executed. Thus, in the scope of EVITA, we defined a new profile, named **AVATAR**, based on SysML – itself a UML profile.

3. **AVATAR** is a SysML profile, fully supported by TTool. It targets the modeling of embedded systems with time and security constraints. In AVATAR, a system is modeled as a set of communicating SysML blocks. The behavior of each block is described using SysML state machines, in which time constraints can be expressed. Security constraints can be expressed directly on the block diagram, in an attached note, using e.g. keywords such as "*confidentiality*" and "*authenticity*": no specific modeling (e.g., model of an attacker) is thus necessary in AVATAR.

   Moreover, from an AVATAR model (i.e., from a block diagram and a set of state machine diagrams), a formal specification can be derived in UPPAAL and in spi-calculus. From TTool, a press-button approach is implemented so as to automatically derive those specifications and inject them in the corresponding underlying toolkits: UPPAAL, and ProVerif, respectively (see Figure 4). Thus, very different security properties such as freshness properties and confidentiality properties can be proved from the same model. Furthermore, executable code can be automatically generated from AVATAR models. Finally, AVATAR addresses all drawbacks

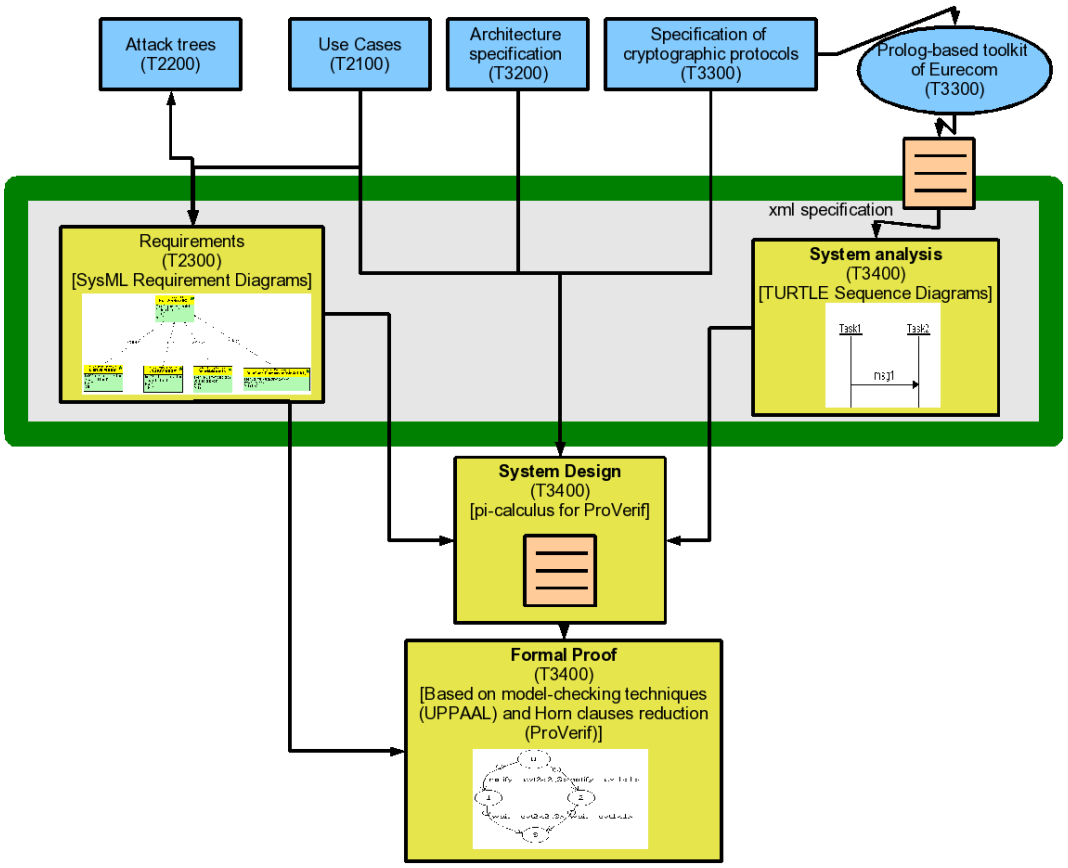


Figure 3 The ProVerif verification approach

mentioned in the two first approaches, that is drawbacks of the TURTLE approach, and drawbacks of the ProVerif approach.

Table 1 explains those three approaches, and discusses the drawbacks and advantages of each of them.

2.3 The Global Composition Approach

The approach for the verification of the EVITA architecture and protocols with respect to satisfying global composition security requirements is based on the Fraunhofer SIT Security Modeling Framework SeMF already introduced in [13]. This framework allows to define the formal model of a system, to formally specify abstract security requirements to be met by the system in terms of this model, and to refine these security requirements by way of so-called *Security Building Blocks* (SeBBs). A SeBB represents an implication stating that certain internal security properties holding in a system imply that certain external security properties hold as well. There are two different types of SeBBs: Formal Security Building Blocks, called *F-SeBBs*, and Mechanism Security Building Blocks called M-SeBBs. F-SeBBs originate from the definitions of the security properties themselves. They reflect theorems that can be proven within SeMF independently of any specific system, based only on the assumptions that are introduced by the internal properties.

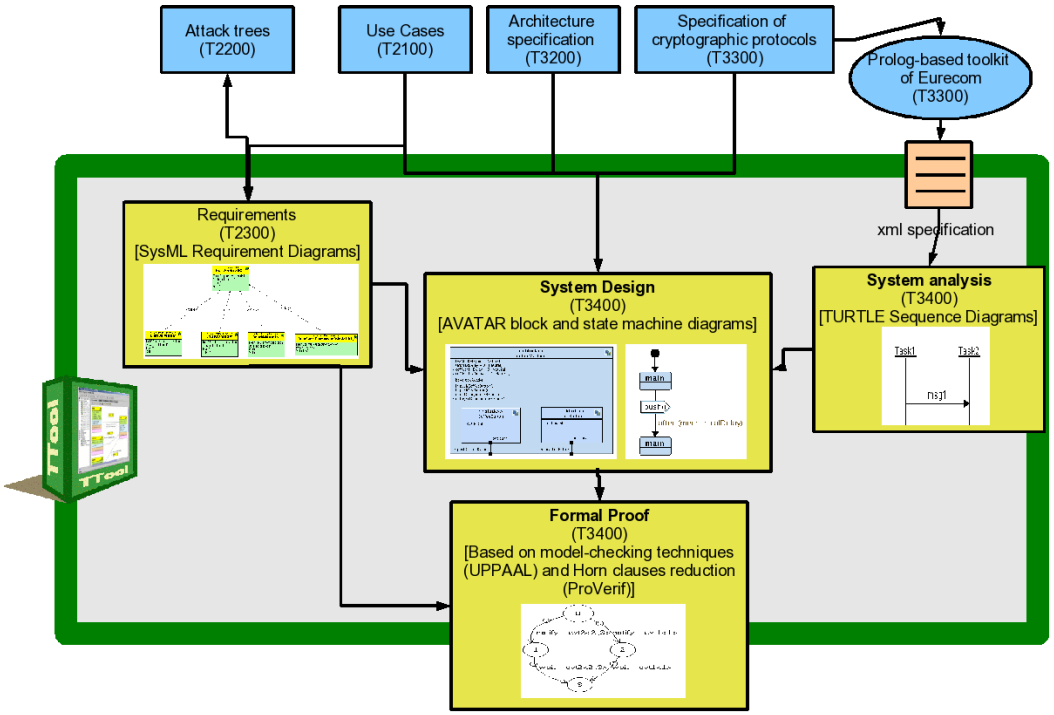


Figure 4 The AVATAR verification approach

M-SeBBs on the other hand represent security primitives. These can be simple security mechanisms (e.g. the generation of a digital signature) or more complicated security protocols. Proofs of M-SeBBs require assumptions that originate from expert knowledge that is external to SeMF.

A simple F-SeBB for example states that for actions a, b, c of a system, $precede(a, c)$ and $precede(c, b)$ holding in the system implies that $precede(a, b)$ holds as well (see [13] for the formal definition of the property $precede$). Another, more complicated F-SeBB states that $trust(P, precede(a, x)$ and $auth(x, b, P)$ holding in a system S implies that $auth(a, b, P)$ holds in S as well. In [13] we have provided a set of F-SeBBs that are used in Section 4 for our verification.

M-SeBBs on the other hand reflect the external properties a security mechanism provides assuming that a set of internal properties is satisfied. These SeBBs constitute inference rules that capture expert knowledge within the area of cryptanalysis. The HMAC SeBB for example states that the external property $precede(sign(P, msg, SS, sig), verify(Q, msg, sig, SS))$ holds if the internal properties $conf(\mathcal{A}(SS), SS, \mathcal{V}(SS), \mathcal{L}, \{P, Q\})$ and $not\text{-}precede(sign(Q, msg, SS, sig), verify(Q, msg, sig, SS))$ are satisfied. This simply states that an HMAC verification using some shared secret is always preceded by the respective HMAC sign operation, provided that the shared secret is indeed only known by the signer and the verifier, and provided that the verifier did not perform the signing itself. M-SeBBs can also capture the nature of security protocols: While the assumptions that are used for the protocol verification serve as internal properties, the external properties are those that are provided by the protocol.

In [13] we have used a set of F- and M-SeBBs to refine abstract security requirements towards low level assumptions. This refinement process can be part of a Security Engineering Process. In the verification described in this Section we use SeBBs in the reverse way: We identify all assumptions that are made in the course of designing the EVITA architecture (e.g. the assumption that all secret keys used by the Hardware Security Modules HSMs included in the cars' ECUs are confidential). We further model the protocols described in [18] as M-SeBBs and use the results of the magnifying view approach described in Section 3 verifying these protocols (i.e. use the assumptions as internal properties and the properties provided by the protocol as external properties of the M-SeBB, respectively). For each of the global security requirements we then search for a way to repeatedly apply appropriate SeBBs leading from a subset of the assumptions on architecture and protocol properties to the global security requirement. The subset of assumptions serves as internal properties for the first round of SeBB application which produces a set of external properties. These are taken as internal properties for the next application of SeBBs, etc., until the last step in which the desired global security requirement is (one of) the external propertie(s) of the applied SeBB. The process includes the following steps:

1. Identify a security requirement R that shall be proven.
2. Identify the assumptions on architecture and protocols to be a starting set of internal properties.
3. Search for a SeBB that uses some of these assumptions as internal properties.
4. If R is not one of the external properties of the identified SeBB: add the external properties to the set of internal properties then go back to step 3.
5. If R is contained in the external properties of the SeBB: finished.
6. If no more SeBBs can be applied, there is a problem (some assumption missing, probably caused by a missing application of security mechanisms).

The establishment of a path from assumptions to the global composition security requirement constitutes the proof that the system that satisfies these assumptions indeed provides the respective security property. Note that there may be more than one way to achieve such a proof.

Table 1 Formal verification approaches

Category	TURTLE	ProVerif	AVATAR
Model of the system	UML class and activity diagrams	pi-calculus and horn clauses	SysML Block and State Machine Diagrams
Model of the attacker	Simplified Dolev-Yao model (but freshness can be proved!)	Dolev-Yao	Dolev-Yao
Way to model the attacker	Specific Attacker class in the class diagram. Also, the behavior of this class must be provided	No need to model the attackerQueries	No need to model the attacker
Way to model security properties	Specific Observer class declared in the class diagram. Also, the behavior of this class must be provided	Queries	Basic pragmas listed in a note of the Block Diagram
Underlying proof technique	Model-checking. More precisely, a specific action of the Attacker or of the Observer class is searched for	Horn-clauses resolution: searching whether a given event is reachable, and whether a data can be accessed by an attacker	Horn-clauses resolution: searching whether a given event is reachable, and whether a data can be accessed by an attacker.
Toolkits used for modeling purpose	TTool	Text editor	TTool
Toolkits used for the proof of security properties	CADP, UPPAAL	ProVerif	ProVerif

3 Magnified Verification Results

This sections presents the verification results we obtained using the (1) TURTLE, (2) ProVerif and (3) AVATAR approaches. The verification of one given property is called a Target of Verification (ToV). At first, we define the notion of ToV. Then, a synthesis about verification results is provided. At last, ToVs are described in a more detailed way, including assumptions and models.

3.1 Target of Verification (ToV)

3.1.1 Definition

A ToV is defined as a 7-uple $tov = (P, Pr, A, M, T, O, R)$, with:

- P, a protocol defined in [18].
- Pr, a security property.
  $P \in \{Confidentiality, Authenticity, Integrity, Freshness, Availability\}$.To be compliant with previous EVITA technical reports, we consider the informal definitions already presented in [17], Section 2, as our basic reference.
- A, a set of assumptions which applies to this particular ToV. All assumptions Λ that applies to a ToV are thus given as $\Lambda = A \bigcup G$.
- M, a model relying either on the TURTLE, the ProVerif or the AVATAR approach. M contains only a model of the protocol under evaluation, and definitely not an attacker model, or an observer model of security properties under evaluation. In TURTLE, M is a class diagram and a set of activity diagrams. In AVATAR, M is a SysML / AVATAR Block Diagram and a set of States Machine Diagrams. In ProVerif, M is a spi-calculus specification. For short, we can say that $M \in \{TURTLE, AVATAR, ProVerif\}$.
- T, a model of attacker. That model generally represents a Dolev-Yao attacker.
- O, an observation technique, for observing whether Pr is satisfied, or not. For example, in TURTLE, an observer is a specific class of the TURTLE class model, the activity diagram of that class, a set of synchronized actions added to observed class – both at class diagram and at activity diagrams level – and a set of CTL formulae to search whether specific actions of observers can be reached, or not. In AVATAR, the observation model is implicit once security properties to be studied are declared in the Block Diagram, i.e., queries are automatically generated in ProVerif format from the Block Diagram. In ProVerif, the observation technique is based on a set of ProVerif *queries*.
- R, the verification result. $R \in \{Satisfied, Failed\}$.

3.1.2 General assumptions G

G represents assumptions that apply to all ToVs. G contains the following assumptions:

- The model M of the ToV is assumed to represent all necessary elements of P for proving Pr. Yet, M remains a model of implementation.
- We assume that toolkits that are used for modeling, generating code, and proving properties are free of bugs that could affect the verification result.
- All cryptographic functions, and the material associated to these functions, are assumed to be totally secured. In particular:
  - When a random number is generated, its value is not supposed to be repeated.
  - Whenever a MAC is generated for a message $m1$, it is impossible to find another message $m2 \neq m1$ such that $MAC(m1) = MAC(m2)$.
  - Whenever a signature is generated for a message $m1$, it is impossible to find another message $m2 \neq m1$ such that $Sign(m1) = Sign(m2)$.
  - Certificates are signed with secret material and thus can not be forged.
  - The components of the system are synchronized in time, and that synchronization is assumed to be secured.
  - For symmetric cryptography, if $m2 = encrypt(m1, k1)$, it is impossible, to obtain $m1$ from $m2$ without $k1$.
  - The value of a private key can not be guessed by knowing its public key pair.
  - The values of private keys (symmetric or asymmetric) can not be guessed.
  - Secret material can not be guessed.

As a conclusion of those assumptions, our verification approach relies on the verification of security properties on an implementation model, and so, it does not address formal proofs for computational security, e.g., relying on *certicrypt* or *cryptoverif*.

3.2 Results overview

ToVs are listed in Table 2. Assumptions of ToVs have been ignored here, they are provided in the more refined description of these ToVs. A "Satisfied" followed with a "*" means that the property is satisfied, but the verification phase returns traces of attacks that have been analyzed by hand to say whether they correspond to real attacks, or to model limitations. Section 3.9 elaborates on this.

We now provide more refined explanations of the three verification approaches (i.e., *TURTLE, ProVerif* and *AVATAR*) first described in section 2. An emphasis is put on the verification context, in order to clarify what the verification results really mean. The description of each approach is organized as follows:

1. A description of the modeling approach.
2. A description of security properties representation

Table 2 Synthesis of ToVs

Protocol	Property	General approach	Verification result
Key Master	DoS	TURTLE	Failed
Key Master (modified version)	Freshness	TURTLE	Satisfied
Key Master (modified version)	Integrity	TURTLE	Satisfied
Key Master (modified version)	Authenticity	TURTLE	Satisfied
Key Master (modified version)	Confidentiality	ProVerif	Satisfied
Key Master (modified version)	Authenticity	ProVerif	Satisfied*
Key Master (modified version)	Confidentiality	AVATAR	Satisfied
Key Master (modified version)	Authenticity	AVATAR	Satisfied*
Flashing	Confidentiality	ProVerif	Satisfied
Flashing	Authenticity	ProVerif	Satisfied*
CAM-LDW	Confidentiality	ProVerif	Satisfied
CAM-LDW	Authenticity	ProVerif	Satisfied*

3. An overview of the attacker representation
4. An explanation about the verification process
5. An overview of the main limitations

3.3 TURTLE Verification Approach Overview

3.3.1 System Modeling

Following the EVITA specification, a protocol is determined by the exchanges between a set of Communicating Entities (CE's). Thus, each CE is clearly delimited by a common physical, logical or virtual border inside of which internal operations are performed, and from which messages can be sent and received. The representation of CE's in TURTLE therefore considers internal operations as well as external message receiving and sending. Each CE in the protocol is represented by a TURTLE class which is defined by its name, a *list of attributes*, a *list of communication gates* and an *Activity Diagram*. TURTLE classes are defined within a TURTLE Class Diagram (TCD), which is itself a subpart of a TURTLE design (TD).

The TURTLE Design profile extends two diagrams of the UML2 specification: Class Diagrams which describe the static architecture of the system under design, and activity diagrams which describe the internal behavior of active classes. More precisely, a TURTLE class diagram is made up of stereotyped classes that we call *Tclasses*. The internal behavior of each Tclass must be described using an Activity Diagram. TURTLE classes relations are based on composition operators borrowed from LOTOS. TURTLE also borrows from LOTOS the notion of gates that in TURTLE is a mean for inter or intra classes communication. TURTLE composition operators are meant to attribute associations between two Tclasses, so as to provide those associations with a formal semantics. Composition operators are:

- **Parallel**. Used when the two related classes run in pure interleaving.
- **Synchronization**. Means that the two connected classes communicate throughout *gates*. Gates are attributes of Tclasses. When a synchronization between two Tclasses occurs, data may be exchanged between those two classes. In fact, the synchronization semantics is the one of LOTOS [16].
- **Invocation**. One gate plays the role of the *caller* gate, whereas the other one plays the role of the *callee* gate. When a synchronization occurs between the two caller and callee gates, values can be exchanged only from the caller to the callee. Then, the activity of the caller is blocked until the activity of the callee makes a new call to its callee gate. During that second synchronization, values can be exchanged only from the callee to the caller. More generally, this operator may be used to model method calls (i.e., function calls).
- **Sequence**. *C1 Sequence C2*: each time a new instance of C1 completes, a new instance of C2 is executed.
- **Preemption**. *C1 preempts C2*: as soon as an instance of C2 can execute an action on a gate, an instance of C1 is stopped forever.

TURTLE Activity Diagrams extend UML2 Activity Diagrams with logical and temporal operators. Modified / added logical operators are:

- **Action on a gate**. That gate may be synchronized with another one, or not. If it is not synchronized, the action can be performed immediately. Otherwise, if it is synchronized with another gate[4]: when an activity reaches such a gate, the activity is blocked until the synchronization can be performed.
- **Fork with synchronization**. A synchronization between two gates of two different sub-activities can be defined using the *Fork* operator with an extra annotation describing a synchronization scheme (i.e., a synchronization relation between two gates).
- **Join**. The join operator of UML Activity Diagrams has a semantics based on a TURTLE synchronization between all joining sub-activities.
- **Choice**. The TURTLE choice operator supports non-deterministic guards, i.e., guards of sub-activities can be left empty. When an activity reaches a TURTLE choice, all sub-activities starting from that choice and for which the guard evaluates to true (note: an empty guard means "true") are started in parallel: the first sub-activity performing an action on a gate preempts all others.

TURTLE Temporal operators are:

- **Deterministic delay**.
- **Non-deterministic delay** (i.e., latency).

[4]TURTLE supports only 2-gate synchronization schemes

- **Time interval**. It is the combination of a deterministic delay with a non-deterministic one.
- **Time-limited action on a gate** g. From that operator, two sub-activities may be linked: one executed when the action on g can be performed before a given delay, and one executed when the action on g could not be performed before the given delay.

However, because TURTLE lacks a time capture operator, i.e., a way to store in a variable the current value of the clock, we were not able to use those operators for freshness proof purpose. Instead, another approach, based on the explicit modeling of a clock, was used. Last but not least, a TURTLE Design (i.e., a TCD and all related Activity Diagrams) can be translated into a LOTOS specification, and into a set of communicating automata (UPPAAL). TTool implements a press-button approach for that purpose.

In TURTLE classes, the *list of attributes* is used to define variables. We have used these attributes in order to store the initial knowledge of the Communication Entities, and also to store other elements necessary to execute internal computations (control parameters). The TURTLE framework provides the following default attribute types:

- *Natural*
- *Boolean*
- *Array of Naturals*
- *Array of Booleans*

Additionally, the designer is allowed to define data structures that are a combination of Natural and Boolean elements.

Using TURTLE Class Diagrams for modeling cryptographic protocols

Each CE is modeled with one TClass. All related protocol elements like *communication channels, time counters, crypto functionalities* and *services*, are included in the same model class diagram, using tclasses and behavior of tclasses.

In the next paragraphs we briefly present other TURTLE features that are used for modeling various protocol elements.

PDU structure

Protocol Data Unit (PDU) is defined with a specific data structure. Our PDU definition distinguishes between *plain text data*, *ciphered data*, *Message Authentication Codes* (MAC), *Signatures* and *Certificates*. Table 3 presents the data fields of the PDU structure.

Table 3 PDU data structure.

Entry name	Data type	Use
id	Natural	Message id.
origin	Natural	Reference to the sending CE.

destination	Natural	Reference to the destination CE.
key0	Natural	Value of the first symmetric/asymmetric key that is used for data encryption.
firstIndex0	Natural	Reference to the first PDU token which is encrypted with *key0*.
lastIndex0	Natural	Reference to the last PDU token which is encrypted with *key0*.
key1	Natural	Value of the second symmetric/asymmetric key that is used for data encryption.
firstIndex1	Natural	Reference to the first PDU token which is encrypted with *key1*.
lastIndex1	Natural	Reference to the last PDU token which is encrypted with *key1*.
.	.	.
.	.	.
keyK	Natural	Value of the K-th symmetric/asymmetric key that is used for data encryption.
firstIndexK	Natural	Reference to the first PDU token which is encrypted with *keyK*.
lastIndexK	Natural	Reference to the last PDU token which is encrypted with *keyK*.
keyMAC	Natural	Value of the symmetric key that is used to MAC the message.
firstIndexMAC	Natural	Reference to first PDU token that is protected by the MAC.
lastIndexMAC	Natural	Reference to the last PDU token that is protected by the MAC.
keySign	Natural	Value of the asymmetric key that is used to sign the message.
firstIndexSign	Natural	Reference to first PDU token that is protected by the Signature.
lastIndexSign	Natural	Reference to the last PDU token that is protected by the Signature.
certK_ca	Natural	Reference to the Certification Authority key that is used to sign the certificate.
certPK	Natural	Value of the Public Key to which the certificate is issued.
data0	Natural	First token of the PDU.
data1	Natural	Second token of the PDU.
.	.	.

.	.	.
dataN	Natural	N-th token of the PDU.

Model parameters

The parameters of the model are stored in a data structure called *Parameters*. Such a structure simplifies the use and modification of model values. Examples of fields are:

- Flags of the crypto interface: return values as well as status and errors.
- Execution time of each operation
- Values of message types: *requests*, *acknowledgments* and *nonces*
- Various protocol and communication information: maximum number of protocol executions, maximum re-transmission attempts

An example of *Parameters* fields is presented in Table 4.

Channels

Communication channels between Communication Elements (CE) can be represented in the model as one TURTLE class. Depending on proofs that are targeted, the channel representation may be very abstract, thus reflecting a simplified behavior. In such a simplified view, almost all details in message transfers are abstracted: this concerns for example fragmentation of frames. Also, only one communication at a time can occur on the simplified channel model. However, the simplified channel can simulate random message loss and delay. All those simplifications / abstractions lead to a verification focused in the model of the protocols, and not directly in their implementations. Hence, unless it is explicitly mentioned, the channel representation is based upon a simplified behavior.

The communication between CEs is performed as follows:

1. The channel waits for an incoming message on its input interfaces (i.e., on its gates).
2. When a message is received, the channel identifies the message origin and destination from the PDU structure.
3. A random choice makes it possible to decide between forwarding the message to the right destination, or discarding it (message loss).

The Activity Diagram of a typical channel is provided in Figure **??**.

Crypto Library

The Communicating Entities (CEs) have to perform several cryptographic operations in order to forge and analyze messages. The specification of these cipher cryptographic has been provided in [19]. In the TURTLE approach, we abstract such operations using their interfaces thus simplifying the modeling and verification tasks. Beside the fact that all CE's perform analogous cipher operations, having a single *Crypto Library* in the model simplifies protocol representation and consequently verification. Such a crypto library is represented as a TURTLE class that models the following operations:

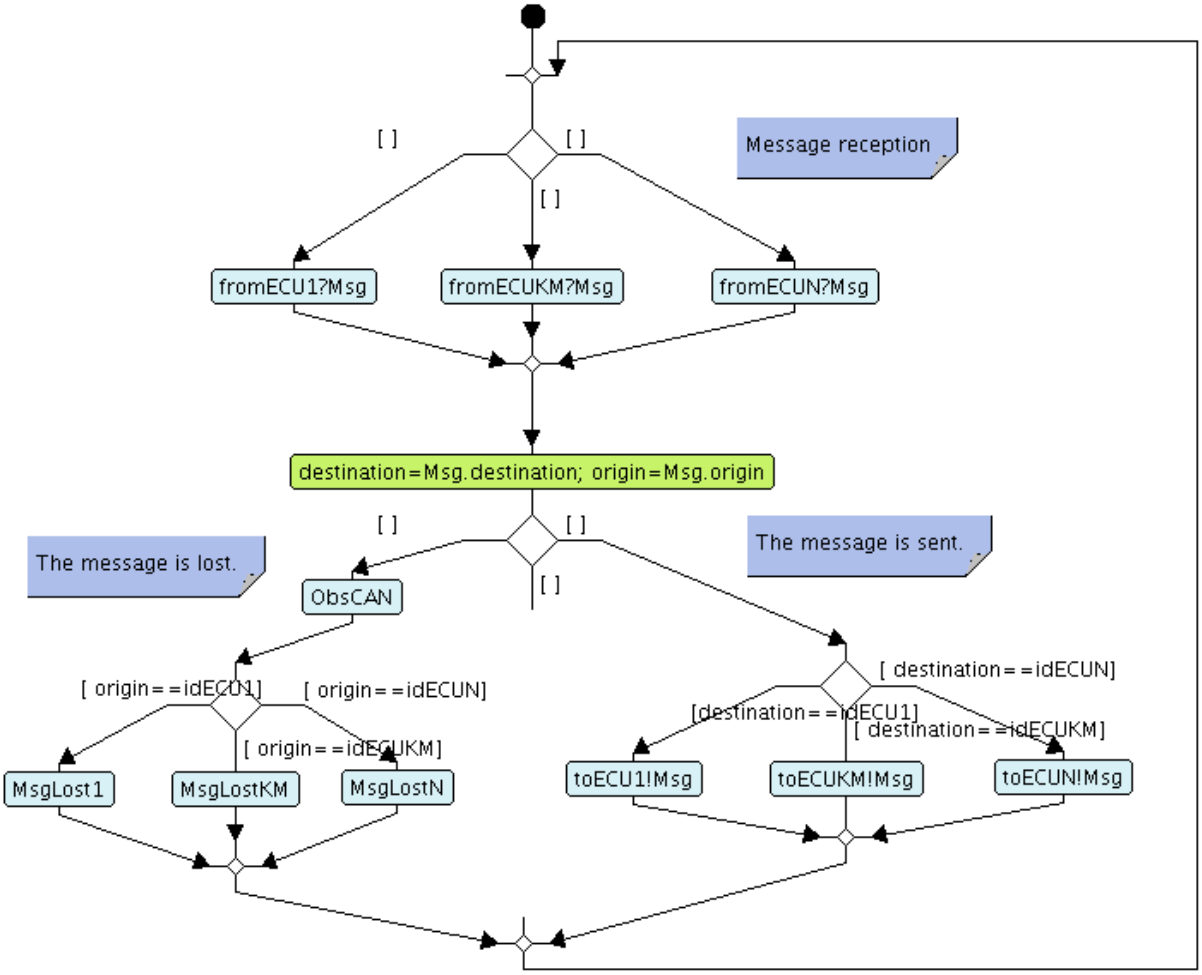

[ ]
[ ]
[ ]
Message reception
fromECU1?Msg
fromECUKM?Msg
fromECUN?Msg
destination=Msg.destination; origin=Msg.origin
[ ]
[ ]
[ ]
The message is lost.
The message is sent.
ObsCAN
[origin==idECU1]
[origin==idECUN]
[origin==idECUKM]
[destination==idECUN]
[destination==idECU1]
[destination==idECUKM]
MsgLost1
MsgLostKM
MsgLostN
toECU1!Msg
toECUKM!Msg
toECUN!Msg

Table 4 Example of *Parameters* fields

Entry name	**Data type**	**Use**
$encryptOK = 1$	Natural	Indicates a successful encryption.
$encryptKO = 0$	Natural	Indicates an unsuccessful encryption.
$macOK = 1$	Natural	Indicates a successful MAC generation or verification.
$macKO = 0$	Natural	Indicates an unsuccessful MAC generation or verification.
$signOK = 1$	Natural	Indicates a successful signature generation or verification.
$signKO = 0$	Natural	Indicates an unsuccessful signature generation or verification.
$Decrypt = 0$	Natural	A decryption operation should be performed.
$Encrypt = 1$	Natural	An encryption operation should be performed.
$createMAC = 2$	Natural	The creation of a MAC code is requested.
$verifyMAC = 3$	Natural	The verification of a MAC code is executed.
$createSign = 4$	Natural	The creation of a signature is demanded.
$verifySign = 5$	Natural	The verification of a signature is requested.
$dlayDecrypt = 5$	Natural	The time cost for decryption.
$dlayEncrypt = 5$	Natural	The time cost for encryption.
$dlayMAC = 5$	Natural	The time for MAC creation and verification.
$dlayCAN$	Natural	The average time for communication through CAN.
$ACK = 1$	Natural	The acknowledgement code.
$N\_x = 11$	Natural	A generated random number that is used as a nonce.
$con\_req = 50$	Natural	The code for a connection request.

- Symmetric and asymmetric deciphering
- Symmetric and asymmetric ciphering
- Retrieve a key value
- Generate and verify MAC's
- Generate and verify signatures
- Generate and verify certificates.

Each CE that requires to use these functions is linked to the crypto library throughout a dedicated gate called $cipherN$. An invocation to the crypto library includes the reference to this invocation gate, a PDU message, the requested operation, a reference to the involved key (key handle) and a reference to the first and last PDU tokens (or fields) on

which the called operation is expected to be applied (e.g., deciphering tokens 1 to 5 of a given PDU). Whenever the crypto library is called, the arguments are analyzed before the corresponding operation is performed. The crypto library directly operates on the PDU structure thus performing symbolic crypto operations. A symbolic operation means that plain PDU values are never modified ($PDU.data\_i$). Instead of that, the corresponding PDU entries are updated (e.g., setting a flag to the *crypted* value) thus reflecting the new structure of the message. For example, the encryption of the first three PDU tokens with the key SK_x is represented with the following assignations:

$$PDU.key0 = SK\_x$$
$$PDU.firstIndex0 = 0$$
$$PDU.lastIndex0 = 2$$

Similar operations are carried out when MAC's and signatures are generated. Certificates can be added to the PDU by setting a reference to the secret key of the Certification Authority ($PDU.certK\_ca$) and the reference to the public key that is bound with the certificate ($PDU.certPK$). This public key should correspond with the CE's identity to who the certificate is issued. The verification of MAC's and signatures requires a key handle as well as the the first and last indexes that are respectively protected with the MAC or signature. Thus, if a CE – or an attacker – doesn't know a key, it is not allowed to verify MAC's nor signatures associated with such key. Analogously, a certificate can not be appropriately opened nor verified if the CE doesn't know the reference of the Certification Authority or its respective public key. Once the cipher operations are performed, the crypto library generates a status code that is returned with the resulting PDU. An overview of the Activity Diagram of the crypto library is presented in Figure 5.

The notation used for cryptographic elements in TURTLE Design is the one defined in the crypto library. This notation is presented in Table 5.

Table 5 Notation used for protocol modeling in TURTLE

Notation	**Description**
SK_x	Asymmetric secret key of the CE x.
PK_x	Asymmetric public key of the CE x.
$PsSK\_x$	Asymmetric pseudo secret key of the CE x.
$PsPK\_x$	Asymmetric pseudo public key of the CE x.
$SesK$	Symmetric session key that is shared between a group of CE's.
PSK_x	Symmetric pre shared secret key of the CE x.
N_x	Random number that is used as a nonce by the CE x.
CA	Reference to a Certification Authority.
ts	time stamp.

UTC discrete time

The general hypothesis of synchronized clocks is realized through a very simple approach. First, unless it is explicitly mentioned, the operations that are executed in the

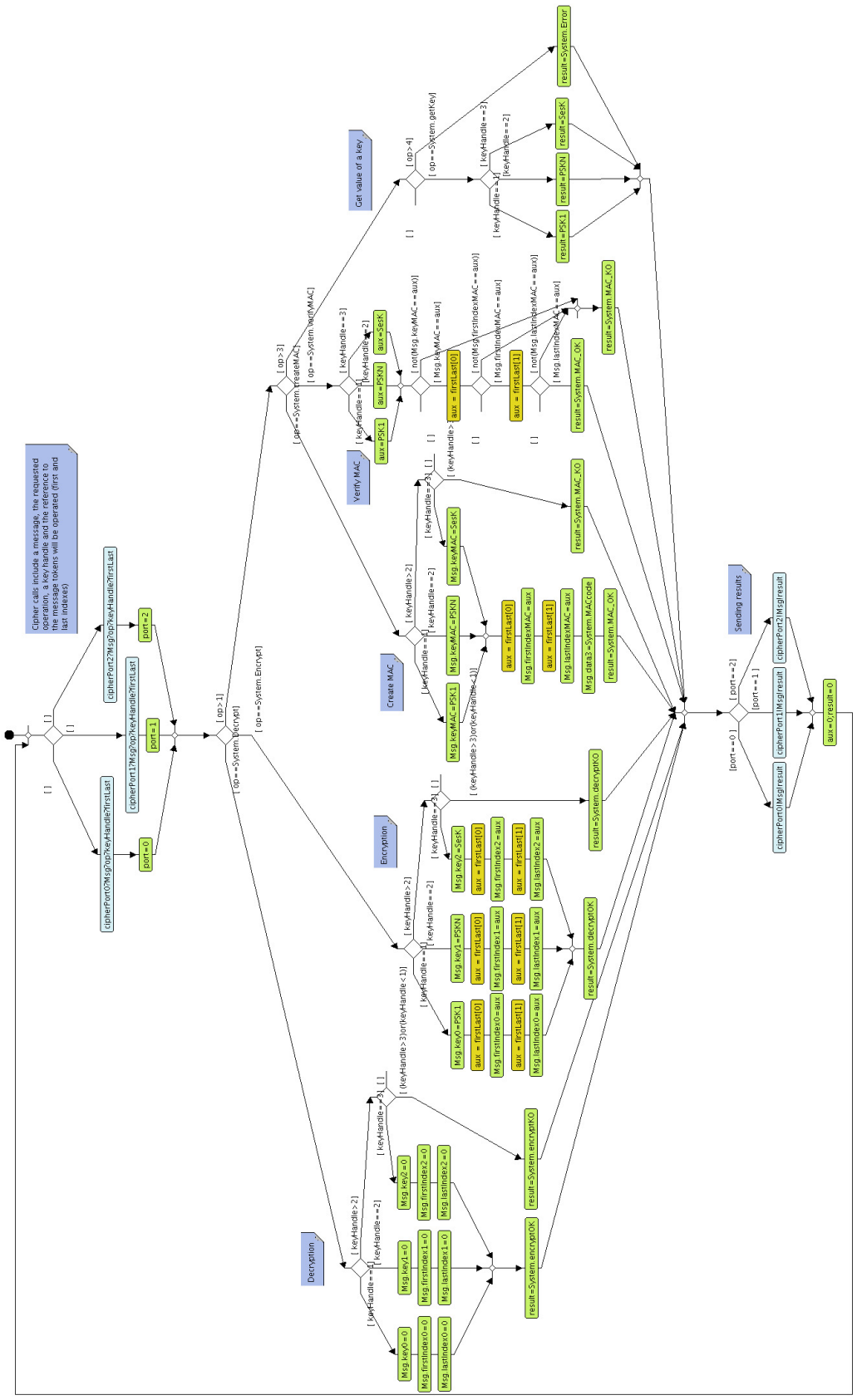


Figure 5 Activity diagram of the crypto library TURTLE class

model have a deterministic duration, even if non deterministic delays can also be modeled. Operation delays are stored in the *Parameters* Tclass of the model (see *Model parameters*). Secondly, the UTC time is represented as a TURTLE class – called UTC – which includes a list of invocation gates. Every CE, or any other class that takes time to perform operation, is linked to UTC through a dedicated gate that is called $utcX$. UTC includes two local attributes ($actualTime$ and $timeUpdate$) whose initial value is 0. Right after a time consuming operation is finished, the respective CE or class calls UTC thus informing the respective operation time delay. Afterwards, UTC catches the $timeUpdate$ and updates $actualTime$:

$$actualTime = actualTime + timeUpdate$$

Finally the new $actualTime$ is returned through the same invocation gate thus providing the current time. The, UTC returns to its initial state thus waiting for an incoming time update. Since the invocation as well as other UTC operations are carried out without the passage of time, the UTC model preserves synchronization. Thus, a CE or class can retrieve the actual time by sending a $timeUpdate$ with value 0.
Obviously, this simplified UTC model assumes a sequential protocol execution. Indeed, it implies that the UTC interface is used by a single CE at a time whilst the others are waiting for. As a consequence of this behavior, no operations nor time requests are performed in the mean time. An overview of the Activity Diagram of a UTC class is shown in Figure 6.

The UTC AD can be modified to overcome the limitations of the sequential protocol execution. This alternative is explained in the next paragraphs.

1. Several time consuming operations OP_x can be initiated by different CE's (denoted by x).

2. Whenever an operation is initiated by x the actual time is retrieved from UTC and stored in the variable $OP\_x.t_0$.

3. Right after an operation is finished, a request for time update is sent to the UTC. The request includes $OP\_y.t_0$ and the delay of the corresponding operation $Parameters.dlayOP$.

4. The $actualTime$ variable is updated according to the next options:

   (a) $actualTime == OP\_y.t_0$: In this case, no other CE has updated the UTC since the operation OP_y was initiated. Therefore the time can be updated by $actualTime = actualTime + dlayOP$.

   (b) $OP\_y.t_0 + dlayOP > actualTime > OP\_y.t_0$: Indicates that other CE(s) has(have) updated the UTC. In such case, the $actualTime$ is just updated by the difference $(OP\_y.t_0 + dlayOP) - actualTime$. It implies that: $actualTime = (OP\_y.t_0 + dlayOP)$.

   (c) $actualTime >= (OP\_y.t_0 + dlayOP)$: Indicates that other CE(s) has(have) updated the UTC. Since the $actualTime$ is greater than or equal to the update request, the variable $actualTime$ remains unchanged.

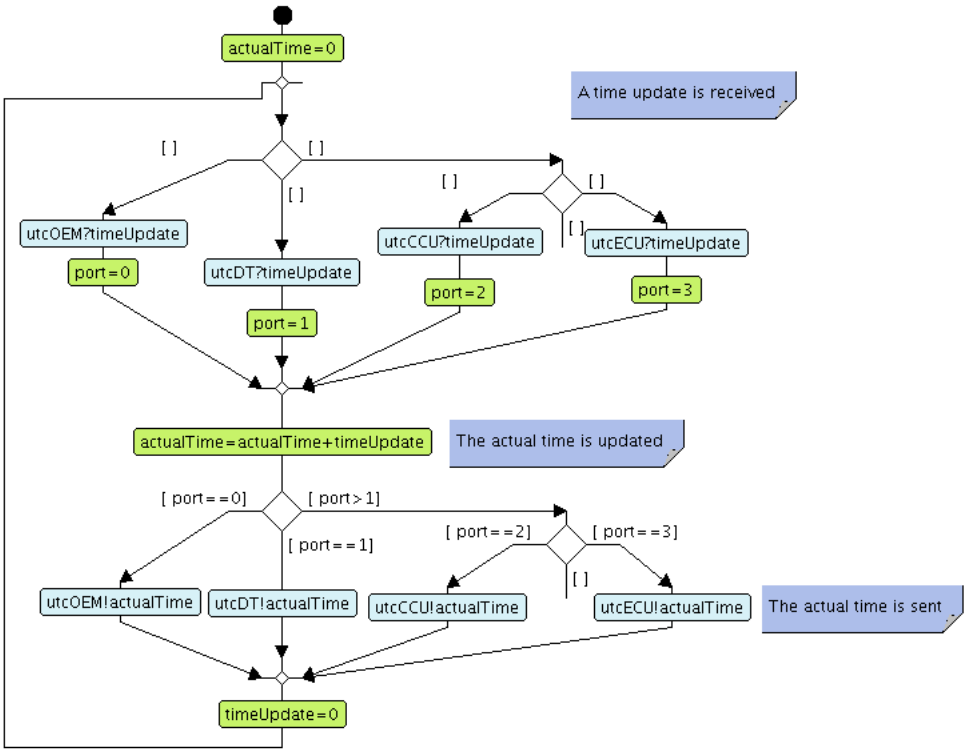


Figure 6 Activity diagram of a UTC class

As a conclusion, *UTC* can handle sequential or no sequential executions, and also deterministic or non deterministic delays.

3.3.2 Properties

This subsection addresses how security properties are represented in TURTLE.

The representation of properties requires that a model of the protocol is completed. In addition, we assume that the security requirements targeted by the protocol have been correctly identified. TCD elements can then be complemented with new instances to adequately represent the security property. Additionally, the verification of each security property relies on an special TURTLE class that is called *Security Observer*. As a conclusion, the modeling of security properties – and the observation of those properties – relies on additional classes that are added to the model of the system.

Each security property is thus associated with a *Security Observer* (*SO*). The *SO* is allowed to interact with all the CE's and TURTLE classes in the TCD. A *SO* can directly retrieve information from CE classes, in order to known in which protocol stage is currently each CE. However, the SO information retrieval is carried out without modifying the original protocol temporal and logical behavior. More precisely, every CE has a dedicated gate from which *SO* can retrieve information (unidirectional exchange, from the investigated CE to *SO*).

How information are retrieved and used by a *SO* is expressed within the Activity Diagram of that *SO*. A given internal gate is used to signal that the security property is satisfied, or violated. Searching for the (non) reachability of that action makes it possible to study security properties of the model.

3.3.3 Attacker

The attacker is a mean to explore possible misuse of the protocol features thus playing a major role in the verification process. Ideally, a generic attacker should be able to challenge the TCD with a wide variety of inputs thus ensuring the coverage of attacks space. To achieve its goal, the attacker should generate such challenges and correctly assess successful attacks.

In our approach each attacker targets a specific security property (e.g., confidentiality, or authenticity). Since security properties are represented as TURTLE classes, the attacker can be defined in terms of the negation of the *Security Observer* logic. More specifically, the negation of a security property determines a generic goal for the respective attacker. Moreover, the attacker can challenge the TCD with inputs that satisfy the negated security property. Such space of inputs should contain the attacks that are addressed in the risk and threat analysis of EVITA D2.3 [17], thus ensuring attack coverage.

First, generic attackers are represented in the TCD model as TURTLE classes. Secondly each attacker is associated with a *Security Observer* class. This *SO* class is taken as an initial reference for the definition of the *Attacker*; on the one hand, the elements

in the *Security Observer* can be used for the *Attacker* definition, on the other hand the *Security Observer* provide insights for the *Attacker* behavior.

The *Attacker* behavior is defined through its Activity Diagram. Unlike its related *Security Observer*, an *Attacker* can only retrieve information from classes representing **channels**, and can definitely not interact directly with CEs or crypto libraries. More precisely, the *Attacker* behavior is based upon the Dolev-Yao Threat model. In this approach the attacker can intercept, alter and inject messages into the communication media and its power is only limited by the restrictions imposed by cryptographic functions. Also, attacks that alter the physical infrastructure are not considered. We also assume that operations performed by the attacker on messages – e.g., substitution of values – can be done in 0-time.

Finally, in order to ensure that the *Attacker* behavior is compliant with the model hypotheses of the ToV, each attack operation have to be first validated by a specific TURTLE class called *Hypotheses*. The *Hypotheses* class determines whether the attack operation is compliant with the model hypotheses, or not. For example, the attacker may try to modify a PDU token by executing $PDU.data0 = Attacker.decrease(PDU.data0, 1)$. unfortunately, $data0$ may possibly take the value of a random nonce or secret key thus violating a general hypothesis (See section *General Assumptions*). Thus, the *Hypotheses* class enforces valid attacks, and so it avoids the presence of false attacks.

As a summary, the *Attacker* model takes into account:

1. The elements in the associated *Security Observer*.
2. The space of options determined by the negated security property.
3. The exchanges in the protocol.
4. The restrictions imposed by model hypotheses (valid operations).
5. The assessment of successful attacks.

In order to explore all possible interactions with CEs, a generic attacker approach is used:

1. *Message interception.* The *Attacker* intercepts a message in a channel.
2. *Message analysis.* The message content is analyzed to determine the possible operations that can be performed.
3. *Attack assessment.* As a part of the previous analysis, the assessment of successful attacks is made, and if it is positive, the result is provided by executing a specific action.
4. *Attack technique.* An attack technique is started in a non-deterministic way.
5. *Attack validation.* The attacker operations are validated by the *Hypotheses* class.

6. *Forge a new message.* After validation, the attack technique is applied and a new message is forged.

7. *Store knowledge.* Received and modified PDU's are used to increase the *Attacker* knowledge.

8. *Send forged message.* The forged message is sent back to the channel.

And so on.

3.3.4 Formal Verification

This section describes how security properties can be represented and verified in TURTLE designs. As explained before, every property is represented using a TURTLE class named *Security Observer*. Additionally, every *Security Observer* is associated to an *Attacker* class that targets attacks that may violate the property under study. We recall that a TURTLE design represents a single session of the modeled protocol. Beside this fact and because of sequential behavior of CE's, the channel is assumed to transmit only one message at a time. Consequently, whenever a *Security Observer* observes a sent message, the subsequent reception truly corresponds with the most recent delivered message. *Security Observer*s and channels could obviously be improved to reflect more complex behaviors.

The System Model, the *Security Observer*, the *Attacker* and the *Hypotheses* are all represented in the same TURTLE Design. Using TTool, this integrated model can be automatically translated to a formal specification in LOTOS [16] or in UPPAAL [7]. To generate those formal specifications, TTool implements a press button approach, and so, no previous knowledge in underlying formal languages is required.

LOTOS [16] is an ISO standardized Formal Description Technique for distributed processing system specification and design. A LOTOS specification, itself a process, is structured into sub-processes. A LOTOS process is a black box which communicates with its environment through gates using a multiple rendezvous offer. Values can be exchanged when synchronization occurs. That exchange can be mono- or bi-directional.
Parallelism and synchronization between processes are expressed by composition operators. Those operators include process sequencing, synchronization on all communication gates, synchronization on some gates, non deterministic choice, preemption, and total interleaving (parallel composition with no synchronization). Composition operators are identified by their symbols (see Table 6).

CADP [1] is a toolbox aims at offering formal verification features for LOTOS specifications. More particularly, CADP can generate a reachability graph from a LOTOS specification. A reachability graph provides all possibles traces of the system in the form of a LTS.

Definition 1. *Labeled Transition System.*
A Labeled Transition System (LTS) is a 4-tuple (Q, Σ, q_0, L)*, where* Q *is a finite set*

Table 6 LOTOS operators

Operators	Description	Example
[]	Choice	$P[a,b,c,d] = P1[a,b][]P2[c,d]$
|||	Parallel	$P[a,b,c,d] = P1[a,b]\|\|\|P2[c,d]$
$\|[b,c,d]\|$	Parallel composition with synchronization on several gates (b,c,d)	$P[a,b,c,d,e] = P1[a,b,c,d]\|[b,c,d]\|P2[b,c,d,e]$
hide b in —[b]—	Parallel composition with synchronization on gate b, where gate b is hidden	$P[a,c] = hidebinP1[a,b]\|[b]\|P2[b,c]$
>>	Sequential composition	$P[a,b,c,d] = P1[a,b] >> P2[c,d]$
[>]	Disrupt	$P[a,b,c,d] = P1[a,b][> P2[c,d]$
;	Process prefixing by action a	$P[a] = a; P1[a]$
stop	Process which cannot communicate with any other process	$P[a] = stop$
exit	Process which can terminate and then transform itself into stop	$P[a] = exit$

of states, Σ is a finite set of actions called alphabet, $q_0 \in Q$ is the initial state and $L \subset Q \times \Sigma \times Q$ is a ternary relation that determines the structure of the LTS.

An LTS is fully described by the set of sequences of actions that can be performed from its initial state q_0.

Definition 2. *Trace of an LTS.*
The trace of an LTS is a sequence of actions $\alpha_0, \ldots, \alpha_{n-1}$, $\alpha_i \in \Sigma$, that is associated to a sequence of states $q_0, \ldots, q_n$, $q_i \in Q$, such that $(q_i, \alpha_i, q_{i+1}) \in L$ for $i = 0, \ldots, n-1$.

As explained before, proving a security property relies on the reachability of a given observer action.

Definition 3. *Reachable state.*
Let $LTS = (Q, \Sigma, q_0, L)$ a Labeled Transition System. A state $q \in Q$ is reachable if there exists at least one trace $\alpha_0, \ldots, \alpha_{n-1}$ which associated sequence of states is $q_0, \ldots, q_n$ such that $q_n = q$.

The set of all possible traces in the LTS determines its language which is denoted by $\mathcal{L}(LTS)$. Whenever a TURTLE Design (TD) is translated to a LOTOS specification, an LTS representing all reachable states of the model may be computed – if the system is of reasonable size. Otherwise, other formal verification techniques can be used, e.g, *on-the-fly model-checking*. However, to simplify this document, we only consider systems for which the LTS can be generated. Such an LTS is denoted by LTS_{TD}.

Traces that lead to a state in LTS_{TD} contains actions of either the system model, the *Security Observer* or the *Attacker*. Also, those actions correspond to either:

1. *Output Data*: Data output on a communication channel, either by a Communicating Entity (CE) or by an *Attacker*.

2. *Input Data*: Data input from a communication channel, and to either a Communication Entity or to an attacker.

3. *Cryptographic function*: Each call to a cryptographic function of the crypto library corresponds to two actions (one for the function request, one for the return from the function).

4. *Non-synchronized actions*: actions on gates which are not synchronized between classes, that is, internal actions of classes, including actions generated by *Security Observer*. Those last actions are the ones for which reachability shall be studied.

The syntax of TURTLE actions is established in the following definition.

Definition 4. *TAction.*
An action $\alpha \in \Sigma$ is a TURTLE action (Taction) iff it is compliant with the following grammar

$$\begin{aligned} \textit{Taction} ::= &\ \textit{TClass.Tgate}\ [(!\textit{Tattribute} \mid ?\textit{Tattribute})^*] \\ \textit{Tclass} ::= &\ \textit{c\_name} \\ \textit{Tgate} ::= &\ \textit{g\_name} \\ \textit{Tattribute} ::= &\ \textit{at\_name} \mid \textit{Tdata.at\_name} \\ \textit{Tdata} ::= &\ \textit{d\_name} \end{aligned}$$

where "!" means a data output and "?" a data input. The semantics of names is as follows:

c_name:	*The name of a TURTLE class.*
g_name:	*The name of a gate defined in a class c_name.*
d_name:	*The name of an attribute whose type is a TURTLE data structure.*
at_name:	*The name of an attribute defined in a class c_name or a name of the field of the data structure of the attribute d_name.*

An instance showing TURTLE actions and names is presented in figure 7.

Note that if $\Gamma = (Q, \Sigma, q_0, L)$ is an LTS representing all reachable traces of a TURTLE Design, then Σ is entirely composed of TURTLE actions of this TURTLE Design. TURTLE Designs can execute specific TURTLE actions thus showing correct (or incorrect) termination of model execution.

Definition 5. *Traces for the formal verification of security properties.*
Let's note TD-Sec a TURTLE Design built for security proof purpose, i.e., built as explained in previous sections. A trace proving that no security properties are violated in TD-Sec always terminates with the TURTLE action $\alpha =$ Tclass.EndOK (See definition 4)

Finally, the following notation is used to represent possible message alterations during a transfer in an insecure channel.

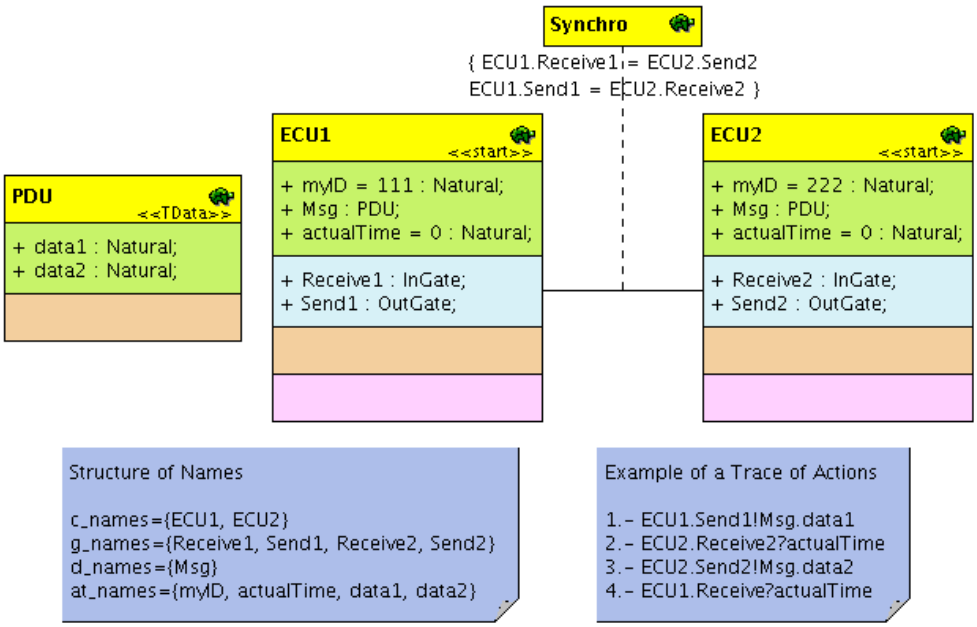


Figure 7 TURTLE actions and names examples

Notation 1. *Sending and receiving actions.*
TURTLE actions denoting the output of PDU take the form $\alpha =$ *Tclass.send!PDU whilst corresponding input actions on PDU take the form* $\alpha =$ *Tclass.receive?PDU'.*

In the next paragraphs, we show how security properties are represented and verified in the TURTLE approach.

Verification of Integrity

The *Security Observer* for Integrity knows every transmitted message. Indeed, whenever a message is sent, a random choice in the channel allows either message loss or CE reception. In the first case, the *Security Observer* activates the gate $IntegrityMsgLost$. The verification of integrity is performed right after a CE inputs a message. First, the *Security Observer* verifies that the sent and received messages are the same. Secondly, if a difference is found, hence the property is not satisfied, so the gate $IntegrityKO$ is executed. Otherwise, if no difference was observed, the property is satisfied and thus $IntegrityOK$ is executed.

The integrity *Attacker* intercepts every message in the communication channel, and accordingly update its local knowledge. After interception of a PDU, the *PDU* contents is analyzed; plain text tokens, ciphered tokens, MAC's, Signatures and Certificates are thus classified (token types). Actions performed by the *Attacker* are among the basic following ones:

RndModification(i). The data token i is increased or decreased by a random value x.

Interchange(i,j). The data tokens i, j, $i \neq j$, are interchanged. Tokens i, j should belong to the same type.

Substitution(i). The data token i is substituted for a randomly selected one from W_{at}. Substitution is performed over tokens of the same type.

Dropping(). The attacker drops the entire PDU.

Nothing(). The attacker does nothing with the PDU.

The *Attacker* randomly chooses one basic action – and its arguments if applicable. Before applying such an action, a copy of PDU is stored in W_{at}. The stored PDU can be later used for substitution operations. Afterwards, the selected action, along with the PDU, are delivered to the $Hypotheses$ class. Among other validations, the $Hypotheses$ class verifies that resulting tokens do not take neither GRN's nor secret key values. If the action does not violate any hypotheses, it is validated. Otherwise the *Attacker* is forced to take a different action. Eventually, the *Attacker* applies an action to the PDU thus forging a new message PDU'. Then, a random choice models whether the PDU' is stored in W_{at}, or is discarded. Finally the forged PDU' is delivered into the communication channel.

Formal verification can be performed once *Security Observer*, *Attacker* and $Hypotheses$ have been added to the model as classes interacting with the protocol model. As a result of such verification, TTool / CADP / UPPAAL formally determines the set of possible actions and reachable states thus forming the associated LTS_{TD}. Finally, the LTS_{TD} is analyzed in order to determine whether the property is satisfied, or not, and also to determine possible attack(s). Trace evaluation is made according to the following rules:

1. Whenever the *Attacker* alters a PDU ($PDUModified$) the $Integrity\ Observer$ should eventually perform $IntegrityKO$ for the same PDU.

2. If the *Attacker* did not modify the PDU ($PDUNotModified$), then the $Integrity\ Observer$ should eventually perform $IntegrityOK$ for the same PDU.

3. Whenever a message is lost, the $Integrity\ Observer$ should activate $IntegrityMsgLost$ for the lost PDU.

The verification of integrity by the *Security Observer* is formalized in the following definition.

Definition 6. *Integrity verification.*
Let $Tr := \alpha_0, \ldots, \alpha_n$ *a trace in* LTS_{TD}*, with* α_i *a TURTLE action. We say that* Tr ***verifies integrity*** *for a PDU iff there exist actions* α_k, α_l, α_m *in* Tr*, with* $0 \leq k < l < m \leq n$, $k, l, m \in \mathbb{N}$ *such that*

1. $\alpha_k = SecurityObserver.receive?PDU$

2. $\alpha_l = SecurityObserver.receive?PDU'$

3. $\alpha_m = SecurityObserver.IntegrityOK!PDU'$ *or*
   $\alpha_m = SecurityObserver.IntegrityKO!PDU'$.

Table 7 Trace evaluation for verification of Integrity in TURTLE Designs

Actions of *Security Observer* / **Actions of** *Attacker*	$IntegrityOK$	$IntegrityKO$
$PDUNotModified$	Trace OK	Model Flaw
$PDUModified$	Model Flaw	Trace OK

4. $\forall j$ *such that* $k < j < l$ *or* $l < j < m$, $\alpha_j =$ *Tclass.Tgate... and* *Tclass* $\neq$ *SecurityObserver*
.

Finally, the evaluation of integrity for LTS_{TD} traces is summarized in Table 7.
Thus, we adopt the following definition for an integrity attack trace.

Definition 7. *Integrity attack.*
Let $Tr := \alpha_0, \ldots, \alpha_n$ *a trace in* LTS_{TD}.
We say that Tr *is an* ***integrity attack*** *for a PDU iff there exist actions in* Tr *namely* $\alpha_i,\ \alpha_j,\ \alpha_k,\ \alpha_l,\ \alpha_m$, *with* $0 \leq i < j < k < l < m \leq n,\ i, j, k, l, m \in \mathbb{N}$, *such that:*

1. $\alpha_i = SecurityObserver.receive?PDU$
2. $\alpha_j = Attacker.PDUModified!PDU'$
3. $\alpha_k = SecurityObserver.receive?PDU'$
4. $\alpha_l = SecurityObserver.IntegrityKO!PDU'$
5. $\alpha_m = CE_X.EndOK$
6. Tr *and* α_i, α_k *and* α_l *satisfy Definition 6.*
7. $\forall r$ *such that* $l < r < m$, $\alpha_r \notin \{\alpha_i,\ \alpha_j,\ \alpha_k,\ \alpha_l,\ \alpha_m\}$.

Remark 1. *Definitions 6 and 7 rely on the simplified communications channel. Of course, those definitions have to be modified for more complex scenarios.*

Verification of Authenticity

Similarly to the integrity observer, the authenticity *Security Observer* knows every delivered message. Moreover, the identity of the sending CE is captured. Thus, the *Security Observer* is able to correctly asses message *author* ($receive?PDU?author$). According to the channel policy, the message is either lost or sent to a CE class. In the first case the $MsgLostObs$ gate is activated. Otherwise the message is received by a CE class and thus the *Security Observer* activates $receive?PDU'$. Right after, the *claimed_author* is determined according to the following rules:

1. If PDU' includes a MAC, then *claimed_author* is the owner of the MAC key.
2. If PDU' includes a signature, then *claimed_author* is the owner of the signature's key.

3. If PDU' includes a certificate, then *claimed_author* is the owner of the private key associated to the public key that is bounded with the certificate.

4. If PDU' does not include neither MACs, signatures, nor certificates, then *claimed_author* is set to $PDU'.origin$.

Remark 2. *Our approach assumes that each key and each certificate are bounded to a unique CE. The authenticity Security Observer may verify such correspondences using the CryptoLibrary.*

Once *claimed_author* has been determined, the *Security Observer* verifies authenticity by comparing *author* and *claimed_author*; if they are equal, then authenticity is satisfied and the gate *AuthenticityOK* is activated. Otherwise, *AuthenticityKO* is executed.

Remark 3. *In order to correctly verify real author and claimed_author, the Security Observer should firstly verify that PDU' truly corresponds with PDU. Therefore, to ensure such correspondence and to simplify modeling and verification, we assume* ***that authenticity is verified right after verification of integrity****.*

The authenticity *Attacker* is defined in terms of possible violations of message authenticity. Indeed, some of his basic operations are an immediate extension of integrity *Attacker* operations. However, the authenticity *Attacker* behaves on a higher level of PDU semantics mainly targeting authentication tokens. The *Attacker* relies on the following operations:

AlterMsgOrigin(). $PDU.origin$ is substituted by a different origin randomly taken from W_{at}.

AlterMsgDestination(). $PDU.destination$ is substituted by a different destination randomly taken from W_{at}.

SubstituteMAC(). If PDU contains a MAC, it is replaced by a different one randomly taken from W_{at}.

SubstituteSign(). If PDU contains a signature, it is replaced by a different one randomly taken from W_{at}.

SubstituteCert(). If PDU contains a certificate, it is replaced by a different one randomly taken from W_{at}.

Nothing(). The *Attacker* does nothing with the PDU.

Whenever a message is intercepted, its content is classified (signature, etc.) and stored in W_{at}. Afterwards, a basic action is randomly selected. The action and the PDU are sent to the *Hypotheses* class for evaluation purpose. Among other possible validations, the *Hypotheses* class determines if modified PDU tokens do not take neither nonces nor secret key values. In such a case, the *Attacker* is forced to select a different action.

Right after validation, the *Attacker* applies the selected action to the PDU thus forging a new message PDU'. A random choice allows either storing modified PDU' values in W_{at} or skipping storing. Last but not least, PDU' is delivered into the communication channel.

The so built LTS_{TD} is formally verified using the following rules for trace evaluation:

1. Whenever the *Attacker* alters the authenticity of a PDU ($PDUNotAuthentic$) the *Authenticity Observer* must eventually execute action $AutenticityKO$ for the same PDU.
2. If the *Attacker* did not modify the PDU ($PDUAuthentic$), then the *Authenticity Observer* must eventually execute $AuthenticityOK$ for the same PDU.
3. For each lost message, the *Authenticity Observer* must execute a $MsgLostObs$ action.

The next definition formalizes the operations that are performed by *Security Observer*s in order to verify message authenticity.

Definition 8. *Authenticity verification.*
Let $Tr := \alpha_0, \ldots, \alpha_n$ *a trace in* LTS_{TD}.
We say that Tr ***verifies authenticity*** *for a PDU iff there exist actions* α_k, α_l, α_m *in* Tr, $0 \leq k < l < m \leq n$, $k, l, m \in \mathbb{N}$ *such that*

1. $\alpha_k = SecurityObserver.receive?PDU?author$
2. $\alpha_l = SecurityObserver.receive?PDU'$
3. *Either* $\alpha_m = SecurityObserver.AuthenticityOK!PDU'!claimed\_author$
   with $claimed\_author = author$, *or*
   $\alpha_m = SecurityObserver.AuthenticityKO!PDU!claimed\_author'$
   with $claimed\_author' \neq author$.
4. $\forall j$ *such that* $k < j < l$ *or* $< l < j < m$, $\alpha_j =$ *Tclass.Tgate...* *and* *Tclass* $\neq$ *SecurityObserver*.

The validation of authenticity in the overall LTS_{TD} model is intuitively explained in Table 8.

Table 8 Trace evaluation for verification of Authenticity in TURTLE Designs

Actions *Security Observer* / **Actions** *Attacker*	*AutthenticityOK*	*AuthenticityKO*
PDUAuthentic	Trace OK	Model Flaw
PDUNotAuthentic	Model Flaw	Trace OK

The previous evaluation is formalized in the next definition thus complementing formal evaluation of authenticity in LTS_{TD}.

Definition 9. *Authenticity attack.*
Let $Tr := \alpha_0, \ldots, \alpha_n$ *a trace in* LTS_{TD}.
We say that Tr *is an* ***authenticity attack*** *for a* PDU *iff there exist actions in* Tr *namely* $\alpha_i,\ \alpha_j,\ \alpha_k,\ \alpha_l,\ \alpha_m$, $0 \leq i < j < k < l < m \leq n,\ i, j, k, l, m \in \mathbb{N}$, *such that:*

1. $\alpha_i = SecurityObserver.receive?PDU?author$
2. $\alpha_j = Attacker.PDUNotAuthentic!PDU'$
3. $\alpha_k = SecurityObserver.receive?PDU'$
4. $\alpha_l = SecurityObserver.AuthenticityKO!PDU'!claimed\_author$
   with $claimed\_author \neq author$.
5. $\alpha_m = CE_X.EndOK$
6. α_i, α_k *and* α_l *satisfy definition 8.*
7. $\forall r$ *such that* $l < r < m$, $\alpha_r \notin \{\alpha_i,\ \alpha_j,\ \alpha_k,\ \alpha_l,\ \alpha_m\}$.

Remark 4. *Definitions 8 and 9 rely on the simplified communications channel. Of course, those definitions could be adapted to more complex channels.*

Verification of Freshness

As in previous properties, verification of freshness relies on a *Security Observer* that is allowed to know every transmitted message. Whenever a PDU is sent into the communications channel, the *Security Observer* knows PDU and the time at which the PDU was sent ($receive?PDU?sendTime$). According to the channel policy the PDU may be lost ($MsgLostObs$). Otherwise, the *Security Observer* is sure that the next input message truly corresponds with the most recently sent. Thus, the *Security Observer* receives PDU' as well as the reception time $recTime$ ($receive?PDU'?recTime$). The evaluation of freshness is based upon two criteria. The first one ensures that PDU' is not a copy of a previously exchanged message. Hence, the *Security Observer* records every received PDU and compare its contents in order to avoid message replay. The second criterion ensures that the difference between $recTime$ and $sendTime$ is less than a given $threshold$. If PDU' satisfies those freshness criteria, then the action $FreshnessOK!PDU'$ is executed, otherwise the action $Freshness!PDU!PDU'$ is executed.

Remark 5. *The freshness Security Observer assumes that* PDU' *truly corresponds with* PDU. *Since freshness observer is focused on temporal properties, we assume that only tokens of messages concerned with freshness are modified. Consequently, other possible attacks cannot be traced by a freshness Security Observer.*

The freshness *Attacker* is defined in terms of basic operations that can modify PDU time context. These operations are:

IncreaseTS(). If the PDU includes a time stamp, it is randomly increased by x.

DecreaseTS(). If the PDU includes a time stamp, it is randomly decreased by x.

Delay(). The PDU is delayed an amount of time d. To perform such an operation the attacker requests a time update ($utc!d$) to the UTC class.

Replay(). The PDU is discarded and replaced by a PDU' from W_{at}. Also, this function accordingly modifies PDU' destination, taking the one provided in PDU.

Nothing(). The time context of the PDU remains unchanged.

Whenever a PDU is intercepted by the *Attacker*, it is stored in W_{at}. PDU tokens are classified, and stored. Then, a random choice determines the basic operations to apply. Afterwards, the PDU and the selected basic operation are informed to the $Hypotheses$ class. Among other operations, the $Hypotheses$ class verifies that modified tokens do not take nonces nor secret key values. In case of respected hypotheses, the *Attacker* applies the basic operation. Otherwise, the *Attacker* is forced to select another operation. In the first case, a PDU' with a modified time context is obtained, and *Attacker* executes $PDUNotFresh$. Otherwise, the *Attacker* executes $PDUFresh$. Then, a random choice decides whether the modified PDU' tokens are stored in W_{at}, or not. Last, the message is sent to its destination.

The traces of LTS_{TD} are evaluated to determine freshness satisfaction, and possible attacks. This evaluation is based upon the following rules:

1. Whenever the *Attacker* creates PDU' as being equal to PDU with an altered time context ($PDUNotFresh$), the $Freshness\ Observer$ must eventually execute $FreshnessKO$ for the associated PDU'.
2. Whenever the *Attacker* creates PDU' as being equal to PDU, then the $Freshness\ Observer$ must eventually execute $FreshnessOK$ for the associated PDU'.
3. Whenever a message is lost, the $Freshness\ Observer$ should activate $MsgLostObs$.

Verification of freshness in traces is performed according to the following definition.

Definition 10. *Freshness Verification.*
Let $Tr := \alpha_0, \ldots, \alpha_n$ *a trace in* LTS_{TD}.
We say that Tr ***verifies freshness*** *for a* PDU *iff there exist actions* $\alpha_k,\ \alpha_l,\ \alpha_m$ *in* Tr, $0 \leq k < l < m \leq n,\ k, l, m \in \mathbb{N}$ *such that:*

1. $\alpha_k = SecurityObserver.receive?PDU?sendTime$
2. $\alpha_l = SecurityObserver.receive?PDU'?recTime$
3. *Either* $\alpha_m = SecurityObserver.FreshnessOK!PDU'$ *or* $\alpha_m = SecurityObserver.FreshnessKO!PDU!PDU'$ *is activated.*
4. $\forall j$ *such that* $k < j < l$ *or* $l < j < m$, $\alpha_j =$ $\mathsf{Tclass.Tgate}$... *and* $\mathsf{Tclass} \neq SecurityObserver$.

Table 9 Trace evaluation for verification of Freshness in TURTLE Designs

Actions *Security Observer* / **Actions** *Attacker*	$FreshnessOK$	$FreshnessKO$
$PDUFresh$	Trace OK	Model Flaw
$PDUNotFresh$	Model Flaw	Trace OK

The verification of LTS_{TD} traces with respect to *Attacker* operations is summarized in Table 9.

We finalize this section by providing a definition that characterizes a trace with freshness attack traces.

Definition 11. *Freshness attack.*
Let $Tr := \alpha_0, \ldots, \alpha_n$ *a trace in* LTS_{TD}.
We say that Tr *is a* ***freshness attack*** *for a* PDU *iff there exist actions in* Tr *namely* $\alpha_i,\ \alpha_j,\ \alpha_k,\ \alpha_l\ \alpha_m$, $i < j < k < l < m,\ i, j, k, l, m \in \mathbb{N}$, *such that:*

1. $\alpha_i = SecurityObserver.receive?PDU?sendTime$
2. $\alpha_j = Attacker.PDUNotFresh!PDU'$
3. $\alpha_k = SecurityObserver.receive?PDU'?recTime$
4. $\alpha_l = SecurityObserver.FreshnessKO!PDU!PDU'$
5. $\alpha_m = CE_X.endOK$
6. Tr *and* α_i, α_k *and* α_l *satisfy definition 10*
7. $\forall r$ *such that* $l < r < m$, $\alpha_r \notin \{\alpha_i,\ \alpha_j,\ \alpha_k,\ \alpha_l,\ \alpha_m\}$.

Remark 6. *Definitions 10 and 11 rely on the simplified communications channel. More complex channels require to modify accordingly these definitions.*

3.3.5 Limitations and Conclusions

TURTLE Designs are composed of Communicating entities, crypto libraries, *Attacker*, *Security Observer* and channels. The combination of them makes it quite simple to model protocols, and their related security properties. The latter can be evaluated using model-checking techniques. TTool efficiently supports the modeling phase, as well as the formal verification at a push of a button, relying on CADP.

3.4 ProVerif Verification Approach Overview

3.4.1 System Modeling

ProVerif is a toolkit that relies on Horn clauses resolution for the automated analysis of security properties over cryptographic protocols. ProVerif takes as input a set of Horn

Clauses, or a specification in pi-calculus and a set of queries, and outputs, for each query, whether it is satisfied (*true*) or not satisfied (*false*). Additionally, ProVerif tries to identify a trace explaining how it came to the conclusion that a query is not satisfied. In this subsection we explain how EVITA protocols can be modeled and how security properties can be proved on those models, with ProVerif.

Pi-calculus is a formal language that belongs to the family of formal languages based on process algebras. In ProVerif, a specification takes the form of a system represented as a *pi-calculus* [4] process and properties represented as *queries*. The definition of processes is presented in Table 10.

Table 10 Definition of the process calculus

Notation	**Semantics**
$M, N ::=$	Terms
x, y, z	Variables
a, b, c, k	Names
$f(M_1, \ldots M_n)$	Constructor application
$P, Q ::=$	Processes
$\bar{M}\langle N\rangle.P$	Output N in M then P
$M(x).P$	Input M in x then P
0	Null process
$P\|Q$	Parallel composition
$!P$	Replication of P
$(\nu a)P$	a is restricted to P
$let\ x = g(M_1 \ldots M_2)\ in\ P\ else\ Q$	Destructor application
$if\ M = N\ then\ P\ else\ Q$	Conditional
$evt(M).P$	Event $evt(M)$ then P

The syntax of processes is given in Table 11 when this syntax differs from the definition. A complete specification of the ProVerif process grammar can be found in [10].

Table 11 Equivalences between pi and ProVerif process notation

Notation	
pi process	**ProVerif process**
$\bar{c}\langle M\rangle.P$	$out(c, M); P$
$c(M).P$	$in(c, M); P$
$(\nu a)P$	$private\ free\ a.$ (inside of P)
$begin(M).P$	$event\ begin(M); P$
$end(M).P$	$event\ end(M); P$
$begin\_ex(M).P$	not denoted
$end\_ex(M).P$	not denoted
not denoted	$phase\ n; P$

For modeling security protocols, the ProVerif developers suggest to rely on processes for modeling Communicating Entities (CE). The toolkit already includes a Dolev-Yao-based model of attacker. The modeling of CEs, the attacker process as well as verification details are further described in next sections.

To ease cryptographic protocol description, we decided to split a ProVerif model into several sections named *Basic Blocks, Hypotheses, Queries, Variables, Processes* and *Main Process*. In the next paragraphs, we describe the purpose of those different sections. Other model components are described in subsequent sections.

Basic Blocks

The basic blocks represent the crypto primitives and other functions used for modeling purpose. Additionally, Basic Blocks may include the so called *predicates*, *clauses* and *equations*. Basic Blocks are meant to define the most basic structures and functions used by processes to *modify* variables, and *create* or *reduce* data structures. The Basic Blocks sections is usually provided at the beginning of the pi-calculus specification. Terms involved in blocks, i.e., in function and data structure definitions, do not require a previous declaration. Since each definition of block is independent from another definition, the same terms can be used in several definitions. Blocks that are used to generate new structures are called *constructors*. Analogously, Blocks that are used to reduce structures are called *destructors*. Several reserved words are used to declare constructors and destructors; `fun, reduc, equation,` etc.. Moreover, unless explicitly specified with the keyword `private`, a block can be used in every process in the model. A block labeled with the keyword `private` cannot be used by attackers ($(\nu a)Q$). Finally, Table 12 defines a set of blocks useful for our cryptographic proofs.

Table 12 Basic Blocks in ProVerif Modeling

Notation	Type	Semantics
$encrypt(x, k)$	Constructor	Term x encrypted with key k
$decrypt(encrypt(x, k), k) = x.$	Destructor	Cipher text decrypted with key k
$MAC(m, k)$	Constructor	MAC code of term m with key k
$verifyMAC(m, MAC(m, k), k) = validMAC$	Equation	Verification of the MAC of m with key k
$Pk(k)$	Constructor	The public key associated to the private key k
$host(k)$	Constructor	The host associated to the private key k
$encryptSK(m, k)$	Constructor	Encryption of the term m with private key k
$decryptPK(encryptSK(m, k), Pk(k)) = m$	Destructor	Decryption with the public key $Pk(k)$
$encryptPK(m, Pk(k))$	Constructor	Encrypt term m with the public key $Pk(k)$
$decryptSK(encryptPK(m, Pk(k)), k) = m$	Destructor	Decryption with the private key k
$Hash(m)$	Constructor	The hash of term m

Continued on next page

Table 12 Basic Blocks in ProVerif Modeling

Notation	Type	Semantics
$encryptSK(Hash(m), k)$	Constructor	The signature of term m with key k
$verifySign(m, encryptSK(Hash(m), k), Pk(k)) = validSign$	Equation	Verification of a signature with the public key $Pk(k)$
$Cert(id, sign)$	Constructor	A certificate for the identity id with signature $sign$
$getID(Cert(id, sign)) = id$	Destructor	Get the identity part id of the certificate
$getSign(Cert(id, sign)) = sign$	Destructor	Get the signature $sign$ of the certificate
$verifySign(id, sign, Pk(k\_ca)) = validSign$	Equation	Verify signature of a certificate
$minus(N, one)$	Constructor	Decrease one to term N
$shares(k, kmID)$	Constructor	The key k is shared to the Key Master $kmID$
$Psk(shares(k, kmID)) = k$	Destructor	Retrieves the shared key k
$theKMof(shares(k, kmID)) = kmID$	Destructor	Retrieves the Key Master ID $kmID$
$group((x1, x2, x3))$	Constructor	A group of terms denoted by $x1$, $x2$ and $x3$
$get\_i(group(x1, x2, x3)) = x_i$	Destructor	Retrieves the term i of a group

Continued on next page

Hypotheses

This section is intended to declare the secrecy assumptions that should be verified in the model. A term is not accessible by an attacker iff it is declared as `private` (See Basic Blocks) or if it is defined inside of a process. And so, non-private globally declared terms are assumed to be known by every process in the model. Thus, the Hypotheses section is a mean to formally express secret material that shall never be known by attackers. For example, private keys should be restricted for the attacker from the very beginning. Consequently, they should be either declared with the label `private` or inside of a process. Table 13 exemplifies a few secrecy assumptions.

Variables

Free variables, names and other terms can be declared as follows.

`[private] free` *name*.

As it was mentioned, free variables labeled with `private` are initially restricted for the attacker, but they are known by all other processes of the model. Variables can be also defined inside processes by using the reserved word `new`. The syntax is as follows:

`new` *name*;

Table 13 Example of secrecy assumptions in ProVerif

Hypothesis	**Semantics**
not attacker : *sk_ca*.	The secret key *sk_ca* is initially restricted for the attacker.
not attacker : *pk_ecu*.	The public key *pk_ecu* is initially restricted for the attacker.
not attacker : $firmware$.	The term $firmware$ is initially restricted for the attacker.
not SesK.	The term *SesK* is initially restricted for the attacker.

The scope of so defined variables is local to the process, and to subsequent processes. Local variables are useful to represent secret material to be interchanged between processes, or to be used internally by a process. Hence, secret keys and random numbers are usually defined using `new`. At last, data structures can be used to define terms:

`data` *name/n.*

This provides a mean for the definition of terms that are composed by a certain number n of elements. `data` is useful to define tags: Tags are a mean to differentiate between instances of the same function. Such differentiation enforces termination of the ProVerif resolution algorithm by avoiding unification between such instances which prevents loops in resolution process. Some instances of variables definitions are presented in Table 14.

Processes

In our approach, we associate to each Communicating Entity (CE) in the protocol one single process (and corresponding sub-processes). The definition of those processes relies on previously defined Basic Blocks and Variables. Since each process is associated with a single CE in the protocol, the process model is aligned to the CE specification. Consequently, the initial knowledge of each CE should be taken into account in the process definition. The process definition begins with the `let CE_name =` declaration. Also, each process has a specific identifier. To send and receive information, processes are allowed to use channels declared as free variables. Finally, CEs follow the underlying generic approach:

1. *Message forging*
2. *Message sending*
3. *Message reception*
4. *Message validation* (e.g., checking MAC, certificates)
5. And so on...

In the first stage, (*forging a message*), the information to be sent is arranged in a tuple. The first element of this tuple is always the identity of the process. If the message should be MACed, signed or should include a certificate, then the respective Basic Blocks

Table 14 Examples of ProVerif variables declaration

Term declaration	**Semantics**
free c.	A non restricted channel *c*
*private free c*1.	A restricted channel *c*1
private free pk_ecu.	The public key *pk_ecu* is declared as restricted for the attacker
private free firmware.	The term *firmware* is declared as restricted for the attacker
private free SesK.	The term *SesK* is declared as restricted for the attacker
data validMAC/0.	A term to inform successful MAC verification
data validSign/0.	A term to inform successful signature verification
data validCert/0.	A term to indicate successful certificate verification
data Ack/0.	A term to represent acknowledgment
data one/0.	A term to represent the number 1
new sk_ccu;	The key *sk_ccu* is declared inside of a process
*new N*1*_dt*;	The nonce *N*1*_ccu* is declared inside of a process
data ccu_id/0.	The identification associated to *ccu*
data ecu_id/0.	The identification associated to *ecu*
*data tg*1/0.	The tag 1
*data tg*2/0.	The tag 2

are used to generate additional elements of the message. The example below shows the message `Con_req, N1_ecu` that is forged by the CE `ecu_id`. Such message is additionally protected with a MAC generated using the key `SesK`.

```
new N1_ecu;
let M1= (ecu_id, Con_req, N1_ecu, MAC((Con_req,N1_ecu),SesK)) in
```

The just forged message can be output in a channel named *c* as follows (*message sending*):

```
out(c, M1);
```

Or, in a more concise way:

```
out(c, (ecu_id, Con_req, N1_ecu, MAC((Con_req,N1_ecu),SesK)));
```

Channels in ProVerif have a broadcast semantics, and so all processes can listen to all messages. The Attacker can also listen to those messages, apart when those messages are sent on private channels. Additionally, since channels are broadcast channels, no message destination is required. However, whenever a CE process receives a message from a channel, it can review whether the pattern of the received message corresponds to the expected pattern. In case of pattern match, the process accepts the message i.e. it goes on to the next pi-calculus operator. For instance, the next code shows how the message `M1` in the previous example is received by a process.

```
in(c, m1);
let (hostX, =Con_req, N1, MAC1)=m1 in
```

In this example, `m1` should match the pattern `(hostX, =Con_req, N1, MAC1)` in order to be accepted, otherwise the process can not continue its execution (blocking point). The *validation* phase of a received message depends upon the mechanisms that the CE process is allowed to use. In any case such primitives should be previously defined as a part of the Basic Blocks section. Apart from the attacker, ProVerif is not able to identify whether a process is allowed to use a primitive, or not. Consequently, the fact that primitives are used in compliance with the protocol specification is up to the designer. In the previous example, the process that inputs `m1` must know the key `SesK` in order to verify the MAC. Assuming that the key `SesK` is known by the receiving process, then the MAC verification of `m1` can be performed as follows:

```
let Res1 = verifyMAC((Con_req, N1), MAC1, SesK) in
if Res1 = validMAC then (
 ... process continuation )
else (
... process in case of wrong MAC )
```

The *validation* of signatures can be analogously performed. For instance, the signature of a message `m` is composed by the hash of the message encrypted with the respective private key: `encryptSK(Hash(m),k)`. Therefore, the validation of such signature requires the respective public key `Pk(k)`.

```
let Res = verifySign(m,encryptSK(Hash(m),k),Pk(k)) in
if Res = validSign then (
 ... process continuation )
else (
... process in case of wrong signature )
```

The *validation* of certificates takes two steps; in the first one the public key and the signature of the certificate are obtained. In the second one, the signature is verified with the public key of the Certification Authority. The code below exemplifies the procedure.

```
let PK = getID(Certificate) in
let signCert = getSign(Certificate) in
let Res = verifySign( PK,signCert, pk_ca) in
if Res = validSign then (
 ... process continuation )
else (
... process in case of wrong certificate )
```

After *message validation*, the receiving process may obviously either forge a new message (phase 1) or wait for a new incoming message (phase 3). When forging a message that corresponds to an answer of the previously received message / data, the newly forged message may also rely on nonce modification, time stamps as well as authentication codes. For example:

```
out (c, (ccu_id, Ack, minus(N1,one), MAC((Ack, minus(N1,one)),SesK)));
```

Main Process

The *Main Process* is intended to start CEs processes with specific parameters. Moreover, the *Main Process* can precise the number of sessions for each process.

The *Main Process* is itself a process and thus is described with operators provided in Tables 10 and 11. The declaration of that process begins with the reserved word `process`. Afterwards, the *declaration* of variables is made. Since such *declarations* are inside of a process, the attacker is unable to know their values. Thus, secret material can be securely defined. If required, a specific identification of CE processes can also be generated during that variable instantiation.

```
new ccu_id; (* identification of ccu *)
new csc_id; (* identification of csc *)
new sk_ecu1; (* declaration of secret key of ecu1 *)
new sk_ecu2; (* declaration of secret key of ecu2 *)
```

Afterwards, the relations between declared variables can be established. Indeed, respective Basic Blocks are required in order to specify such relations.

```
let ecu1_id = host(sk_ecu1) in (* host associated to sk_ecu1 *)
let ecu2_id = host(sk_ecu2) in (* host associated to sk_ecu2 *)
let pk_ecu1 = Pk(sk_ecu1) in (* public key associated to sk_ecu1 *)
let pk_ecu2 = Pk(sk_ecu2) in (* public key associated to sk_ecu2 *)
```

Additionally, public keys are sent on an insecure channel so that the attacker can access to them (they are not meant to be kept secret):

```
out (c, pk_ecu1); (* broadcast public key pk_ecu1 *)
out (c, pk_ecu2); (* broadcast public key pk_ecu2 *)
```

Finally, CEs processes are started. According to the protocol specification, CE processes may be executed concurrently: the operator `|` is used for that purpose. To generate an infinite number of sessions of a process, the operator `!` is used. Process replication is indeed of paramount importance for the verification of security properties. For example the *Main Process* ends with the execution of such processes, that are started an infinite number of times, and in parallel.

```
((!processECU1)|(!processECU2))
```

3.4.2 Properties

Properties are represented with ProVerif *queries*. The semantics of queries is directly related to facts and events in processes. A complete specification of the query grammar is provided in [10]. Our specific description of queries is mainly focused in the ⟨`realquery`⟩ syntax. In this context, the verification of some hypotheses *hyp* based upon a set of facts *gfact* can be directly expressed as:

⟨realquery⟩ := ⟨gfact⟩ ==> ⟨hyp⟩.

Next paragraphs demonstrate how confidentiality and authenticity properties can be modeled.

Confidentiality

In previous sections we described how confidential material is declared (See Hypotheses). With ProVerif, A secrecy assumption on a variable can be automatically verified: Our verification of confidentiality is based on that secrecy assumption. Indeed, CE processes execution determine exchanges that may compromise secrecy of variables. Thus, to determine if confidentiality is preserved for a term *key*, the following query can be used:

```
query attacker:key.
```

Such query leads to a verification process that determines whether the attacker is able to obtain/derive *key*, or not.

Authenticity

The verification of authenticity relies in the concept of injective agreement as proposed in [8]. Roughly speaking, an injective agreement establishes an injective relation between two sets of events. Thus, to prove an authenticity property, events must first be appropriately declared in CE processes based upon the syntax in Table 11. Events can be defined with respect to a term M.

```
event Name(M);
```

Whenever a process flow reached the previous code, the event is executed and a reference to the execution of $eventName(M)$, is recorded. Consequently, a infinite replication of CE processes may lead to an infinite set of executed $eventName(M)$. For the verification of the authenticity of a message sent from `processA` to `processB`, two sets must be considered. The first set contains executed events whose informal semantics is "*I am processA and I have sent a message M to processB*" whilst the second one contains executed events with informal semantics "*I am processB and I have received and validated a message M*". To create the first set, `event sendAToB(M)` is included in `processA`, right before the sending of M:

```
event sendAToB(M); out (c, M);
```

Analogously, `event getAToB(M)` is included in `processB`, right after the reception and validation of M:

```
int (c, M); event getAToB(M);
```

An injective relation from the set of received messages and the set of sent messages proves that all messages M received by processB were necessarily sent by processA. To verify such an injective relation the following query can be used:

```
query evinj:getAToB(x) ==> evinj:sendAToB(x).
```

3.4.3 Attacker

The attacker paradigm is based upon the Dolev-Yao approach which has been previously described and referred. The attacker is represented in ProVerif as an arbitrarily pi-process that can listen to all messages sent on non-private channels. The so acquired knowledge constitutes the basics for attacker reasoning. Thus, from the attacker viewpoint only two processes exist: the attacker process itself, denoted by Q, and the process that represents the overall protocol model, which is denoted by P. Indeed, since P is defined in terms of CE processes, we can say that each CE process is a subprocess of P. The attacker Q is defined as follows [8]:

Definition 12. *S-Adversary*
Let $\mathcal{S}$ a finite set of names used by a process P. A process Q is an $S-adversary$ of P if and only if

1. *The process Q does not include free variables (variables that are not assigned).*
2. *The set of free names in Q, denoted by $fn(Q)$, satisfies $fn(Q) \subset \mathcal{S}$.*
3. *Q does not contain events.*

Definition 12 establishes the fact that the attacker may know free names from P (variables defined with `data`) and thus his initial knowledge is finite. Moreover, the attacker is unable to declare events and use knowledge about executed events. Note, that executed events are not part of the process P but are associated to respective sub process execution sessions.

The reasoning capability of the attacker takes into account that sub processes which compose P may be infinitely replicated with the operator `!`.

The next rules define the operations that an *S-adversary* Q is allowed to perform [8]. We recall that Q is defined in terms of the set of free names $\mathcal{S}$ of P.

1. The *S-adversary* knows an element a if and only if $a \in W_{at}$. W_{at} is the initial knowledge of the *S-adversary*.
2. The *S-adversary* can generate and unbounded number of new names b, unless b is restricted in P or b is a free name in $\mathcal{S}$. Every new name b is included in W_{at}.
3. For each non private constructor f of arity n, if the *S-adversary* knows $M_1, \ldots M_n$ then $f(M_1, \ldots M_n)$ is included in W_{at}.
4. For each non private destructor $g(M_1, \ldots M_n) \rightarrow M$ of arity n, if the *S-adversary* knows $M_1, \ldots, M_n$, then M is included in W_{at}.
5. If $g(M_1^{'}, \ldots, M_n^{'}) \rightarrow M^{'}$ and there exist a substitution σ in the finite set of substitutions of g such that $M_i^{'} = \sigma M_i$, $i = 1, \ldots, n$, and $g(M_1^{'}, \ldots, M_n^{'}) \rightarrow M^{'}$, then $M^{'}$ is included in W_{at}.
6. For every $\bar{c}\langle M \rangle$, if the *S-adversary* knows c then M is included in W_{at}.
7. If c, $M \in W_{at}$ then the *S-adversary* can output $\bar{c}\langle M \rangle$.

In other words, the attacker knowledge is based on message interception, and use of constructors and destructors from data contained in the acquired knowledge.

3.4.4 Formal Verification

To verify a query, ProVerif implements a resolution algorithm that first translates the complete pi-process specification to Horn Clauses [9]. That translation identifies elements of the attacker Q and elements of the protocol P . To verify a query, the resolution algorithm determines, based upon a set of inference rules, if the attacker reasoning is able to derive a trace that contradicts the query, thus proving that the query is false. Otherwise, if the attacker is unable to find such a trace, then the property is satisfied. Additionally, if facts on which the query is based upon are not reachable, the algorithm informs that the query can not be proved. In this subsection we provide a general description of the resolution algorithm. A detailed description can be consulted in [9].

The resolution algorithm that performs the formal verification of queries is split into several stages, namely *Horn clauses transformation, Simplification, Subsumation, Facts Selection, Resolution, Saturation, Backwards Depth-First Search* and *Termination*. In the next paragraphs we present a brief description of those phases.

Horn clauses transformation. The whole pi-process specification is translated into an equivalent Horn clauses specification. Indeed, every pattern in the process syntax in Table 10, is associated to a respective set of Horn clauses. Thus the process P is accordingly translated. Since the attacker Q is a pi-process, its associated reasoning rules are also represented as a set of Horn clauses. The initial set of Horn clauses $\mathcal{R}_0$ is built by the application of the attacker capabilities to the Horn clauses of P. Consequently, $\mathcal{R}_0$ is composed by clauses of the form $H_n \Rightarrow attacker(M_1) \wedge \ldots \wedge attacker(M_k)$ which means that the hypothesis H_n implies that attacker knows the terms $M_1, \ldots, M_k$. Indeed, the hypothesis H_n was directly obtained from the translation of process P.

Simplification. In order to have an efficient algorithm, several simplification functions are applied to the initial set $\mathcal{R}_0$. These functions aim to identify and then eliminate clauses that are unnecessary for process verification without compromising completeness [9]. Thus, the simplification eliminates tautologies; if in a clause $H \Rightarrow C$ the conclusion C is part of the hypothesis H' of another clause, then $H \Rightarrow C$ is removed. Additionally, if the hypotheses H of a clause $H \Rightarrow C$ includes $attacker(M)$ and M is not used elsewhere in the clause, then the $attacker(M)$ is useless and is therefore removed. Finally, the secrecy assumptions are facts F that prune the search space by removing F from clauses $H \Rightarrow C$ where F is part of C. As a conclusion, the simplification algorithm generates a new set of clauses $\mathcal{R}_1$.

Subsumation. A clause $H_1 \Rightarrow C_1$ subsumes $H_2 \Rightarrow C_2$ when there exists a substitution σ such that $\sigma C_1 = C_2$ and the facts of σH_1 are contained in H_2. Thus, the subsumation relation determines clauses that are "contained" by more generic ones. Thus, if $H_1 \Rightarrow C_1$ subsumes $H_2 \Rightarrow C_2$ and both of them are contained in $\mathcal{R}_1$, then $H_2 \Rightarrow C_2$ can be eliminated. After *Subsumation* phase, all the subsumed clauses are eliminated thus generating a reduced set of clauses $\mathcal{R}_2$.

Facts selection. In order to speed up even more the resolution algorithm, a selection function is applied to the set $\mathcal{R}_2$. According to the authors, such technique does

not compromise completeness of verification [9]. The selection function takes a clause $H \Rightarrow C$ and returns a set of facts F that are part of H. The set of facts F will be used in the *Resolution* phase to identify substitution clauses that can be reduced/simplified. Several rules are provided to classify non selectable facts in hypotheses. For instance, since facts of the form $attacker(x)$ with x variable, will be unified with all facts of the form $attacker(y)$, they are classified as non selectable. Moreover, clauses of the form "*if* `event(M)` *has been executed then ...*" are never verified by ProVerif, consequently, facts of the form `event_ex(M)` are not selectable. As a result of the selection process $sel()$, a set of selectable facts F is associated to each clause $H \Rightarrow C$ i.e., $F = sel(H \Rightarrow C)$.

Resolution. Let two clauses, namely $R_1 = (H_1 \Rightarrow C_1)$ and $R_2 = (H_2 \Rightarrow C_2)$. The *Resolution* phase looks for a possible unification between a fact F, with $F \in H_2$, and C_1. If there exists F, then a new inferred clause R_3 can be defined as the application of R_1 and R_2 one after the other. If we denote this unification with σ, R_3 can be written $R_3 = R_1 \circ R_2(F)$ and $R_1 \circ R_2(F) = \sigma(H_1 \cup (H_2 \backslash \{F\})) \Rightarrow \sigma C_2$. The new clause R_3 is equivalent to the two initial clauses R_1 and R_2. However, since the fact F has been removed from H_2, the number of terms as well as the number of clauses is decreased. Note that the *Resolution* process operates over the set $\mathcal{R}_2$. Moreover, for every clause $R = (H \Rightarrow C)$ in $\mathcal{R}_2$, the facts F from H are selected according to the function $sel(H \Rightarrow C)$ described in the previous phase.

Saturation. The *Saturation* phase is an iteration of previous phases that takes $\mathcal{R}_2$ as the initial set. Indeed, For each clause in $R \in \mathcal{R}_2$ such that $sel(R) = \emptyset$ the algorithm looks for clauses $R^{'} \in \mathcal{R}_2$ that can be unified through the *Resolution* procedure. In such case the clauses R and $R^{'}$ are replaced by the derived clause $R \circ R^{'}(F)$. In order to reduce procedure complexity, the clause $R \circ R^{'}(F)$ is *simplified*. After a first iteration, a new set of clauses $\mathcal{R}_3$ is obtained. Afterwards, the *subsumed* clauses of $\mathcal{R}_3$ are eliminated. The *saturation* procedure is exhaustively repeated over subsequent sets $\mathcal{R}_i$ until a fixed point is found. Indeed, a fixed point is obtained if after iteration on clause R the same clause is obtained. The result of this phase is the set of clauses whose hypotheses do not have selectable facts, i.e., $sel(R) = \emptyset$. This set is denoted by $Saturation(\mathcal{R}_0)$. The structure to generate $Saturation(\mathcal{R}_0)$ is stored.

Backwards Depth-First Search. The search is based upon a criterion for derivable facts: a fact F is said to be derivable from a set of clauses $\mathcal{R}$ if and only if there is a clause $H \Rightarrow C$ obtained from $Saturation(\mathcal{R})$ satisfying that F is part of the conclusions C. The algorithm additionally requires that every instance of F can be derived from $\mathcal{R}$ and that the last implication of such derivation is included in $Saturation(\mathcal{R})$. Thus the set $Saturation(\mathcal{R})$ determines patterns for space exploration. To determine if a query R is derivable from the set $\mathcal{R}_0$ with respect to $Saturation(\mathcal{R}_0)$ a backwards depth-first search is performed. Such search takes a *selected* fact F from $Saturation(\mathcal{R}_0)$ and tries to backwardly derive the hypothesis of the clause. The search ends when the query R is subsumed by a clause of $Saturation(\mathcal{R}_0)$ or when the set of selectable facts $sel(R)$ is empty. More particularly, a confidentiality queries directly is verified directly if the clause can be derived from $\mathcal{R}_0$ [3].

Termination. In order to force algorithm termination, some protocol instances should be tagged. Tagging is a technique that includes a tag to every crypto primitive in the model. Thus, differentiation between instances of the same function [11] is ensured. Through this procedure, the algorithm prevents loops generated in deduction of clauses when the *resolution phase* is implemented. For instance, the substitution $minus()$ can be unified with the clause $attacker(encrypt(N, k))$ as follows: $attacker(encrypt(minus(N, one), k)$. Consequently the clause $attacker(encrypt(N, k))$ $\Rightarrow attacker(encrypt(minus(N, one), k)$ is deduced. However the fact $attacker(encrypt(N, k))$ can be eventually derived from $attacker(encrypt(minus(N, one), k)$. Consequently a loop is generated. The inclusion of tags in primitives avoids such possibility. In the previous example, the tagging technique leads to $attacker(encrypt((tag1, N), k))$ and to $attacker(encrypt((tag2, minus(N, one)), k)$. Since $tag1$ and $tag2$ are not part of the possible substitutions, the clauses are not unified thus avoiding the loop.

3.4.5 Limitations and Conclusions

Security properties such as *confidentiality* and *authenticity* can be proved using the ProVerif environment. The verification of queries in ProVerif is based upon a formally proved algorithm. Despite its advantages, the authors recognize some limitations [9]. Indeed, the translation of pi-process models into Horn clauses introduces approximations. As a consequence of those approximations, it is stated that the tool fails to prove protocols that initially need to keep a value as secret and later reveal it [9]. Ans so, the tool is not complete since false attacks may be produced. Also, to our knowledge, a suitable way to model the time is not provided by the framework. Consequently, freshness properties can not be directly represented nor verified.

3.5 AVATAR Verification Approach Overview

The main idea of AVATAR [15] is to integrate in one environment the possibility to describe the system (e.g., the EVITA system), and to be able to perform formal verification of security properties, and automatically generate C Code (e.g., for WP 4000). AVATAR can also be seen as a new version of TURTLE, that addresses drawbacks of the latter, and makes it more adequate for both the formal proof of security properties, and also more adequate for code generation purpose. AVATAR has been defined, and partially implemented in TTool, in the scope of the EVITA project.

3.5.1 System Modeling

General modeling approach

AVATAR is based on SysML Block Diagrams (BD) and State Machine Diagrams (SMD). Basically, AVATAR Block Diagrams are made upon blocks and unidirectional communications between blocks. Communications are asynchronous or synchronous, and they are based on signals carrying values. Blocks can contain other blocks (hierarchical structure), and support other object-oriented paradigms as well (e.g. access rights on attributes and methods). Attributes can be of either *Integer* or *Boolean* type, and data structures can be built upon specific data structure blocks stereotyped << *datatype* >>.

The behavior of each block is defined with a State Machine Diagram. An AVATAR State Machine Diagram supports all operators, apart from the history one. AVATAR also supports hierarchical states, but do not support the deep or swallow reactivation of states. Last but not least, AVATAR SMD distinguishes between functional delays $after(d_{min}, d_{max})$ and computational delays $computeFor(d_{min}, d_{max})$. The semantics of AVATAR has been first defined in UPPAAL, but the semantics of a subset of the profile has also been defined in the pi-calculus language supported by ProVerif. This semantics is further explained in Section 3.5.4.

AVATAR-Sec: a subset of AVATAR

AVATAR-Sec is the subset of AVATAR that is relevant for achieving security proofs. Elements in AVATAR, but not in AVATAR-Sec, are simply ignored when translating an AVATAR modeling to ProVerif.

- **Communications**. AVATAR communications are based upon unidirectional one-to-one synchronous or asynchronous communications. In AVATAR-Sec, only asynchronous communications are supported. Since all communications are assumed to be private in AVATAR, AVATAR-Sec assumes that each block has defined two signals: *chout* and *chin*, which can be respectively used for receiving and sending purpose. No channels need to be declared for using those signals, which are considered to be sent on / received from a public broadcast channel common to all blocks.

- **Attributes**. The type of attributes is not used in AVATAR-Sec. The modification of variables in an AVATAR-Sec SMD is thus ignored. However, data structures can be declared and used, for information purpose, e.g., defining a key or a message simplifies the modeling task.

- **Methods**. Each block of AVATAR-Sec contains a pre-defined list of cryptographic functions (e.g., *decrypt*(), *encrypt*(), $MAC()$, etc.). Other methods can be declared by blocks, but they are translated as ProVerif *events*.

- **Logical operators of SMD**. Apart from variable assignation, all operators are supported i.e., *start states*, *stop states*, *states*, *hierarchical states*, *choices*, *sending of signals*, *receiving of signals* and *method calls*.

- **Temporal operators of SMD**. They are not supported by AVATAR-Sec.

Modeling knowledge

Knowledge sharing, at the beginning of a protocol run, is a very important point for achieving security proofs. However, object-oriented models do not support such sharing, making it mandatory to modify AVATAR for that purpose. For such modeling elements which do not fit at all with the object-oriented paradigm, the best options is probably to complement Block Diagrams with notes containing specific pragmas.
Finally, the following pragma can be used in AVATAR-Sec Block Diagrams:

```
♯ InitialCommonKnowledge BlockID.attribute (blockID.attribute)*
```

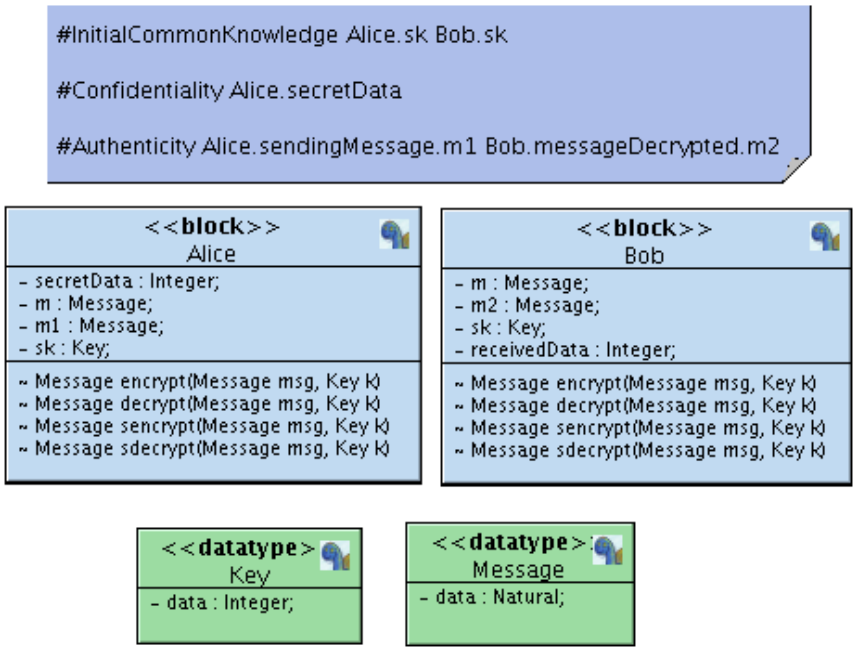


Figure 8 Example of an AVATAR Block Diagram

And, for example, to model that the key $k1$ of Block $block1$ is the same as the key $k2$ of Block $block2$ at the beginning of the system execution:

```
♯ InitialCommonKnowledge block1.k1 block2.k2
```

Figure 8 contains an AVATAR Block Diagram. The latter has two blocks (*Alice* and *Bob*), and declares two data structures (*Key* and *Message*). Alice and Bob declares a set of cryptographic methods (all of them are not visible on the diagram). Also, the note declares that *sk* is a pre-shared data (a key) between Alice and Bob:

```
♯ InitialCommonKnowledge Alice.sk Bob.sk
```

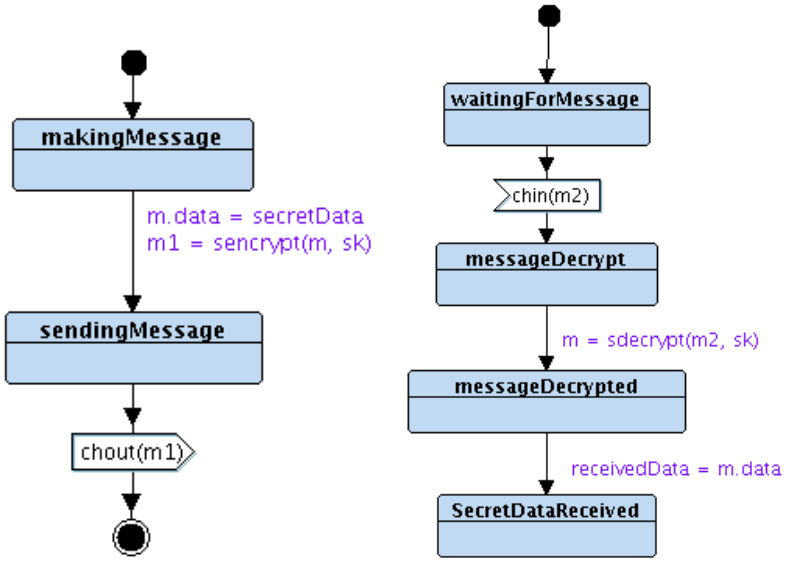
The behavior of *Alice* and *Bob* is provided into two respective State Machine Diagrams (see Figures 9-a and 9-b, respectively). Alice first puts its $secretData$ into a message $m.data = secretData$, then encrypt this message $m1 = sencrypt(m, sk)$ with the symmetric encryption function, and finally sends the resulting message on the broadcast channel $chout(m1)$. *Bob* waits for a message on the broadcast channel $chin(m2)$. Then, *Bob* tries to decrypt the received message $m = sdecrypt(m2, sk)$ and then extracts from the message the $secretData$ sent by Alice $receivedData = m.data$.

3.5.2 Properties

Confidentiality and authenticity can be directly expressed at Block Diagram level.

Confidentiality

Confidentiality in AVATAR-Sec can be modeled as a simple pragma provided in the note of a Block Diagram. The confidentiality must be specified as the confidentiality of



a – Alice b – Bob

Figure 9 State Machine Diagrams of Alice and Bob in AVATAR

an attribute of a block:

```
♯ Confidentiality block.attribute
```

Coming back to the example provided in Figure 8, the following statement provided in the note models that the *secretData* attribute of *Alice* shall remain confidential i.e. never accessible to an attacker:

```
♯ Confidentiality Alice.secretData
```

Authenticity

Authenticity in AVATAR-Sec can be also modeled as a pragma given in the note of a Block Diagram. This authenticity is meant to prove that a message received by a block $block2$ was really sent by a block $block1$. To model so, the authenticity pragma must specify two states, one of the sender block, i.e. one state $s1$ of $block1$, and one state $s2$ of $block2$. Also, for the authenticity to be proved with ProVerif, $s1$ shall be put right **before** the sending of the message $m1$ for which authenticity is to be proved. Analogously, $s2$ shall be put right **after** message $m2$ – equal to $m1$ – has been received (and validated, see section 3.4.2). Finally, the authenticity pragma is as follows:

```
♯ Authenticity block1.s1.m1 block2.s2.m2
```

For example, in Figure 8, the authenticity statement models the fact that all messages $m1$ sent by *Alice* after state *sendingMessage* shall be authentic for *Bob* receiving it into a message named $m2$ before its state *messageDecrypted*.

```
♯ Authenticity Alice.sendingMessage.m1 Bob.messageDecrypted.m2
```

3.5.3 Attacker

In AVATAR-Sec, the attacker model is implicit i.e. there is no need to model it either at Block Diagram level, or at State Machine Diagram level. More precisely, the attacker model AVATAR-Sec relies on is the one of ProVerif, see section 3.4.3.

3.5.4 Formal Verification

TTool now fully supports AVATAR-Sec, and therefore implements a press-button approach for verifying confidentiality and authenticity security properties from AVATAR-Sec models. Briefly, the AVATAR-Sec translation process is as follows:

Definition 13. *AVATAR-Sec translation process*
Let $\mathcal{T}$ the translation process that takes as input a Block Diagram BD, and a set of pragmas P, and Pr the resulting ProVerif specification:
$Pr = \mathcal{T}(BD, P)$.

- A BD contains a set of attributes, a set of functions, a set of signals and a reference to a State Machine Diagram (SMD).
- An SMD is a set of interconnected logical operators: start state, stop states, transitions – with attribute settings and function calls -, choices, states, sending in a channel, receiving in a channel.
- The type of a pragma is either $InitialCommonKnowledge$, $Confidentiality$, or $Authenticity$.

$\mathcal{T}$ applies the following set of rules:

1. For each block $b \in BD$, a "first" process fp is generated. Then, for each state s of the State Machine Diagram smd of b, another process ps is generated.
2. fp instantiates all attributes that are not referred in $InitialCommonKnowledge$ or $Confidentiality$ pragmas are created: $new\ attr$;. Then, fp makes a call to the ps process corresponding to the start state of smd.
3. Each ps is created as follows. An event is first called for tracing the reachability of states $Event\ entering\_state\_nameofs()$;. Then, each branch of logical operators linked from s is considered until another state is reached on that branch:
   - Sending on a channel c of a message m is translated as a $out(c, m)$;.
   - Receiving on a channel c of a message m is translated as a $in(c, m)$;.
   - The affectation of a variable is translated using a *let* operator, e.g.: $let\ m1.data = (m2.data, m3.data)$;.
   - The call of a cryptographic function is translated with a ProVerif cryptographic function and a *let* operator: $let\ themac = MAC(msg1.data, keyOfGroup.data)$;.

- The call of a non-cryptographic function is translated with a simple call to an event of the same name of the function, and with the same parameters, e.g., $Event\ function(par0, par1);$.
- The various branches starting from state s are selected using the $if...else$ ProVerif statement.

4. The main process mp of the ProVerif specification instantiates all attributes listed in $InitialCommonKnowledge$ pragmas. Then, it instantiates in parallel, and for an infinite number of sessions, all fp processes, e.g., $(!fp_1)|(!fp_2)|...|(!fp_n)$.

5. $Confidentiality$ pragmas referencing a block b and an attribute $attr$ of b are translated as a declaration of $attr$ as follows: $private\ free\ attr.$ and with a query of the following form:
$query\ attacker : attr.$

6. $Authenticity$ pragmas of the form $b1.state1.attr1b2.state2.attr2$ are translated using statements of the following form: $query\ evinj : b2\_state2(attr2) \Rightarrow evinj : b1\_state1(attr1).$.
Additionally, in the process ps where $s = s1$, a call to $Event\ b1\_state1(attr1)$ is added at the beginning of the process. Similarly, a call to $Event\ b2\_state2(attr2)$ is added at the beginning of the process ps where $s = s2$.

TTool executes $\mathcal{T}$ at the push of a button. At first, several options are proposed for the code generation, in particular how choices are taken into account. Then, TTool makes a call to ProVerif, and outputs the following results (see for example Figure 10, the result provided by TTool when applying the verification on the Alice-Bob example):

- For each $Confidential$ pragma, an information is provided to indicate whether the data provided in the pragma was proved to be secret, or not.
- For each $Authentitiicy$ pragma, an information is provided to indicate whether the authenticity holds, or not.
- For each state s of each smd, and information is provided to indicate whether the s is reachable, or not.
- Each query that could not be proved is also listed. An option makes it possible to obtain the trace leading to the query violation: the trace is the raw trace output by ProVerif.

3.5.5 Limitations and Conclusions

AVATAR-Sec makes it possible to prove security properties from a model made with a well-known modeling approach i.e, structuring the system with a SysML Block Diagram, and describing the behavior of each block with a State Machine Diagram. No or little knowledge of ProVerif is necessary to perform first proofs. Adaptations of this AVATAR-Sec shall nonetheless be studied for supporting more AVATAR operators, and for proving more security properties (e.g., integrity, freshness).

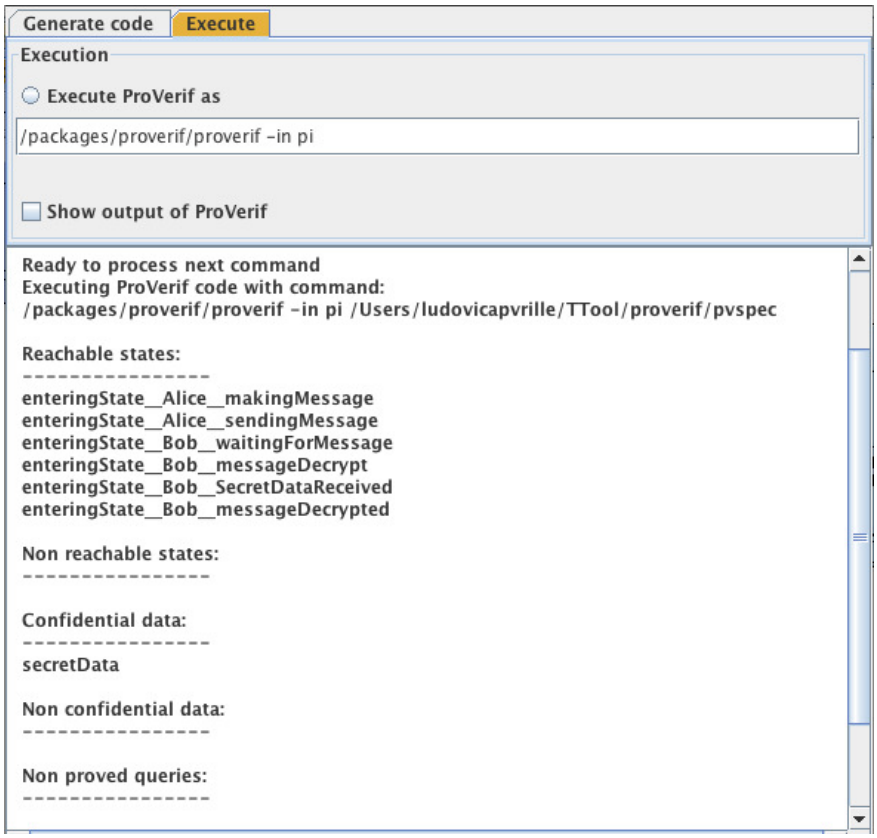


Figure 10 Example of verification results as displayed by TTool

3.6 Keying Protocol with Key Master

3.6.1 Protocol Description

The Keying Protocol with Key Master ECU aims to securely distribute a randomly generated key among the members of a so called group of Electronic Control Units (ECUs). The key to be distributed is referred as Session Key (SesK). In our description, the ECU that creates the SesK is referred as generator. When the protocol is triggered, the generator creates the SesK and sets its flag to 'verify' (*use_flag=verify*, see [19], Key Data Structures). Thus, other ECUs in the group are enforced to use the SesK only for MAC verification. Afterwards, the SesK is sent to the Key Master for group distribution. Since the Key Master owns the Pre-shared Symmetric Key (PSK) of every ECU in the group, the generator encrypts the SesK with its PSK. Such message includes a time stamp and is finally protected with a MAC (Message 1). After reception, the Key Master verifies Message 1 and in case of a valid request, the SesK is imported into its HSM. From this point, the Key Master is responsible for SesK distribution. Consequently, the SesK is protected with the PSK of the respective target ECU. Such message is time stamped and MAC protected (Message 2). After reception of Message 2, the target ECU verifies its validity and afterwards imports the new SesK into its HSM. Finally, a message including an acknowledgement flag (ACK) is sent by the target ECU to the Key Master thus informing SesK acceptance (Message 3). Message 3 also includes a time stamp and is MAC protected. The Key Master receives the acknowledgement and a security check is performed. The Key Master should repeat the just described procedure for every ECU in the group (different from the generator). After SesK distribution, the Key Master informs the results to the generator (partial or total accomplishment). The message includes the respective ACK code, the time stamp and is MAC protected (Message 4). Finally, after reception of Message 4, the generator verifies message validity and afterwards the protocol ends.

Once distributed, the SesK key allows for authentic unidirectional communications between the generator and the rest of the group. Indeed, the generator can efficiently broadcast messages that are MAC protected with the SesK thus ensuring its authenticity. Additional information of this protocol can be found in EVITA technical report D3.3 [18]. A figure of the just described sequence is presented in 11.

3.6.2 Targeted Security Properties

According to the specification provided in EVITA D2.3 [17], we specifically target the following Security Requirements:

1. *Authenticity_101* (EVITA D2.3 [17], page 31)
2. *Integrity_104* (EVITA D2.3 [17], page 34)
3. *Freshness_102* (EVITA D2.3 [17], page 37)
4. *Confidentiality_101* (EVITA D2.3 [17], page 42)

The rest of the section is dedicated to the proof of whether these security requirements are satisfied – or not - in the Keying Protocol.

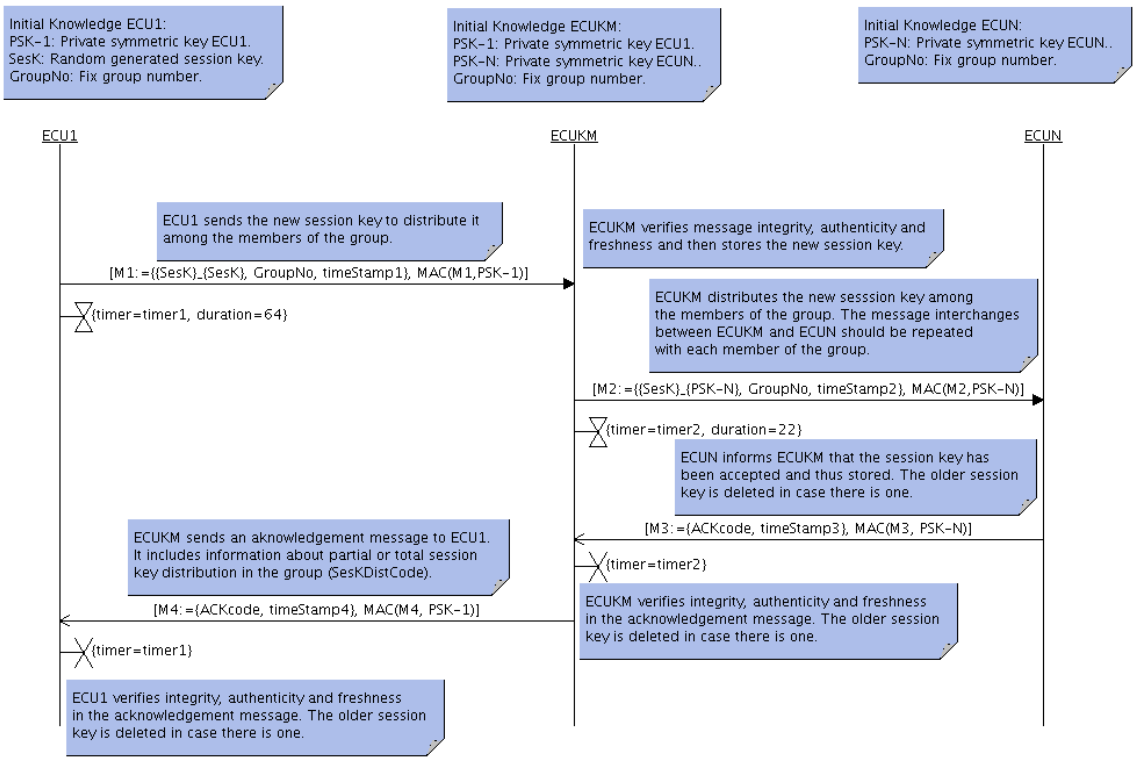


Figure 11 Sequence Diagram of the Keying Protocol

3.6.3 Model for Verification of DoS (TURTLE)

We strongly participated to the definition of protocols. The proof of this is the section on protocol evaluation in [18]. If the document is dedicated to protocols as defined in [18], the definition of protocols was an iterative process, with brainstorming on various versions of protocols, during which models and verification techniques were used to identify first flaws, and correct them accordingly. An example of this concerns the Key Master Protocol, and more particularly denial of Service attacks. Indeed, if an attacker intercepts messages from Key Master to destination ECUs, then, the protocol was stuck, and the Security Watch Dog could not be informed of the problem. Thus, the protocol was enhanced with the use of timers, the use of a retransmission counter, and information sent to the security Watchdog whenever a problem was encountered when distributing the session key. For example, a first version of the protocol, sensitive to DoS attacks between KM and ECUs, is exemplified in the trace provided in Figure 12. The second version corresponds to a corrected version of the first trace, see Figure 13.

TURTLE Analysis Diagrams (i.e., TURTLE Sequence Diagrams) were often used to identify similar flaws.

3.6.4 Model for Verification of Integrity (TURTLE)

The elements that compose the model are briefly described in the following list (See Figure 14). Complementary explanations on the use of TURTLE for proving security properties can be found in section 3.3.

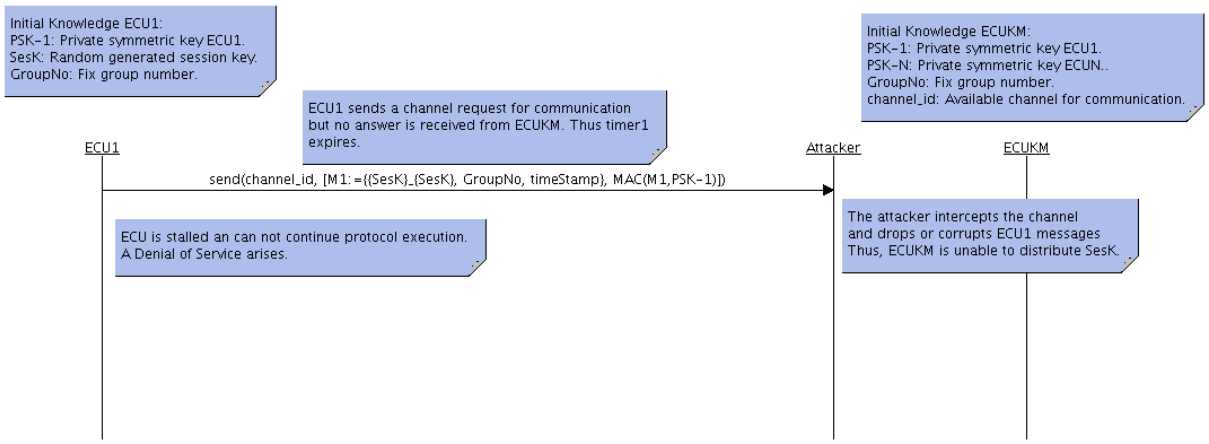


Figure 12 Keying Protocol scenario without timers

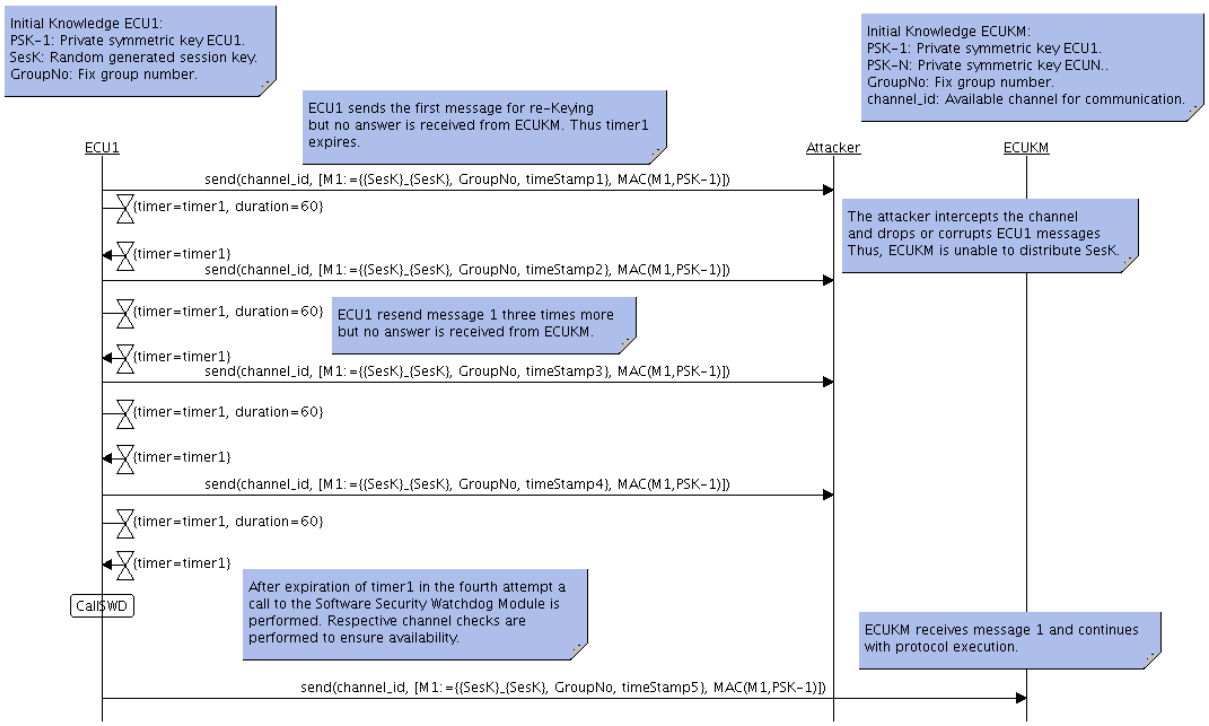


Figure 13 Improved version of Keying Protocol scenario

PDU: The data structure for messages in the protocol.

Parameters: The data structure that includes the parameters of the model.

ECU1: The class that represents the SesK generator ECU.

ECUKM: The class that represents the Key Master ECU in the group.

ECUN: The class that represents a target ECU for SesK distribution.

CAN: The communication channel.

CryptoLibrary: This class includes the crypto primitives in the model.

UTC: A class representing a model for the time.

IntegrityObserver: The class that performs verification of integrity in the model.

The hypotheses for this model were already presented in a previous section (See 3.1.2). We do not consider additional hypotheses. The definition of the *Integrity Observer* is based upon the previous approach presented in section 3.3.4. Indeed, the *Security Observer* verifies that messages remain unchanged between two observations. The first observation is taken when a message is sent whilst the second one when the message is received (See Figure 15). Since the relation between the *Integrity Observer* and principals is trusted, he is able to retrieve correct information about concerned events. The comparison between sent and received messages is based upon the whole PDU contents. The property is satisfied if no changes are identified between sent and received messages. In such a case the respective gate is activated. The *Integrity Observer* is able to identify if a message that was sent is not received. In the case of PDU lost, the respective gate is activated. Such integrity fault is assumed as a characteristic of the communication channel.

3.6.5 Results for Verification of Integrity (TURTLE)

The results are shown in Table 15.

3.6.6 Model for Verification of Authenticity (TURTLE)

The verification of authenticity takes the same TD model as in the verification of Integrity (See 3.6.4). However, the Integrity Observer is removed and the next elements are added to the base model (See Figure 14):

AuthenticityObserver: The class that performs verification of authenticity in the model.

The hypotheses for this model were already presented in a previous section (See 3.1.2). We do not consider further hypotheses. The definition of this Observer is based upon the approach presented in section 3.3.4. The Authenticity Observer retrieves information whenever a message is sent and received. This Observer establishes definitions for claimed and real author. Through those definitions the Observer is able to know the real source

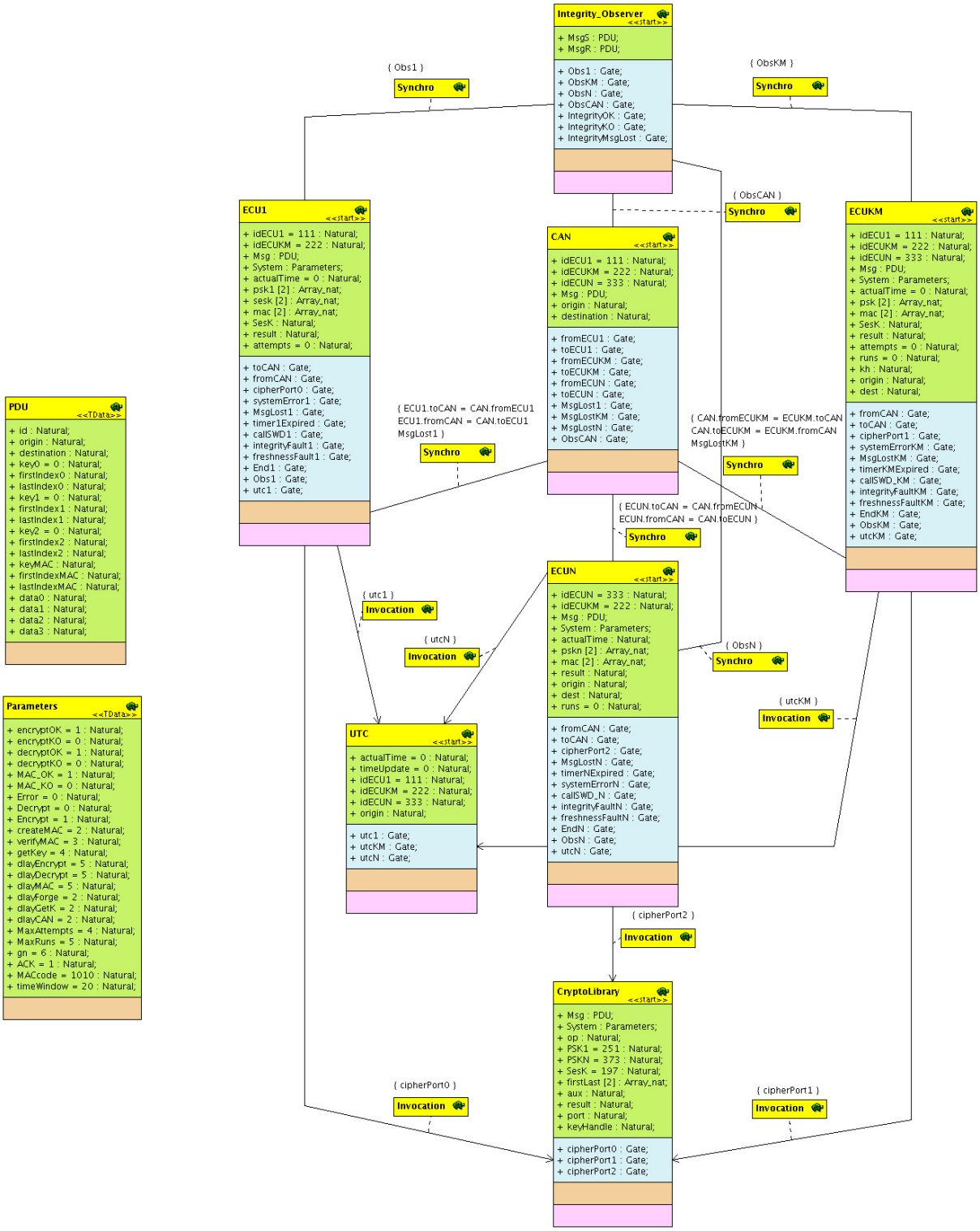


Figure 14 Overview of the TCD model for the Keying Protocol

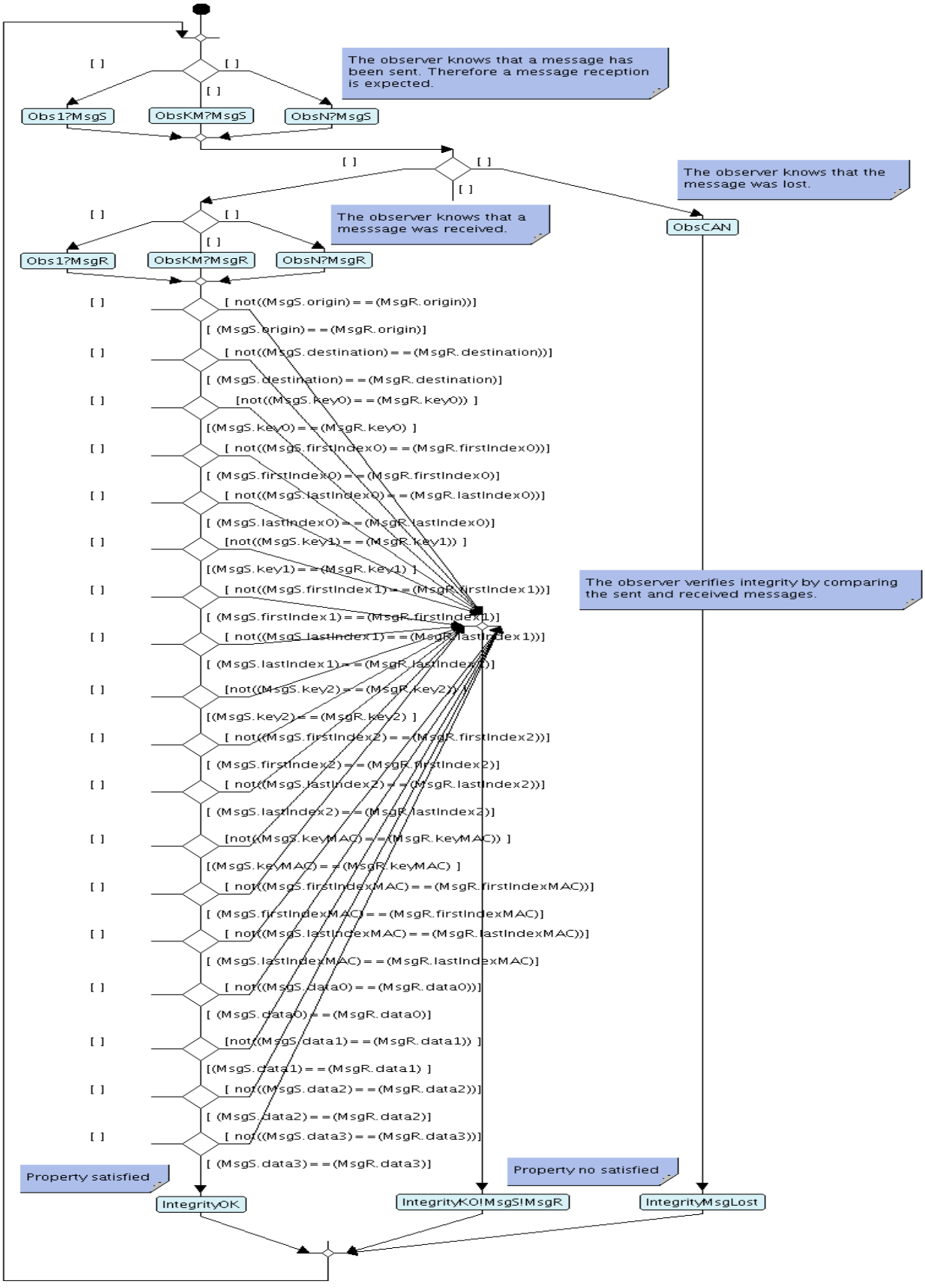


Figure 15 Activity Diagram of the Integrity Observer for the Keying Protocol

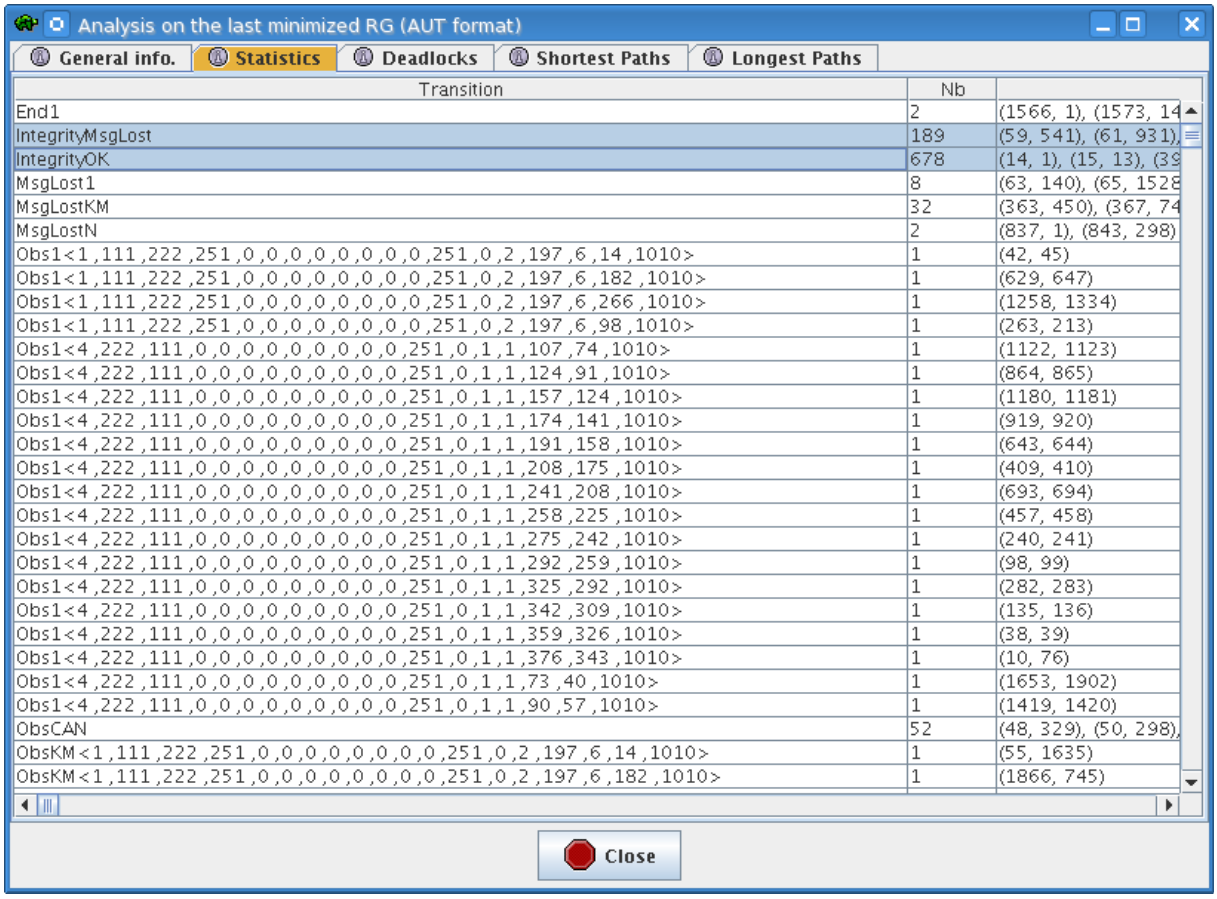


Figure 16 Statistical results for verification of Integrity in the Keying Protocol

Table 15 Results for verification of Integrity in the Keying Protocol

Verification Scheme	TCD model and LOTOS Reachability Graph
File model	KeyingProtocolKMasterIntegrity.xml
Property representation	Integrity Security Observer
Gate for "Property Satisfied"	*IntegrityOK*
Gate(s) for "Property Not Satisfied"	*IntegrityKO*
Additional gates	*IntegrityMsgLost*
Nb. of states	1904
Nb. of transitions	2769
Nb. of transitions *IntegrityOK*	678
Nb. of transitions *IntegrityKO*	0 (See Figure 16)
Nb. of transitions *IntegrityMsgLost*	189
Observations	The integrity property is satisfied, however the possible interactions with a generic attacker should be considered.

of a message and to determine if the claimed author truly corresponds with such real source (See Figure 17). In such a case the property is satisfied and the respective gate is activated. We assume that the Observer knows the owner of a Pre-shared Symmetric Key (PSK). Thus, the claimed author corresponds to the owner of the PSK that was used to MAC the message. Since the relation between the Authenticity Observer and principals is trusted, the Observer is able to retrieve correct information about real and claimed authors. A relevant characteristic of our Authenticity Observer is that it requires message integrity.

3.6.7 Results for Verification of Authenticity (TURTLE)

The results are shown in Table 16.

3.6.8 Model for Verification of Freshness (TURTLE)

The verification of freshness takes the same TD model as in the verification of Integrity (See 3.6.4). As in the verification of authenticity, the Integrity Observer is removed and the next elements are added to the base model (See Figure 14):

FreshnessObserver: The class that performs verification of freshness in the model.

The hypotheses for this model were already presented in a previous section (See 3.1.2). We do not consider additional hypotheses. The definition of this Observer is based upon the approach already presented in section 3.3.4. The *Freshness Observer* works in parallel with the *UTC time* model; whenever a message is sent or received, the Observer knows the UTC times for those events. Additionally the Observer stores previously received messages in a list. Thus he is able to compare the just received message with previous

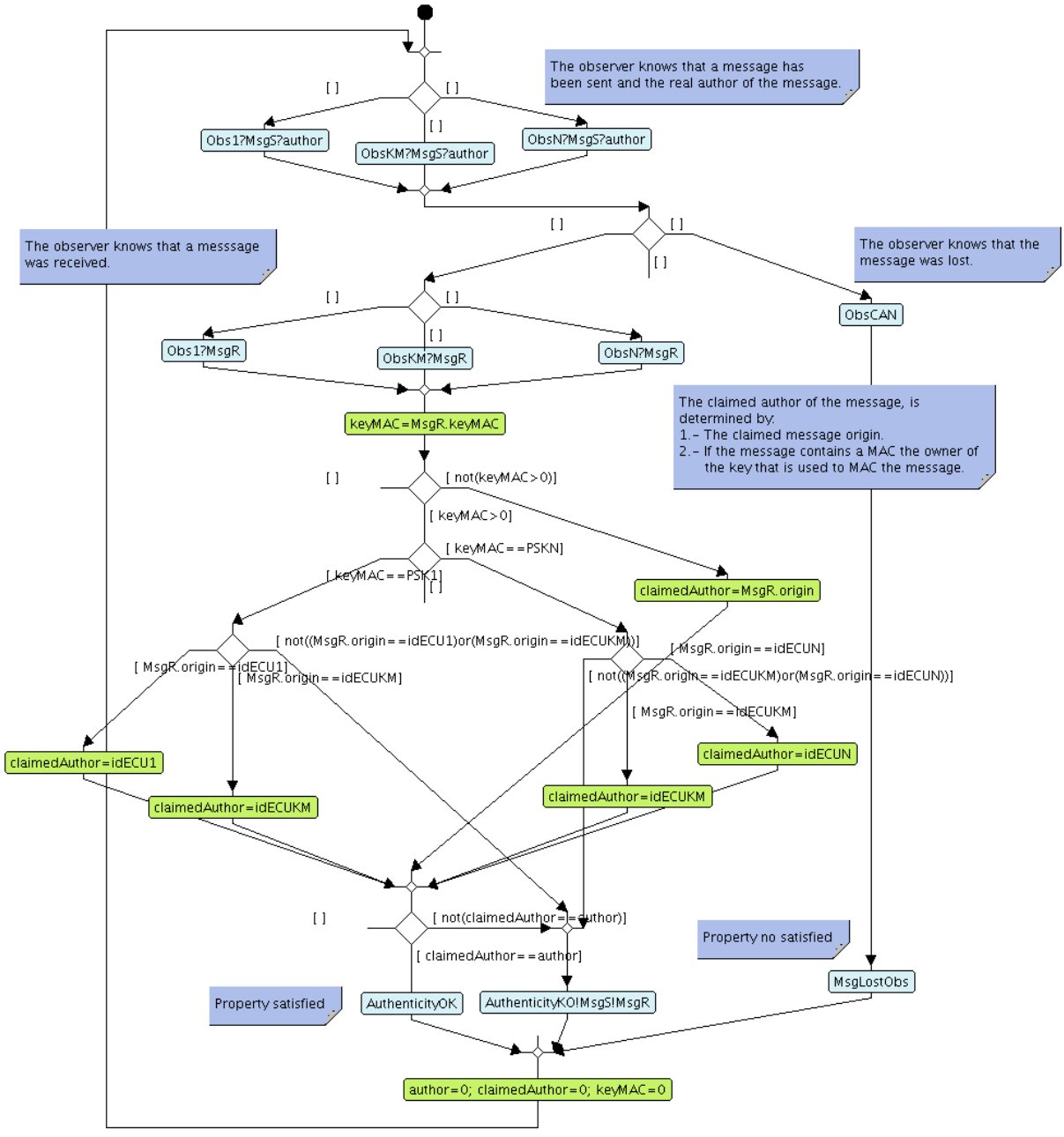


Figure 17 Activity Diagram of the Authenticity Observer for the Keying Protocol

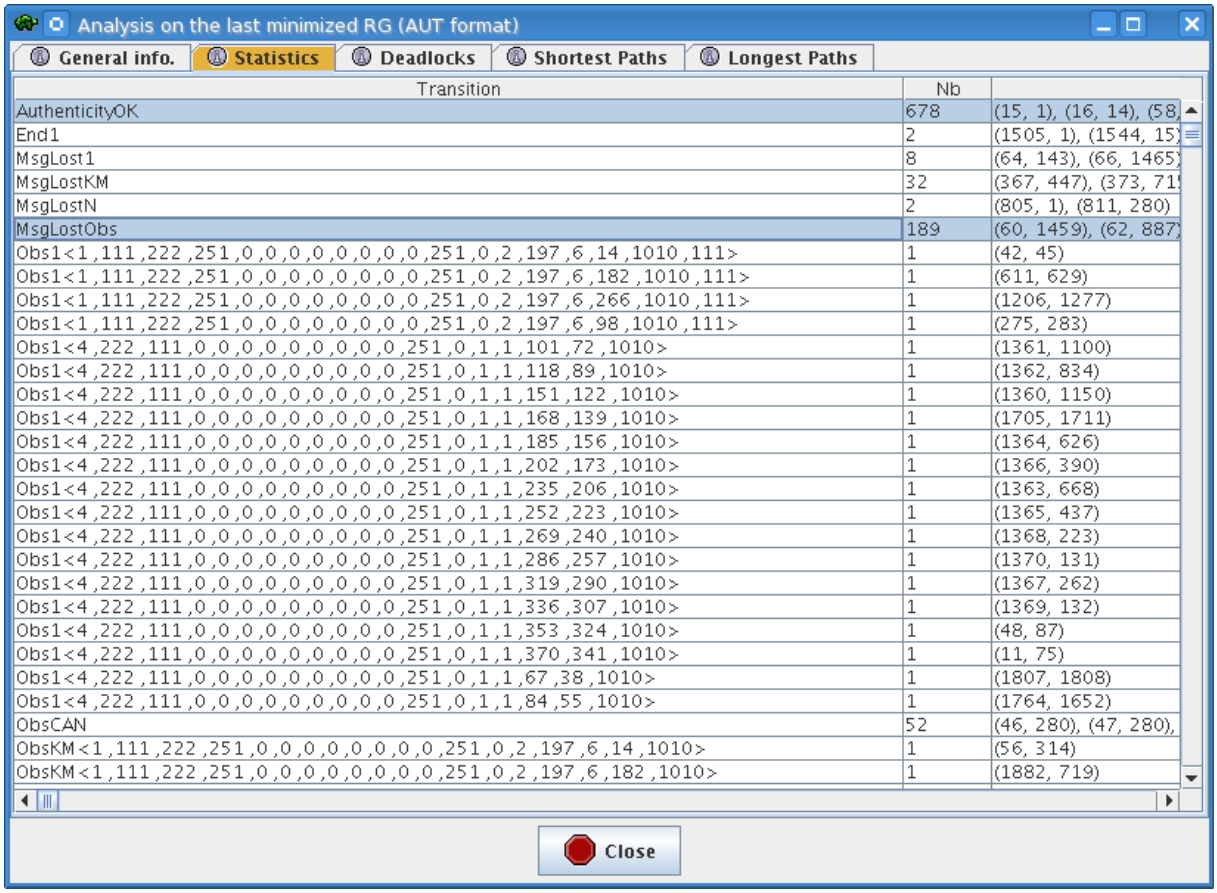


Figure 18 Statistical results for verification of Authenticity in the Keying Protocol

Table 16 Results for verification of Authenticity in the Keying Protocol

Verification Scheme	TD model and Reachability Graph generation and analysis using the LOTOS code generator of TTool and CADP
File model	KeyingProtocolKMasterAuthenticity.xml (TTool file)
Property representation	Authenticity Security Observer
Gate for "Property Satisfied"	*AuthenticityOK*
Gate(s) for "Property Not Satisfied"	*AuthenticityKO*
Additional gates	*MsgLostObs*
Nb. of states	1904
Nb. of transitions	2769
Nb. of transitions *AuthenticityOK*	678
Nb. of transitions *AuthenticityKO*	0 (See Figure 18)
Nb. of transitions *MsgLostObs*	189
Observations	The Authenticity property is satisfied, however the interactions with a generic attacker should still be considered.

ones. The reference for comparison is the time stamp. If the received message is in the list then the property is not satisfied and the respective gate is activated (See Figure 19). If the message is not in the list, then an evaluation is performed; if the difference between the reception time and the time stamp is less than a given threshold then the property is satisfied and consequently the respective gate is activated. Received messages are always stored in the Observer list. When the list is full, the index is reinitialized and the list is thus overwritten (freshness window). Since the relationship between the Freshness Observer and the principals is trusted, the Observer is able to retrieve correct information about concerned PDU's. Additionally, if the message is lost the respective gate is activated. A characteristic of our Freshness Observer is that it requires message integrity.

3.6.9 Results for Verification of Freshness (TURTLE)

The results are shown in Table 17.

3.6.10 Model for Verification of Confidentiality (ProVerif)

The ProVerif model can be separated in the next main sections. Complementary explanations can be found in section 3.4.

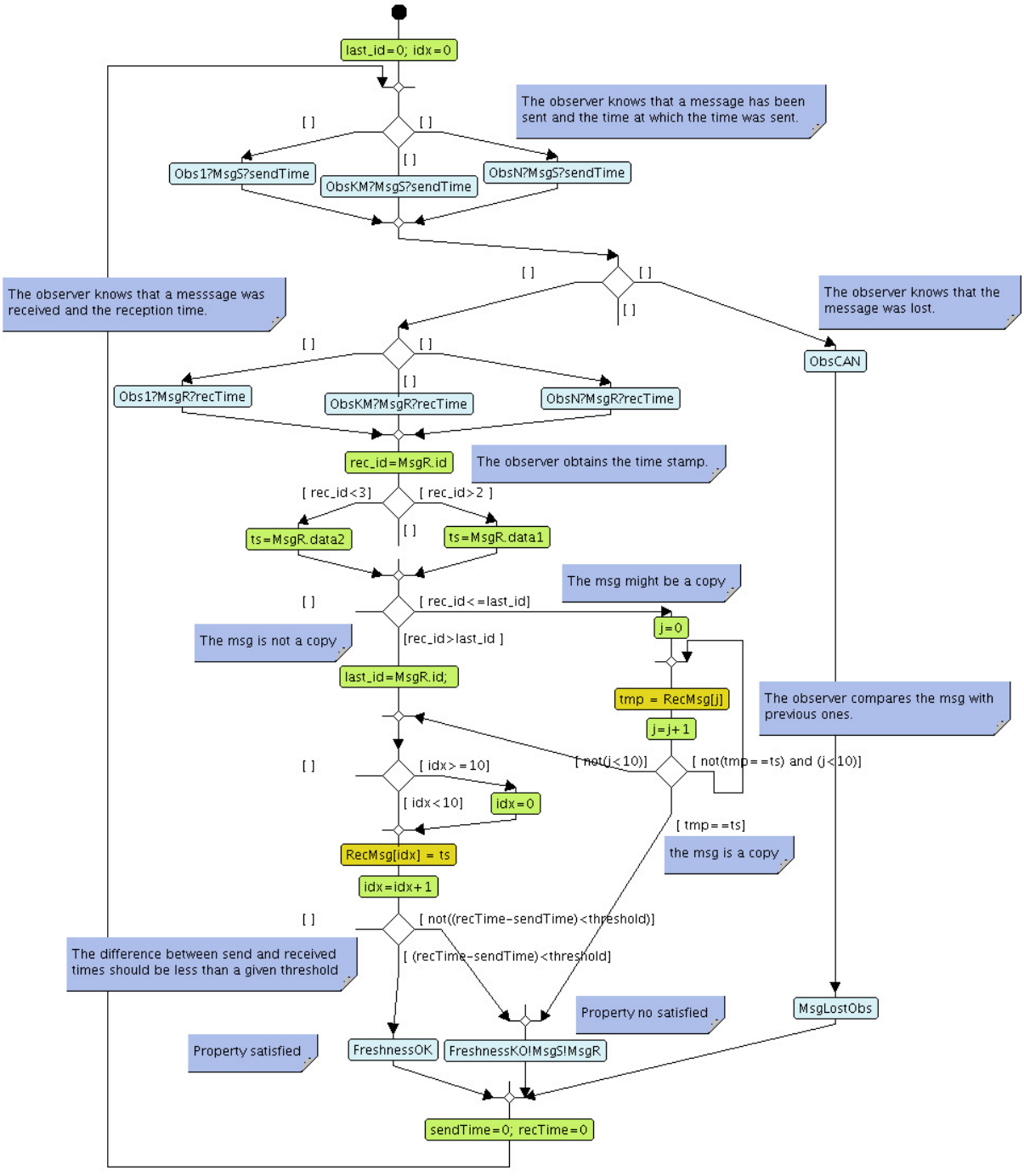


Figure 19 Activity Diagram of the Freshness Observer for the Keying Protocol

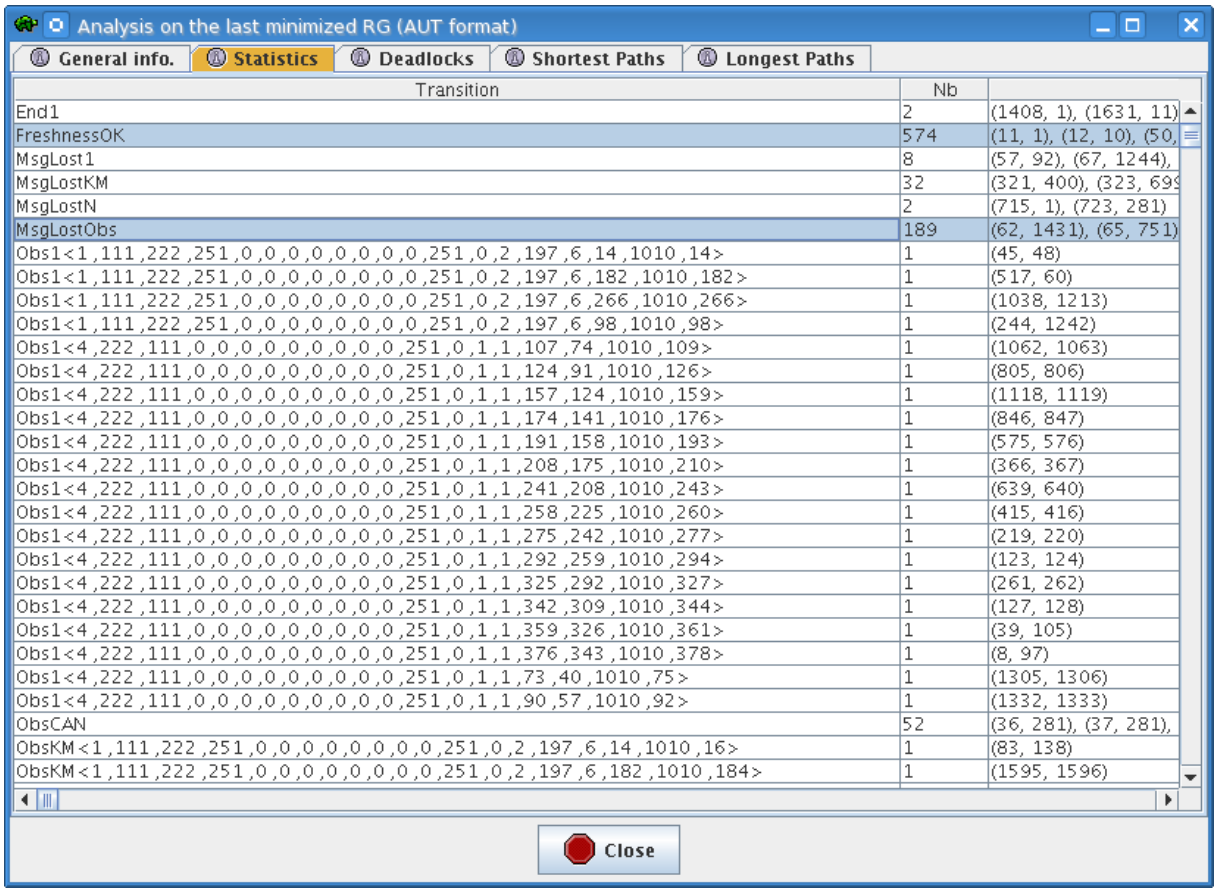


Figure 20 Statistical results for verification of Freshness in the Keying Protocol

Table 17 Results for verification of Freshness in the Keying Protocol

Verification Scheme	TD model and Reachability Graph generation and analysis using the LOTOS code generator of TTool and CADP
File model	KeyingProtocolKMasterFreshness.xml (TTool file)
Property representation	Freshness Security Observer
Gate for "Property Satisfied"	*FreshnessOK*
Gate(s) for "Property Not Satisfied"	*AuthenticityKO*
Additional gates	*MsgLostObs*
Nb. of states	1800
Nb. of transitions	2561
Nb. of transitions *FreshnessOK*	574
Nb. of transitions *FreshnessKO*	0 (See Figure 20)
Nb. of transitions *MsgLostObs*	189
Observations	The freshness property is satisfied in the model, however interactions with a generic attacker should still be considered.

Basic Blocks: This section includes the crypto primitives and related model functions. Additionally it can include the predicates, clauses and equations which conform the basic structures in the model.

Hypotheses: Includes the model for secrecy assumptions. Such secrecy assumptions establish the information that is initially restricted to the attacker.

Queries: This section contains a list of queries that are addressed to verify the targeted properties.

Processes: Includes a list of communicating processes which determines the protocol definition. In almost all cases each process represents a principal in the protocol.

Main: Since the processes may be defined through parameters (free variables and names), this section is intended to provide specific values for such parameters. Moreover, this section determine an order for processes execution and the number of sessions for each process. Thus, the *Main* section implements a specific instance of the model.

1. Basic Blocks
   The *Basic Blocks* for the model are presented bellow. These blocks are almost generic and are used whenever a Security Protocol is modeled at this level.

```
(* Each agent in the protocol has a group *)
.
data group/1.
```

```
data true/0.
.
reduc get1(group((x1,x2,x3)))=x1.
reduc get2(group((x1,x2,x3)))=x2.
reduc get3(group((x1,x2,x3)))=x3.
.
 (* Each agent shares a PSK with ECU-KM *)
.
fun host/1.
private reduc getKey(host(x))=x.
.
 (* To decrypt/encrypt messages with a key *)
.
fun encrypt/2.
reduc decrypt(encrypt(x, k),k)=x.
.
 (* MACs in messages *)
.
fun MAC/2.
fun verifyMAC/3.
.
equation verifyMAC(m, MAC(m, k), k)=true.
reduc getMsg(MAC(m,k))=m.
```

.

Note that all the basic blocks can be used by the attacker excepting `getKey()` which is a private block and can only be used by principals. This restriction implies that the attacker is unable to derive a Pre-shared Symmetric Key (PSK) from a ECU name.

2. Hypotheses
   The section *Hypotheses* expresses the fact that all the Symmetric Preshared Keys (PSK's) and the SesK are initially unknown by the attacker. Indeed, the syntax of such semantics is implicitly related with the initial attacker knowledge, thus `not psk1.` implies that the attacker ignores `psk1`. The content of this section is shown bellow.

```
 (* Initial secrecy assumptions ( the keys can not be derived from clauses
) *)
.
not psk1.
not pskn.
not SesK.
```

.

3. Queries
   The queries for verification of Confidentiality directly question whether the confidential data can be derived/inferred by the `attacker`, or not. Such queries are

addressed in the resolution algorithm implemented in ProVerif. Moreover, queries verification is based upon the given hypotheses and the exchanges in the protocol. The section *Queries* is presented next.

```
 (* Queries for Verification of Secrecy/Confidentiality *)
.
query attacker:SesK;
 attacker:psk1;
 attacker:pskn.
```

.

4. Processes
   In agreement with the protocol description, the section *Processes* includes a process for ECU1, a process for ECUKM and a process for ECUN. Additionally, a process for the Security Watchdog Module (SWD) is included. Note that the communication channel `c` is an insecure one (Dolev-Yao approach), therefore the attacker knows every exchange between principals. Additionally note that the SWD process send alerts through a secure channel `c1`. The definition of the just mentioned processes is presented in line. The semantics of processes captures the agents behavior in the protocol.

```
 (* The process for ECU1 *)
.
let processECU1 =
.
 (* Msg 1:  Sends the new SesK to ECUKM for distribution among the members
of the group *)
.
 new ts1;
 new SesK;
 out(c, (ecu1,encrypt(SesK,psk1), gn, ts1, MAC((encrypt(SesK,psk1), gn,
ts1),psk1)));
.
 (* Msg 4:  Receives an acknowledgement informing the SesK distribution *)
.
 in(c, m4);
 let (hostX, ACK4, TS4, MAC4)=m4 in
.
 if hostX=ecukm then
 let Res4=verifyMAC((ACK4, TS4),MAC4, psk1) in
 if Res4=true then
.
 (* Use of the SesK in ECU1*)
.
 if ACK4=Ack then
 if TS4<>ts1 then
.
 new ts5;
 out(c, (ecu1,gn,ts5, MAC((gn,ts5),SesK)));
```

```
 out(c, (ecu1,gn,ts5, MAC((gn,ts5),SesK))).
.
 (* The process for ECUKM *)
.
let processECUKM =
.
 (* Msg 1:  The KM receives the distribution request from an ECU *)
.
 in(c, m1);
 let (hostY, Encrypt1, GN1, TS1, MAC1)=m1 in
 let PSKY=getKey(hostY) in
 let Res1=verifyMAC((Encrypt1, GN1, TS1),MAC1, PSKY) in
 if Res1 = true then
.
 if GN1=gn then
 let SESK=decrypt(Encrypt1,PSKY) in
.
 (* Msg 2:  The KM distributes the SesK among the ECUs of the group *)
.
 let hostZ=get3(gn) in
 if hostZ<>hostY then
 let PSKZ=getKey(hostZ) in
 new ts2;
 out(c, (ecukm,
encrypt(SESK,PSKZ),GN1,ts2,MAC((encrypt(SESK,PSKZ),GN1,ts2), PSKZ)));
.
 (* Msg 3:  The KM receives an acknowledgement from ECUN *)
.
 in(c, m3);
 let (hostW, ACK3, TS3, MAC3)=m3 in
.
 let PSKW=getKey(hostW) in
 let Res3=verifyMAC((ACK3, TS3),MAC3,PSKW) in
 if Res3 = true then
 if ACK3 = Ack then
 if TS1<>TS3 then
.
 (* Msg 4:  The KM sends an acknowledgement to ECU1 *)
.
 new ts4;
 out(c, (hostY,Ack,ts4, MAC((Ack,ts4),PSKY)));
.
 (* Use of the SesK in ECUKM*)
.
 in(c, mx);
 let (=hostY,=GN1,TS5, MAC5)=mx in
 if verifyMAC((GN1,TS5),MAC5,SESK)=true then 0
.
```

```
 else callSWD.
.
 (* The process for ECUN *)
.
let processECUN =
.
 (* Msg 2:  The ECUN receives the message 2 *)
.
 in(c, m2);
 let (hostU, Encrypt2, GN2, TS2, MAC2)=m2 in
.
 if hostU=ecukm then
 let Res2=verifyMAC((Encrypt2, GN2, TS2),MAC2, pskn) in
 if Res2=true then
 if GN2=gn then
 let SESK1 = decrypt(Encrypt2, pskn) in
.
 (* Msg 3:  The ECUN sends an acknowledgement to ECUKM *)
.
 new ts3;
 out(c, (ecun, Ack, ts3, MAC((Ack,ts3),psn)));
.
 (* Use of the SesK ECUN*)
.
 in(c, mx1);
 let (hostV, =GN2, TS6, MAC6)=mx1 in
.
 if verifyMAC((GN2, TS6),MAC6,SESK1)=true then 0
 else callSWD.
.
let callSWD =.
.
 out(c1, Alert).
.
```

5. Main
   The last section of the model define specific relations between `ecu1` and `ecun`. Indeed, such ECU hosts preshare symmetric keys with the Key Master ECU, denoted by `ecukm`. Moreover, they belong to the same group which is denoted by `gn`. Thus, such group is composed by `ecu1`, `ecukm` and `ecun`. Note that the symbol `!` indicates infinite replication of processes whilst the symbol `|` indicates parallel processes execution. The lines that correspond to the *Main* section are as follows:.

```
 (* The process main *)
.
process
 new psk1; let ecu1=host(psk1) in
 new pskn; let ecun=host(pskn) in
.
```

```
 let gn=group((ecu1,ecukm,ecun)) in
.
 ((!processECU1) | (!processECUKM) | (!processECUN))
.
```

3.6.11 Results for Verification of Confidentiality (ProVerif)

The results are shown in Table 18.

Table 18 Results for verification of Confidentiality in the Keying Protocol

Verification Scheme	ProVerif Model and attacker queries
File model	pi_EVITA_KeyingProtocol_Confidentiality (ProVerif file)
Processes in the model	*processECU1, processECUKM, processECUN, callSWD*
Property representation	*query attacker:SesK; attacker:psk1; attacker:pskn.*
Secrecy assumptions	*not attacker:psk1. not attacker:pskn. not attacker:SesK.*
Nb. of phases	No phases were used
Nb. of rules for completion	200
Forcing completion	No tags were used
RESULT *not attacker:SesK*	True
RESULT *not attacker:pskn*	True
RESULT *not attacker:psk1*	True
Observations	The protocol preserves the secrecy of confidential data.

3.6.12 Model for Verification of Authenticity (ProVerif)

The ProVerif model for verification of Authenticity can be models as a set of the following sections of code:

Basic Blocks: This section includes the crypto primitives and related modeling functions. Additionally it can include the predicates, clauses and equations which conform the basic structures in the model.

Hypotheses: Includes the sentences for secrecy assumptions. These secrecy assumptions establish the information that is initially hidden for the attacker.

Queries: This section contains a list of queries that are addressed to verify the targeted properties.

Variables and Tags: This section includes the declaration of free variables and names that are used in the model. Additionally, it contains the declaration of tags that are used to forcing resolution algorithm termination.

Processes: Includes a list of communicating processes which determines the protocol definition. In almost all cases each process captures the semantics of a principal in the protocol.

Main: Since the Processes may be defined through parameters (free variables and names), this section is intended to provide specific values for such parameters. Moreover, this section determines an order for processes execution and the number of sessions for each process. Thus, the *Main* section implements a specific instance of the model.

1. Basic Blocks
   The *Basic Blocks* used in the model for verification of authenticity are presented below. Note that the function `getKey()` is a private block and can not be used by the attacker. As a consequence, the attacker can not derive a pre-shared key from the name of a host ECU.

```
 (* Each agent in the protocol has a group *)
.
data group/1.
data true/0.
.
reduc get1(group((x1,x2,x3)))=x1.
reduc get2(group((x1,x2,x3)))=x2.
reduc get3(group((x1,x2,x3)))=x3.
.
 (* Each agent shares a PSK with ECU-KM *)
.
fun host/1.
private reduc getKey(host(x))=x.
.
 (* To decrypt/encrypt messages with a key *)
.
fun encrypt/2.
reduc decrypt(encrypt(x, k),k)=x.
.
 (* MACs in messages *)
.
fun MAC/2.
fun verifyMAC/3.
.
equation verifyMAC(m, MAC(m, k), k)=true.
reduc getMsg(MAC(m,k))=m.
.
```

2. Hypotheses
   To be consistent, the hypotheses for confidential data should also be included for verification of authenticity. As it was mentioned, these assumptions restrict that confidential data are initially known by the attacker. Note that the section *Hypotheses* does not consider additional assumptions. Indeed, the hypotheses are written in a different manner but the semantics remains unchanged. The content of this section is in line.

```
 (* Initial secrecy assumptions ( the keys can not be derived from clauses
) *)
.
not attacker:psk1.
not attacker:pskn.
not attacker:SesK.
.
```

3. Queries
   The verification of authenticity uses the injective agreement approach that was described in a previous section (See 3.4.2). In compliance with security requirements, the *Queries* section includes one query for every process that needs authentication of an external non trusted process. Such query assumes that the communication between processes is performed through an insecure channel (`c`). Thus a query of the form

   `evinj:endComA(x)==>evinj:beginComA(x)`,

   models that whenever the event `endComA()` with parameter `x` happens in a process A, at least one event `beginComA()` with the same parameter has been executed in the process B. If such query is true, then the authenticity of process B is ensured, according to the process A perspective. Note that the events should be adequately located in the processes to be authenticated. The queries that were formulated for the verification of authenticity are presented just below. The three parameters that were used for `begin` events respectively correspond to the local host, the remote host and the SesK value. In the case of `end` events, the parameters respectively correspond to remote host, local host and SesK value.

```
 (* Queries for Verification of Authenticity *)
.
query evinj:endECU1_ECUKM(x1,x2,x3)==>evinj:beginECU1_ECUKM(x1,x2,x3).
query evinj:endECUKM_ECU1(x1,x2,x3)==>evinj:beginECUKM_ECU1(x1,x2,x3).
query evinj:endECUKM_ECUN(x1,x2,x3)==>evinj:beginECUKM_ECUN(x1,x2,x3).
query evinj:endECUN_ECUKM(x1,x2,x3)==>evinj:beginECUN_ECUKM(x1,x2,x3).
.
```

4. Variables and Tags
   Since in the verification of authenticity we need to force termination of the resolution algorithm, we include the section *Variables and Tags*. This section contains the definition of tags that are used for differentiation of instances of the same primitive. As it was mentioned (See 3.4.4), tags are a mean to avoid derivation of clauses that provoke loops. Such clauses come from unification of functions with the same kind of arguments. Note that the tags can be known by the attacker.

```
 (* Declaration of some free variables that the attacker may know *)
.
data ecukm/0.
```

```
data Alert/0.
data Ack/0.
data tg1/0.
data tg2/0.
data tg3/0.
data tg4/0.
data tg5/0.
data tg6/0.
data tg7/0.
```

.

5. processes
   In the section *Processes* we include the same processes that those already presented in the previous model for confidentiality (See 3.6.10). However, in this section we show how the tagging technique is applied. Indeed, whenever a crypto primitive is used, it includes an assigned `tag`. Consequently, the reception and verification of messages should be accordingly modified. We additionally introduce the notation of `phase n`; all the processes try to execute operations in a `phase i` before continuing with the execution in the `phase i+1`. Phases are indeed a mean to provide synchronization between processes execution. The content of this *Processes* section is shown just below. Note that `event beginX()` and `event endX()` are used for authenticity proof, along with the corresponding queries defining the injection.

```
 (* The process for ECU1 *)
.
let processECU1 =
.
 (* Msg 1:  Sends the new SesK to ECUKM for distribution among the members
of the group *)
.
 phase 1;
 new ts1;
 new SesK;
.
 out(c, (ecu1,encrypt((tg1,SesK),psk1), gn, ts1,
MAC((tg2,encrypt((tg1,SesK),psk1), gn, ts1),psk1)));
.
 (* Msg 4:  Receives an acknowledgement informing the SesK distribution *)
 phase 4;
 in(c, m4);
 let (hostX, ACK4, TS4, MAC4)=m4 in
 let Res4=verifyMAC((tg6, ACK4, TS4),MAC4, psk1) in
 if Res4=true then
.
 (* Use of the SesK in ECU1*)
.
 if ACK4=Ack then
 if TS4=plus(ts1,one) then
```

```
.
 event beginECU1_ECUKM(ecu1, hostX, SesK)
.
 event endECUKM_ECU1(hostX, ecu1, SesK);
.
 phase 5;
 new ts5;
 out(c, (ecu1,gn,ts5, MAC((tg7,gn,ts5),SesK)));
 out(c, (ecu1,gn,ts5, MAC((tg7,gn,ts5),SesK))).
.
 (* The process for ECUKM *)
.
let processECUKM =
.
 (* Msg 1:  The KM receives the distribution request from an ECU *)
 phase 1;
 in(c, m1);
 let (hostY, Encrypt1, GN1, TS1, MAC1)=m1 in
 let PSKY=getKey(hostY) in
 let Res1=verifyMAC((tg2,Encrypt1, GN1, TS1),MAC1, PSKY) in
 if Res1=true then
 if GN1=gn then
 let (=tg1,SESK)=decrypt(Encrypt1,PSKY) in
.
 event beginECUKM_ECU1(ecukm, hostY, SESK);
.
 (* Msg 2:  The KM distributes the SesK among the ECUs of the group *)
.
 phase 2;
 let hostZ=get3(gn) in
 if hostZ<>hostY then
 let PSKZ=getKey(hostZ) in
 new ts2;
 out(c, (ecukm,
encrypt((tg3,SESK),PSKZ),GN1,ts2,MAC((tg4,encrypt((tg3,SESK),PSKZ),GN1,ts2),
PSKZ)));
.
 event beginECUKM_ECUN(ecukm,PSKZ,SESK);
.
 (* Msg 3:  The KM receives an acknowledgement from ECUN *)
 phase 3;
 in(c, m3);
 let (hostW, ACK3, TS3, MAC3)=m3 in
.
 let PSKW=getKey(hostW) in
 let Res3=verifyMAC((tg5, ACK3, TS3),MAC3,PSKW) in
 if Res3=true then
 if ACK3 = Ack then
```

```
 if TS3<>TS1 then
.
 event endECUN_ECUKM(hostW,ecukm, SESK);
.
 (* Msg 4:  The KM sends an acknowledgement to ECU1 *)
 phase 4;
 new ts4;
.
 out(c, (hostY,Ack,plus(TS1,one), MAC((tg6,Ack,plus(TS1,one)),PSKY)));
.
 event endECU1_ECUKM(hostY, ecukm, SESK);
.
 (* Use of the SesK in ECUKM*)
 phase 5;
 in(c, mx);
 let (=hostY,=GN1,TS5, MAC5)=mx in
 if verifyMAC((tg7,GN1,TS5),MAC5,SESK)=true then 0
.
 else callSWD.
.
 (* The process for ECUN *)
.
let processECUN =
.
 (* Msg 2:  The ECUN receives the message 2 *)
 phase 2;
 in(c, m2);
 let (hostU, Encrypt2, GN2, TS2, MAC2)=m2 in
.
 if hostU=ecukm then
 let Res2=verifyMAC((tg4, Encrypt2, GN2, TS2),MAC2, pskn) in
 if Res2 = true then
 if GN2=gn then
.
 let (=tg3,SESK1) = decrypt(Encrypt2, pskn) in
.
 event beginECUN_ECUKM(ecun, hostU, SESK1);
.
 (* Msg 3:  The ECUN sends an acknowledgement to ECUKM *)
.
 phase 3;
 new ts3;
 out(c, (ecun, Ack, ts3, MAC((tg5,Ack,ts3),pskn)));
.
 event endECUKM_ECUN(hostU,ecun, SESK1);
.
 (* Use of the SesK ECUN*)
 phase 5;
```

```
 in(c, mx1);
 let (hostV, =GN2, TS6, MAC6)=mx1 in
.
 if verifyMAC((tg7,GN2,TS6),MAC6,SESK1)=true then 0
 else callSWD.
.
let callSWD =
.
 out(c1, Alert).
.
```

6. Main
   The section *Main* in the current model is a copy of the corresponding section in the model that was used for verification of confidentiality (See 3.6.10). However, in the current model the infinite replication of processes (`!`) is mandatory whenever two processes aim to authenticate. Indeed, the sets on which injective agreements are based upon can not be created without such replication. The lines for this section are presented below.

```
 (* The process main *)
.
process
 new psk1; let ecu1=host(psk1) in
 new pskn; let ecun=host(pskn) in
.
 let gn=group((ecu1,ecukm,ecun)) in
.
 ((!processECU1) | (!processECUKM) | (!processECUN))
.
```

3.6.13 Results for Verification of Authenticity (ProVerif)

The results are shown in Table 19.

3.6.14 Results for Verification of Confidentiality (AVATAR-Sec)

The AVATAR-Sec block Diagram of the keying protocol is provided in Figure 21. The AVATAR-Sec model for the Keying protocol is as follows:

- Only one ECU is the group is considered. That ECU is named ECUN on the model.
- The timer used by the Key Master is modeled with a non-deterministic expiration.
- Once the session key has been distributed to ECUN, ECU1 sens a confidential data to ECUN, using that session key.
- Pre-shared keys have been .... pre-shared. More particularly, *InitialCommonKnowledge* pragmas are used to say that ECU1 and KM have pre-shared a key:
  ♯InitialCommonKnowledge ECU1.PSK1 KM.PSK1
  ♯InitialCommonKnowledge ECUN.PSKN KM.PSKN

Table 19 Results for verification of Authenticity in the Keying Protocol

Verification Scheme	ProVerif Model and queries for injective agreements
File model	pi_EVITA_KeyingProtocol_Authenticity
Processes in the model	*processECU1, processECUKM, processECUN, callSWD*
Injective agreement ECU1 and ECUKM	$query\ evinj : endECU1\_ECUKM(x1,x2,x3) ==> evinj : beginECU1\_ECUKM(x1,x2,x3).$
Injective agreement ECUKM and ECU1	$query\ evinj : endECUKM\_ECU1(x1,x2,x3) ==> evinj : beginECUKM\_ECU1(x1,x2,x3).$
Injective agreement ECUKM and ECUN	$query\ evinj : endECUKM\_ECUN(x1,x2,x3) ==> evinj : beginECUKM\_ECUN(x1,x2,x3).$
Injective agreement ECUN and ECUKM	$query\ evinj : endECUN\_ECUKM(x1,x2,x3) ==> evinj : beginECUN\_ECUKM(x1,x2,x3).$
Secrecy assumptions	*not attacker:psk1. not attacker:pskn. not attacker:SesK.*
Nb. of phases	5 phases
Nb. of rules for completion	800
Forcing completion	Tags were used
RESULT agreement ECU1 and ECUKM	False. An attack trace is found.
RESULT agreement ECUKM and ECU1	False. An attack trace is found.
RESULT agreement ECUKM and ECUN	False. An attack trace is found.
RESULT agreement ECUN and ECUKM	False. An attack trace is found.
Observations	The model is not able to detect replayed messages. Consequently attack traces are found.

Two confidentiality properties are studied:

- The fact that the session key remains confidential:
  ♯ **Confidentiality ECU1.SesK**
- The fact that the data *confData* sent by ECU1 to ECUN, once the session key has been exchanged, is confidential:
  ♯ **Confidentiality ECU1.confData**

Finally, verification results are provided in Table 20.

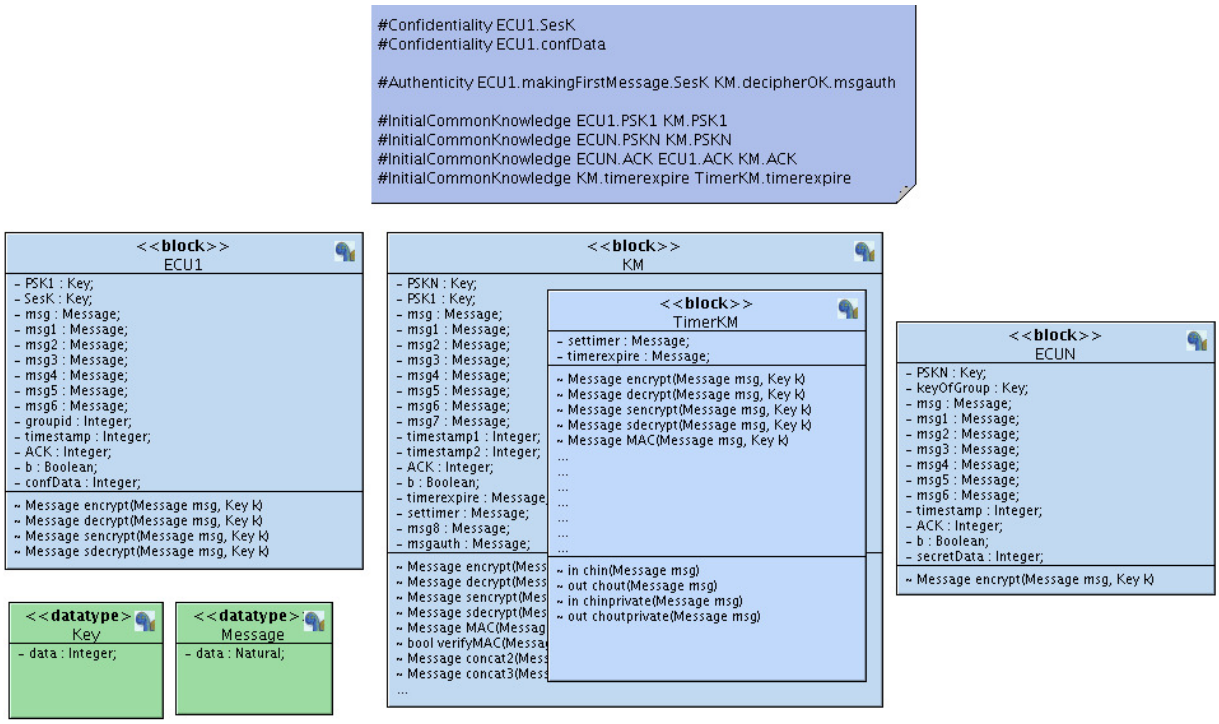


Figure 21 AVATAR Block Diagram for the Key Master protocol

Table 20 Results for verification of Confidentiality with AVATAR-Sec

Verification Scheme	AVATAR-Sec Model
File model	EVITA_AVATAR-Sec_KeyingProtocol.xml (TTool file)
Blocks in the model	*ECU1, ECUKM, ECUN, TimerKM*
Data structures of the model	*Key, Message*
InitialCommonKnowledge	♯InitialCommonKnowledge ECU1.PSK1 KM.PSK1, ♯InitialCommonKnowledge ECUN.PSKN KM.PSKN, ♯InitialCommonKnowledge ECUN.ACK ECU1.ACK KM.ACK, ♯ InitialCommonKnowledge KM.timerexpire TimerKM.timerexpire
Security properties	♯ Confidentiality ECU1.SesK, ♯ Confidentiality ECU1.confData
RESULT ♯ Confidentiality ECU1.SesK	True
RESULT ♯ Confidentiality ECU1.confData	True
States reachability	All states are reachable, apart from the *testMacFailed* state in KM.
Observations	The protocol preserves the secrecy of confidential data.

3.6.15 Results for Verification of Authenticity (AVATAR-Sec)

We have reused the AVATAR-Sec model used for proving confidentiality properties (see subsection 3.6.14). That model has been further enhanced with the following authenticity pragma (see Figure 21):
♯Authenticity ECU1.makingFirstMessage.SesK KM.decipherOK.msgauth
With that pragma, we intend to prove the authenticity of the first message sent by ECU1 to KM. The same approach could be used for other messages. Finally, verification results are provided in Table 21.

Table 21 Results for verification of authenticity with AVATAR-Sec

Verification Scheme	AVATAR-Sec Model
File model	EVITA_AVATAR-Sec_KeyingProtocol.xml (TTool file)
Blocks in the model	*ECU1, ECUKM, ECUN, TimerKM*
Data structures of the model	*Key, Message*
InitialCommonKnowledge	♯InitialCommonKnowledge ECU1.PSK1 KM.PSK1, ♯InitialCommonKnowledge ECUN.PSKN KM.PSKN, ♯InitialCommonKnowledge ECUN.ACK ECU1.ACK KM.ACK, ♯ InitialCommonKnowledge KM.timerexpire TimerKM.timerexpire
Security properties	♯Authenticity ECU1.makingFirstMessage.SesK KM.decipherOK.msgauth
RESULT ♯Authenticity ECU1.makingFirstMessage.SesK KM.decipherOK.msgauth	True
States reachability	All states are reachable, apart from the *testMacFailed* state in KM.
Observations	The protocol preserves the authenticity of the first message sent from ECU1 to KM.

3.7 Remote Flashing Update Protocol

3.7.1 Protocol Description

The Remote Flashing Update Protocol (Flashing Protocol) aims to securely update firmware in the flash memory of an in-car Electronic Control Unit (ECU). The Flashing Protocol involves four Communicating Entities (CE's) : The Diagnosis Tool (DT), The Communications Controller Unit (CCU), the target Electronic Control Unit (ECU) and the remote OEM server (OEM server). To ease the modeling description, the protocol is split into two phases. The first one, called *Diagnosis*, retrieves information related to the current ECU firmware and establishes a secure channel between the Diagnosis Tool and the target ECU. The second stage, called *Download*, securely links the target ECU and the OEM server in order to accomplish the firmware update transfer. Our protocol

description is based upon the extended version of this protocol in [18]. The typical trace of the protocol is presented in Figure 22.

Diagnosis Phase

The *Diagnosis* phase begins with an exchange between a Diagnosis Tool and the in-car CCU. The first message includes a connection request, a nonce $N1\_dt$ and a time stamp (Message 1). The information contained in this message is not confidential and thus, it is only signed with DT's private key (SK_dt). Since DT is an external entity for the in-car CCU, DT should provide its public key PK_dt within a certificate issued by an agreed Certification Authority (CA). After validation of DT credentials, the CCU delivers a connection response that includes the modified nonce $N1\_dt - 1$ and time stamp. In order to preserve the car privacy, CCU uses a previously assigned pair of short-term pseudo secret and public keys ($PsSK\_ccu$, $PsPK\_ccu$). Thus, the message is signed with $PsSK\_ccu$. In order to allow signature verification, CCU also provides the certificate that binds the respective pseudo public key (Message 2). This certificate is issued by the CA used in Message 1. Right after Message 2 validation, the DT will establish a secure channel with the target ECU. Indeed, DT randomly generates a symmetric session key $SesK$ and sets its flag to 'sign' (*use_flag=sign*, see EVITA D3.2 [19], Key Data Structures). Thus, the target ECU is allowed to MAC and verify messages. The generated $SesK$ is encrypted with the pseudo public key provided by CCU. Since PK_dt was provided to CCU, the message is signed with DT's secret key. A nonce $N2\_dt$ and time stamp are included (Message 3). After Message 3 reception, CCU tries to import the $SesK$ to its HSM. In case of success, the $SesK$ is sent to ECU. Indeed, $SesK$ is transferred using the target ECU's public key (PK_ecu) as a transport key (Message 4). The message is then completed with a nonce ($N1\_ccu$), a time stamp and is signed with the CCU's secret key (SK_ccu). After $SesK$ reception, ECU verifies Message 4 signature. Note that ECU already knows CCU's public key. Right after, ECU imports the encrypted $SesK$ into its HSM. In case of success, an acknowledgement code ACK is sent to CCU (Message 5). This acknowledgement message includes the modified nonce $N1\_ccu - 1$, a time stamp and is signed with ECU's private key (SK_ecu). After message validation, CCU sends to DT a successful establishment of authentic channel. The message includes an ACK code, the modified nonce $N2\_dt - 1$, a time stamp and is signed with CCU's pseudo private key (Message 6). Thus, DT and ECU are finally able to perform authentic bidirectional exchanges MACed with $SesK$.

Download Phase

The *Download* phase begins right after the *Diagnosis* phase has been accomplished. The Diagnosis Tool requests ECU's public key through the previously secured channel (Message 7). Indeed, the message includes a nonce ($N3\_dt$), a time stamp and is MAC protected with $SesK$. Note that DT – ECU communication is necessarily performed through CCU which plays the role of gateway. After message reception and in case of valid MAC, ECU encrypts its public key PK_ecu with the so called public Stakeholder Asymmetric Key (PK_sak). Such public key is factory installed in ECU to enforce confidentiality in exchanges with the OEM (see EVITA D3.2 [19], Internal Memory Data). The encrypted key is sent together with a modified nonce ($N3\_dt - 1$), a time stamp and

is MAC protected (Message 8). After reception, DT verifies message validity and afterwards composes a message requesting the firmware to the remote OEM server (Message 9). The message request includes the encrypted ECU's public key, a nonce $N4\_dt$, a time stamp and is signed with DT's secret key (SK_dt). Consequently, DT should share the certificate that binds its respective public key. Indeed, such certificate was issued by an agreed Certification Authority (CA). Once the OEM server receives Message 9 and right after DT credentials validation, the ECU's public key is decrypted and later on used to protect the firmware update. The OEM server composes a message that includes the encrypted firmware, the modified nonce $N4\_dt - 1$, a time stamp and a signature made with the OEM's secret key (SK_oem). Thus, the message should include the certificate that binds the OEM's public key (Message 10). Such certificate was issued by the same agreed CA as for certificate of Message 9. When DT receives the message, the OEM server credentials are validated. In case of successful validation, DT sends the encrypted firmware to ECU that was requested by the firmware update mode (Message 11). Apart from the encrypted firmware and the mode request, the message also includes a nonce $N5\_dt$, a time stamp and is MAC protected with $SesK$. The ECU receives the update, and after MAC validation, another sub-protocol for memory flashing is triggered. The flashing process updates the firmware in the ECU's non-volatile memory. Finally, if the flashing was successfully achieved, then the ECU sends an acknowledgement ACK to DT (Message 12). This message includes the modified nonce ($N5\_dt - 1$), a time stamp and is MAC protected with $SesK$.

3.7.2 Targeted Security Properties

The formal verification of Flashing Protocol targets the following security properties. These requirements are taken from [17].

1. *Authenticity_27* (see [17], page 30)
2. *Authenticity_28* (see [17], page 31)
3. *Authenticity_29* (see [17], page 31)
4. *Authenticity_101* (see [17], page 31)
5. *Authenticity_102* (see [17], page 32)
6. *Authenticity_103* (see [17], page 32)
7. *Confidentiality_1* (see [17], page 38)
8. *Privacy_101* (see [17], page 40)
9. *Confidentiality_101* (see [17], page 42)
10. *Confidentiality_102* (see [17], page 42)

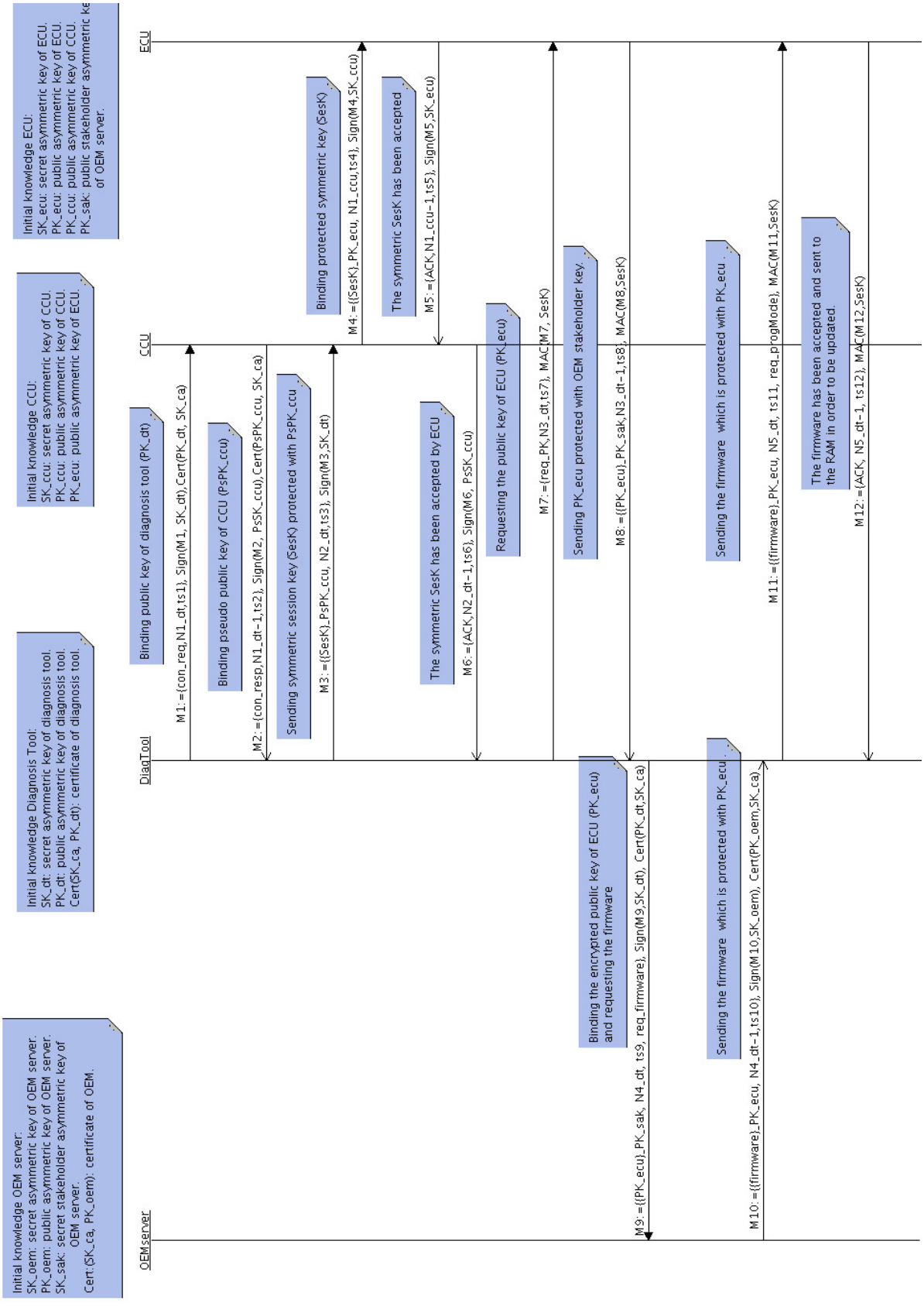


Figure 22 Sequence Diagram for the Flashing Protocol

3.7.3 Models for Verification of Confidentiality (ProVerif)

In order to ease modeling and verification, the Flashing Protocol is split according to *Diagnosis* and *Download* phases.

Diagnosis Phase

Three CE's are involved in the *Diagnosis* phase: DT, in-car CCU and target ECU. Consequently, the inclusion of OEM server in this phase is not necessary: it is therefore omitted. The sections composing the *Diagnosis* model for confidentiality are explained below. A detailed description of the ProVerif approach has been presented in section 3.4.1, thus detailed explanations are skipped.

1. Basic Blocks
   Since the Flashing Protocol relies on symmetric and asymmetric cryptography, the following *Basic Blocks* are included. Note that the attacker is not restricted from calling *Basic Blocks* i.e., the label `private` is not used.

```
 (* Create/verify MACs *)
.
fun MAC/2.
fun verifyMAC/3.
.
equation verifyMAC(m,MAC(m,k), k)= validMAC.
.
 (* Encryption/Decryption with asymmetric primitives *)
.
fun Pk/1.
.
fun encryptPK/2.
fun encryptSK/2.
.
reduc decryptSK(encryptPK(m,Pk(k)),k)=m.
reduc decryptPK(encryptSK(m,k),Pk(k))=m.
.
 (* Create/verify signatures *)
.
fun Hash/1.
fun verifySign/3.
.
equation verifySign(m,encryptSK(Hash(m),k),Pk(k))=validSign.
.
 (* Create/verify certificates *)
.
fun Cert/2.
fun verifyCert/3.
.
reduc getID(Cert(id,sign))=id.
reduc getSign(Cert(id,sign))=sign.
```

```
.
 (* Nonces modification *)
.
fun minus/2.
```

.

2. Hypotheses
   The *Hypotheses* of the model restrict secret keys for the initial attacker knowledge. Additionally, the session key generated by DT ($SesK$), as well as the target ECU's public key are assumed to be unknown for the attacker. Moreover, since car's identity should be kept confidential for external entities, *ccu_id* is assumed to be *secret*.

```
 (* Initial secrecy assumptions; the secret keys can not be derived from
clauses *)
.
not attacker:sk_ca.
not attacker:sk_dt.
not attacker:sk_ccu.
not attacker:sk_ecu.
not attacker:PsSK_ccu.
not attacker:SesK.
not attacker:pk_ecu.
not attacker:ccu_id.
```

.

3. Queries
   In agreement with secrecy assumptions, confidentiality *Queries* verify that material initially assumed secret is never accessible to the attacker.

```
 (* Queries for Verification of Secrecy/Confidentiality *)
.
query attacker:sk_ca.
query attacker:sk_dt.
query attacker:sk_ccu.
query attacker:sk_ecu.
query attacker:PsSK_ccu.
query attacker:SesK.
query attacker:pk_ecu.
query attacker:ccu_id.
```

.

4. Variables
   The *Variables* section includes the (non secure) channel declaration `c`. In fact, since the attacker listens to all non private channels, declaration of different channels only increases model complexity. Consequently, we assume that all the CE's interact through the same channel `c`. Several values – like commands and acknowledgements – in *Diagnosis* exchanges are parameterized: consequently data structures are defined accordingly. In agreement with *Hypotheses*, private declarations for ECU's public key (pk_ecu) and DT's session key ($SesK$) are included. At last, the declaration of tags is made.

```
 (* The channels in the system *)
free c.
.
 (* Some data values may be known by the attacker *)
data Ack/0.
data con_req/0.
data con_resp/0.
data req_PK/0.
data req_firmware/0.
data req_progMode/0.
data dt_id/0.
data ps_ccu_id/0.
data ecu_id/0.
data oem_id/0.
.
private free pk_ecu.
private free SesK.
.
 (* Some values for return *)
data validMAC/0.
data validSign/0.
data validCert/0.
data one/0.
.
 (* Tags for crypto functions *)
data tg1/0.
data tg2/0.
data tg3/0.
data tg4/0.
data tg5/0.
data tg6/0.
data tg7/0.
data tg8/0.
data tg9/0.
data tg10/0.
.
```

5. Processes
   Every CE is represented in the model as a pi-process. Indeed, the *Processes* section includes three processes, namely `processDT`, `processCCU` and `processECU`. Since processes behavior should model only the *Diagnosis* phase, only exchanges from message 1 to 6 are modeled. Also, the label `(* Msg i*)` is used to indicate message exchange. Note that cryptographic primitive blocks are tagged. Indeed, tagging of Flashing Protocol is mandatory in order to ensure termination of the verification algorithm.

```
 (* The process for the Diagnosis Tool *) .
let processDT =
```

```
 new ts1;
 new N1_dt;
 out(c, (dt_id,con_req,N1_dt,ts1,
encryptSK(Hash((tg1,con_req,N1_dt,ts1)),sk_dt),
Cert(pk_dt,encryptSK(Hash((tg2,pk_dt)),sk_ca)))); (* Msg 1 *)
.
 in(c, m2); (* Msg 2 *)
 let (hostX, Resp2,N2,TS2, Sign2,Cert2)=m2 in
.
 if hostX=ps_ccu_id then
 let PSPK_ccu = getID(Cert2) in
 let signCert2 = getSign(Cert2) in
 let valid2 = verifySign((tg4,PSPK_ccu),signCert2, pk_ca) in
 if valid2 = validSign then
 let Res1=verifySign((tg3,Resp2,N2,TS2),Sign2, PSPK_ccu) in
 if Res1=validSign then
.
 if N2=minus(N1_dt,one) then
 if Resp2=con_resp then
.
 new N2_dt;
 new ts3;
 out (c, (dt_id,encryptPK((tg5,SesK),
PSPK_ccu),N2_dt,ts3,encryptSK(Hash((tg6,encryptPK((tg5,SesK),
PSPK_ccu),N2_dt,ts3)),sk_dt))); (* Msg 3 *)
.
 in (c, m6); (* Msg 6 *)
 let (hostY,ACK6,N6,TS6, Sign6)=m6 in
 if hostY=hostX then
 let Res2=verifySign((tg10,ACK6,N6,TS6),Sign6,PSPK_ccu) in
 if Res2=validSign then
.
 if ACK6=Ack then
 if N6=minus(N2_dt,one) then 0.
.
 (* The process for the Communications Control Unit ECU *) .
let processCCU =
 new ccu_id;
 in (c, m1); (* Msg 1 *)
 let (hostP,CON_REQ, N1, TS1, Sign1, Cert1)=m1 in
 let signCert1= getSign(Cert1) in
 let PK_DT = getID(Cert1) in
 let valid1 = verifySign((tg2,PK_DT), signCert1, pk_ca) in
 if valid1 = validSign then
.
 let Res6 = verifySign((tg1,CON_REQ, N1, TS1), Sign1, PK_DT) in
 if Res6 = validSign then
.
```

```
 if CON_REQ=con_req then
.
 new ts2;
 new PsSK_ccu;
 let PsPK_ccu = Pk(PsSK_ccu) in
 out (c, (ps_ccu_id, con_resp, minus(N1,one),
ts2,encryptSK(Hash((tg3,con_resp, minus(N1,one),
ts2)),PsSK_ccu),Cert(PsPK_ccu,encryptSK(Hash((tg4,PsPK_ccu)),sk_ca)))); (*
Msg 2 *)
.
 in (c, m3); (* Msg 3 *)
 let (hostQ,Encrypt3, N3, TS3, Sign3)=m3 in
 if hostQ=hostP then
 let Res7=verifySign((tg6,Encrypt3, N3, TS3),Sign3,PK_DT) in
 if Res7= validSign then
 let (=tg5,SESK1) = decryptSK(Encrypt3,PsSK_ccu) in
.
 if SESK1=SesK then
.
 new N1_ccu;
 new ts4;
 out (c, (ps_ccu_id,encryptPK((tg7,SESK1),pk_ecu),N1_-
ccu,ts4,encryptSK(Hash((tg8,encryptPK((tg7,SESK1),pk_ecu),N1_-
ccu,ts4)),sk_ccu))); (* Msg 4
*)
.
 in (c, m5); (* Msg 5 *)
 let (hostR, ACK5, N5, TS5, Sign5) = m5 in
 let Res8 = verifySign((tg9,ACK5, N5, TS5), Sign5, pk_ecu) in
 if Res8 = validSign then
.
 if ACK5 = Ack then
 if N5 = minus(N1_ccu,one) then
.
 new ts6;
 out (c, (ps_ccu_id, Ack, minus(N3,one), ts6, encryptSK(Hash((tg10,Ack,
minus(N3,one), ts6)),PsSK_ccu))).  (* Msg 6 *)
.
 (* The process for the target ECU *)
.
let processECU =
 in (c, m4); (* Msg 4 *)
 let (hostS, Encrypt4, N4, TS4, Sign4)=m4 in
.
 if hostS=ps_ccu_id then
 let Res9 = verifySign((tg8,Encrypt4, N4, TS4),Sign4, pk_ccu) in
 if Res9= validSign then
.
```

```
 let (=tg7,SESK2) = decryptSK(Encrypt4, sk_ecu) in
 if SESK2 = SesK then
 new ts5;
 out (c, (ecu_id, Ack, minus(N4,one), ts5, encryptSK(Hash((tg9,Ack,
minus(N4,one), ts5)),sk_ecu))).  (* Msg 5 *)
```

.

6. Main
   The *Main* process is used to declare secret keys. Hence, respective public keys are associated using the basic block `Pk()`. Afterwards, public keys are broadcasted in the communications channel `c`. Note that, in agreement with secrecy assumptions, the public key *pk_ecu* is not broadcasted. Indeed, our model is compliant with the fact that in-car public keys are restricted to the in-car domain (privacy). Last but not least, the involved processes are executed in parallel, and an infinite number of times.

```
 (* Main process *)
 process
.
 new sk_dt;
 let pk_dt = Pk(sk_dt) in
 out(c, pk_dt);
 new sk_ccu;
 let pk_ccu = Pk(sk_ccu) in
 out (c, pk_ccu); (* The pk_ccu is internally distributed in the car *)
 new sk_ecu;
 let pk_ecu = Pk(sk_ecu) in (* The pk_ecu is internally distributed in the
car *)
 new sk_ca;
 let pk_ca = Pk(sk_ca) in
 out (c, pk_ca);
.
 ((!processDT)|(!processCCU)|(!processECU))
```

.

Download Phase

Since *Download* is in fact a continuation of the *Diagnosis* phase, it is assumed that message exchanges of the previous phase have been successfully executed. Consequently, respective data are in CE's initial knowledge. Moreover, we assume that protocol exchanges preserve secrecy of confidential material. It implies that the attacker was unable to perform any attack in the previous phase. As a particular case, it is assumed that the session key generated by DT ($SesK$) has been successfully transferred to the target ECU.

The *Download* phase is modeled in Proverif as follows (see section 3.4.1 for more details on the modeling approach).

1. Basic Blocks
   The *Basic Blocks* for modeling *Download* phase are the same as in *Diagnosis* phase. Since no additional blocks were defined, this section is not further described. To preserve model correctness, the *Hypotheses* section consider the same hypotheses as

in *Diagnosis* phase. However, new elements are assumed as secret. Beside new keys, the `ecu_id`, representing ECU identification, and `firmware1`, representing firmware update, are included as secrecy assumptions. The contents of *Hypotheses* section are below.

```
 (* Initial secrecy assumptions; the secret keys can not be derived from
clauses *)
not attacker:sk_ca.
not attacker:sk_dt.
not attacker:sk_ecu.
not attacker:sk_oem.
not attacker:sk_sak.
not attacker:SesK.
not attacker:pk_ecu.
not attacker:ecu_id.
not attacker:firmware1.
```

.

2. Queries
   The *Queries* section is in correspondence with previously stated secrecy assumptions. In fact, it is formally verified that secret material is not delivered to the attacker as a result of protocol exchanges. In line the respective queries.

```
 (* Queries for Verification of Secrecy/Confidentiality *)
.
query attacker:sk_ca.
query attacker:sk_dt.
query attacker:sk_ecu.
query attacker:sk_oem.
query attacker:sk_sak.
query attacker:SesK.
query attacker:pk_ecu.
query attacker:ecu_id.
query attacker:firmware1.
```

.

3. Variables
   The *Variables* section includes declarations for the insecure channel `c`, parameterized values, CE's identifications and tags. The contents of this section are below.

```
 (* The channels in the system *)
free c.
.
 (* Some data values may be known by the attacker *)
data Ack/0.
data con_req/0.
data con_resp/0.
data req_PK/0.
data req_firmware/0.
```

```
data req_progMode/0.
data dt_id/0.
data ps_ecu_id/0.
data oem_id/0.
data N5_dt/0.
data ts11/0.
data ts12/0.
.
private free pk_ecu.
.
 (* Some values for return *)
data validMAC/0.
data validSign/0.
data validCert/0.
data one/0.
.
 (* Tags for crypto functions *)
data tg11/0.
data tg12/0.
data tg13/0.
data tg14/0.
data tg15/0.
data tg16/0.
data tg17/0.
data tg18/0.
data tg19/0.
data tg20/0.
.
```

4. Processes
   Download phase involves three CE's which are associated to respective processes, namely `processDT`, `processECU` and `processOEMserver`. Since this *Processes* section is the continuation of *Diagnosis* phase, messages 7 to 12 are included. Indeed, we put the label `(* Msg i*)` to indicate respective message exchange.

```
 (* The process for the Diagnosis Tool *)
.
let processDT =
 new N3_dt;
 new ts7;
.
 out (c, (dt_id, req_PK,N3_dt,ts7,MAC((tg11,req_PK,N3_dt,ts7),SesK))); (*
Msg 7 *)
.
 in(c, m8); (* Msg 8 *)
 let (hostZ,Encrypt8,N8,TS8,MAC8)=m8 in
 let Res3=verifyMAC((tg13,Encrypt8,N8,TS8),MAC8,SesK) in
 if Res3=validMAC then
```

```
.
 if N8=minus(N3_dt,one) then
.
 new N4_dt;
 new ts9;
 out(c, (dt_id, Encrypt8, N4_dt, ts9,req_firmware,
encryptSK(Hash((tg14,Encrypt8, N4_dt,
ts9,req_firmware)),sk_dt),Cert(pk_dt,
encryptSK(Hash((tg15,pk_dt)),sk_ca)))); (* Msg 9 *)
.
 in (c, m10); (* Msg 10 *)
 let (hostU, EncryptFirm, N10, TS10, Sign10, Cert10)=m10 in
 let PK_oem = getID (Cert10) in
 let signCert10 = getSign(Cert10) in
.
 let valid10 =verifySign((tg18,PK_oem),signCert10,pk_ca) in
 if valid10 = validSign then
.
 let Res4 = verifySign((tg17,EncryptFirm, N10, TS10),Sign10,PK_oem) in
 if Res4=validSign then
.
 if N10=minus(N4_dt,one) then
.
 out (c, (dt_id,req_progMode,EncryptFirm,N3_dt,ts11,MAC((tg19,req_-
progMode,EncryptFirm,N3_dt,ts11),SesK))); (* Msg 11
*)
.
 in (c, m12); (* Msg 12 *)
 let (=hostZ, ACK12, N12,TS12,MAC12)=m12 in
 let Res5 = verifyMAC((tg20,ACK12,N12,TS12),MAC12,SesK) in
 if Res5 = validMAC then
 if ACK12 = Ack then
 if N12=minus(N3_dt,one) then 0.
.
 (* The process for the target ECU *)
.
let processECU =
 new ecu_id;
 in (c, m7); (* Msg 7 *)
 let (hostT, REQ_PK, N7,TS7, MAC7)=m7 in
 let Res10=verifyMAC((tg11,REQ_PK, N7,TS7),MAC7,SesK) in
 if Res10 = validMAC then
 if REQ_PK = req_PK then
.
 new ts8;
 out (c, (ps_ecu_id, encryptPK((tg12,pk_ecu),pk_-
sak),minus(N7,one),ts8,MAC((tg13,encryptPK((tg12,pk_ecu),pk_-
```

```
sak),minus(N7,one),ts8),SesK))); (* Msg 8
*)
.
 in (c, m11); (* Msg 11 *)
 let (=hostT,REQ_PROG,Encrypt11,N11,TS11,MAC11)=m11 in
 let Res11 = verifyMAC((tg19,REQ_PROG,Encrypt11,N11,TS11),MAC11, SesK) in
 if Res11 = validMAC then
 if REQ_PROG=req_progMode then
 let (=tg16, Firmware) = decryptSK(Encrypt11, sk_ecu) in
.
 out (c, (ps_ecu_-
id,Ack,minus(N11,one),ts12,MAC((tg20,Ack,minus(N11,one),ts12),SesK))). (*
Msg 12 *)
.
 (* The process for the OEM server *)
.
let processOEMserver =
 in (c, m9); (* Msg 9 *)
 let (hostL, Encrypt9, N9, TS9, REQ_FIRM, Sign9, Cert9)=m9 in
 let PK_DT = getID(Cert9) in
 let signCert9 = getSign(Cert9) in
 let valid9 = verifySign((tg15,PK_DT),signCert9, pk_ca) in
 if valid9 = validSign then
.
 let Res12 = verifySign((tg14,Encrypt9, N9, TS9, REQ_FIRM),Sign9,PK_DT) in
 if Res12 = validSign then
.
 if REQ_FIRM = req_firmware then
 let (=tg12,Pk_ECU) = decryptSK(Encrypt9, sk_sak) in
.
 new ts10;
 new firmware1;
.
 out (c, (oem_id, encryptPK((tg16,firmware1),Pk_-
ECU),minus(N9,one),ts10,encryptSK(Hash((tg17,encryptPK((tg16,firmware1),Pk_-
ECU),minus(N9,one),ts10)),sk_oem),Cert(pk_oem,encryptSK(Hash((tg18,pk_-
oem)),sk_ca)))).  (* Msg 10
*)
.
```

5. Main
   The *Main* process contains declarations of secret keys. The association between secret and public keys is made with the basic block `Pk()`. Later on, public keys are broadcasted in `c`. Note that ECU's public key is not broadcasted. Since we assume that session key $SesK$ is known by `processDT` and `processECU`, it is declared within *Main* process. The replication of processes ensures that secrecy is preserved no matter the number of sessions.

```
 (* Main process *)
```

```
.
process
 new sk_dt;
 let pk_dt = Pk(sk_dt) in
 out(c, pk_dt);
.
 new sk_ecu;
 let pk_ecu = Pk(sk_ecu) in
.
 new sk_oem;
 let pk_oem = Pk(sk_oem) in
 out(c, pk_oem);
.
 new sk_ca;
 let pk_ca = Pk(sk_ca) in
 out (c, pk_ca);
.
 new sk_sak;
 let pk_sak = Pk(sk_sak) in
 out (c, pk_sak);
.
 new SesK;
.
 ((!processDT)|(!processECU)|(!processOEMserver))
.
```

3.7.4 Results for Verification of Confidentiality (ProVerif)

The results for the *Diagnosis* phase are shown in Table 22 whilst those corresponding to the *Download* phase are presented in Table 23.

3.7.5 Models for Verification of Authenticity (ProVerif)

As for confidentiality, the model for authenticity verification of the the Flashing Protocol is split into two phases: *Diagnosis* and *Download*. We assume that proving authenticity independently for the two phases is equivalent to proving the authenticity for the two joined phases.

Diagnosis Phase

The model for verification of authenticity in the *Diagnosis* phase contains the following sections. Since their contents have been presented in section 3.4, no additional comments are provided.

In fact, the sections *Basic Blocks*, *Hypotheses*, *Variables* and *Main* are the same as in the *Diagnosis* phase model for confidentiality (See 3.7.3). Consequently, we leave out additional comments. Thus we only present respective *Queries* and *Processes*

1. Queries
   The queries for verification of authenticity recall the principle of injective agreement already explained in 3.4.2. Indeed, two information flows are present; between DT

Table 22 Results for verification of Confidentiality in *Diagnosis* phase

Verification Scheme	ProVerif Model and attacker queries
File model	pi_EVITA_FlashingDiagnosis_Confidentiality_1
Processes in the model	*processDT, processCCU, processECU*
Property representation	*query attacker:sk_ca; attacker:sk_dt; attacker:sk_ccu; attacker:sk_ecu; attacker:PsSK_ccu; attacker:SesK; attacker:pk_ecu; attacker:ccu_id.*
Secrecy assumptions	*not attacker:sk_ca. not attacker:sk_dt. not attacker:sk_ccu. not attacker:sk_ecu. not attacker:PsSK_ccu. not attacker:SesK. not attacker:pk_ecu. not attacker:ccu_id.*
Nb. of phases	No phases were used
Nb. of rules for completion	200
Forcing completion	Tags were used
RESULT *not attacker:sk_ca*	True
RESULT *not attacker:sk_dt*	True
RESULT *not attacker:sk_ccu*	True
RESULT *not attacker:sk_ecu*	True
RESULT *not attacker:PsSK_ccu*	True
RESULT *not attacker:SesK*	True
RESULT *not attacker:pk_ecu*	True
RESULT *not attacker:ccu_id*	True
Observations	The phase *Diagnosis* preserves the secrecy of confidential data.

and CCU and between CCU and ECU. Consequently, the authentication of involved CE's is made according to the following queries:

```
 (* Queries for verification of Authenticity *)
query evinj:endComDT_CCU(x1,x2) ==> evinj:beginComDT_CCU(x1,x2).
query evinj:endComCCU_DT(x1,x2) ==> evinj:beginComCCU_DT(x1,x2).
query evinj:endComCCU_ECU(x1,x2) ==> evinj:beginComCCU_ECU(x1,x2).
query evinj:endComECU_CCU(x1,x2) ==> evinj:beginComECU_CCU(x1,x2).
```

.

Thus for instance, the authenticity of CCU from DT's perspective is verified in the first query. The arguments `xi` in `endCom` events are local host and remote host identifications, respectively. Thus, the arguments `xi` in `beginCom` events correspond to remote and local hosts, respectively.

2. Processes
   This phase is composed of three processes, namely `processDT`, `processCCU` and `processECU`. Every process includes the events referenced in respective queries. Note that apart from `events`, this *Processes* section is the same as the respective one in

Table 23 Results for verification of Confidentiality in *Download* phase

Verification Scheme	ProVerif Model and attacker queries
File model	pi_EVITA_FlashingDownload_Confidentiality_1
Processes in the model	*processDT, processECU, processOEMserver*
Property representation	*query attacker:sk_ca; attacker:sk_dt; attacker:sk_ecu; attacker;sk_oem; attacker:sk_sak; attacker:SesK; attacker:pk_ecu; attacker:ecu_id; attacker:firmware1.*
Secrecy assumptions	*not attacker:sk_ca. not attacker:sk_dt. not attacker:sk_ecu. not attacker:sk_oem. not attacker:sk_sak. not attacker:SesK. not attacker:pk_ecu. not attacker:ecu_id. not attacker:firmware1.*
Nb. of phases	No phases were used
Nb. of rules for completion	200
Forcing completion	Tags were used
RESULT *not attacker:sk_ca*	True
RESULT *not attacker:sk_dt*	True
RESULT *not attacker:sk_ecu*	True
RESULT *not attacker:sk_oem*	True
RESULT *not attacker:sk_sak*	True
RESULT *not attacker:SesK*	True
RESULT *not attacker:pk_ecu*	True
RESULT *not attacker:ecu_id*	True
RESULT *not attacker:firmware1*	True
Observations	The *Download* phase preserves secrecy of confidential data.

verification of confidentiality (see 3.7.3, *Diagnosis* phase). Indeed, we recall that exchanges in this phase correspond to messages 1-6 in the protocol description.

```
 (* The process for the Diagnosis Tool *)
.
let processDT =
 new ts1;
 new N1_dt;
 out(c, (dt_id,con_req,N1_dt,ts1,
encryptSK(Hash((tg1,con_req,N1_dt,ts1)),sk_dt),
Cert(pk_dt,encryptSK(Hash((tg2,pk_dt)),sk_ca)))); (* Msg 1 *)
.
 in(c, m2); (* Msg 2 *)
 let (hostX, Resp2,N2,TS2, Sign2,Cert2)=m2 in
.
 if hostX=ps_ccu_id then
 let PSPK_ccu = getID(Cert2) in
```

```
 let signCert2 = getSign(Cert2) in
 let valid2 = verifySign((tg4,PSPK_ccu),signCert2, pk_ca) in
 if valid2 = validSign then
 let Res1=verifySign((tg3,Resp2,N2,TS2),Sign2, PSPK_ccu) in
 if Res1=validSign then
.
 event beginComCCU_DT(hostX,dt_id);
.
 if N2=minus(N1_dt,one) then
 if Resp2=con_resp then
.
 new N2_dt;
 new ts3;
 out (c, (dt_id,encryptPK((tg5,SesK),
PSPK_ccu),N2_dt,ts3,encryptSK(Hash((tg6,encryptPK((tg5,SesK),
PSPK_ccu),N2_dt,ts3)),sk_dt))); (* Msg 3 *)
.
 in (c, m6); (* Msg 6 *)
 let (hostY,ACK6,N6,TS6, Sign6)=m6 in
 if hostY=hostX then
 let Res2=verifySign((tg10,ACK6,N6,TS6),Sign6,PSPK_ccu) in
 if Res2=validSign then
.
 event endComDT_CCU(dt_id,hostY);
.
 if ACK6=Ack then
 if N6=minus(N2_dt,one) then 0.
.
 (* The process for the Communications Control Unit ECU *)
.
let processCCU =
 new ccu_id;
 in (c, m1); (* Msg 1 *)
 let (hostP,CON_REQ, N1, TS1, Sign1, Cert1)=m1 in
 let signCert1= getSign(Cert1) in
 let PK_DT = getID(Cert1) in
 let valid1 = verifySign((tg2,PK_DT), signCert1, pk_ca) in
 if valid1 = validSign then
.
 let Res6 = verifySign((tg1,CON_REQ, N1, TS1), Sign1, PK_DT) in
 if Res6 = validSign then
.
 event beginComDT_CCU(hostP,ps_ccu_id);
.
 if CON_REQ=con_req then
.
 new ts2;
 new PsSK_ccu;
```

```
 let PsPK_ccu = Pk(PsSK_ccu) in
 out (c, (ps_ccu_id, con_resp, minus(N1,one),
ts2,encryptSK(Hash((tg3,con_resp, minus(N1,one),
ts2)),PsSK_ccu),Cert(PsPK_ccu,encryptSK(Hash((tg4,PsPK_ccu)),sk_ca)))); (*
Msg 2 *)
.
 in (c, m3); (* Msg 3 *)
 let (hostQ,Encrypt3, N3, TS3, Sign3)=m3 in
 if hostQ=hostP then
 let Res7=verifySign((tg6,Encrypt3, N3, TS3),Sign3,PK_DT) in
 if Res7= validSign then
 let (=tg5,SESK1) = decryptSK(Encrypt3,PsSK_ccu) in
.
 if SESK1=SesK then
.
 new N1_ccu;
 new ts4;
 out (c, (ps_ccu_id,encryptPK((tg7,SESK1),pk_ecu),N1_-
ccu,ts4,encryptSK(Hash((tg8,encryptPK((tg7,SESK1),pk_ecu),N1_-
ccu,ts4)),sk_ccu))); (* Msg 4
*)
.
 in (c, m5); (* Msg 5 *)
 let (hostR, ACK5, N5, TS5, Sign5) = m5 in
 let Res8 = verifySign((tg9,ACK5, N5, TS5), Sign5, pk_ecu) in
 if Res8 = validSign then
.
 event beginComECU_CCU(hostR, ps_ccu_id);
.
 if ACK5 = Ack then
 if N5 = minus(N1_ccu,one) then
.
 event endComCCU_ECU(ps_ccu_id, hostR);
.
 new ts6;
 out (c, (ps_ccu_id, Ack, minus(N3,one), ts6, encryptSK(Hash((tg10,Ack,
minus(N3,one), ts6)),PsSK_ccu))); (* Msg 6 *)
.
 event endComCCU_DT(ps_ccu_id, hostP).
.
 (* The process for the target ECU *)
.
let processECU =
 in (c, m4); (* Msg 4 *)
 let (hostS, Encrypt4, N4, TS4, Sign4)=m4 in
.
 if hostS=ps_ccu_id then
 let Res9 = verifySign((tg8,Encrypt4, N4, TS4),Sign4, pk_ccu) in
```

```
 if Res9= validSign then
.
 event beginComCCU_ECU(hostS, ecu_id);
.
 let (=tg7,SESK2) = decryptSK(Encrypt4, sk_ecu) in
 if SESK2 = SesK then
 new ts5;
 out (c, (ecu_id, Ack, minus(N4,one), ts5, encryptSK(Hash((tg9,Ack,
minus(N4,one), ts5)),sk_ecu))); (* Msg 5 *)
.
 event endComECU_CCU(ecu_id,hostS).
```

.

Download Phase

The model for verification of authenticity in the *Download* phase is composed of the following sections. The contents are already explained in 3.4, thus we omit additional comments.

As in the previous phase, the sections *Basic Blocks*, *Hypotheses*, *Variables* and *Main* are the same as in the *Download* phase model for confidentiality (See 3.7.3, *Download* phase). Consequently, we only describe *Queries* and *Processes* sections.

1. Queries
   Two information flows between CE's are identified in *Download* phase. The first one between DT and target ECU. The second one corresponds to remote communication between DT and OEM server. The authentication of involved EC's is made according to the following *Queries*:

```
 (* Queries for verification of Authenticity *)
query evinj:endComDT_ECU(x1,x2) ==> evinj:beginComDT_ECU(x1,x2).
query evinj:endComECU_DT(x1,x2) ==> evinj:beginComECU_DT(x1,x2).
query evinj:endComDT_OEM(x1,x2) ==> evinj:beginComDT_OEM(x1,x2).
query evinj:endComOEM_DT(x1,x2) ==> evinj:beginComOEM_DT(x1,x2).
```

.

   For example, the last query verifies the authenticity of DT from the OEM server's perspective. The arguments in `endCom` events respectively correspond to local and remote hosts ID's. Hence, the arguments for `beginCom` events correspond to remote and local hosts ID's.

2. Processes
   The *Processes* section includes three processes, namely `processDT`, `processECU` and `processOEMserver`. The events referred in *Queries* are accordingly included within respective processes. We recall that, *Download* phase covers from message 7 to 12 in the protocol specification. Such exchanges are labeled with `(* Msg i *)`.

```
 (* The process for the Diagnosis Tool *)
.
```

```
let processDT =
 new N3_dt;
 new ts7;
.
 out (c, (dt_id, req_PK,N3_dt,ts7,MAC((tg11,req_PK,N3_dt,ts7),SesK))); (*
Msg 7 *)
.
 in(c, m8); (* Msg 8 *)
 let (hostZ,Encrypt8,N8,TS8,MAC8)=m8 in
 let Res3=verifyMAC((tg13,Encrypt8,N8,TS8),MAC8,SesK) in
 if Res3=validMAC then
.
 event beginComECU_DT(hostZ, dt_id);
.
 if N8=minus(N3_dt,one) then
 new N4_dt;
 new ts9;
 out(c, (dt_id, Encrypt8, N4_dt, ts9,req_firmware,
encryptSK(Hash((tg14,Encrypt8, N4_dt,
ts9,req_firmware)),sk_dt),Cert(pk_dt,
encryptSK(Hash((tg15,pk_dt)),sk_ca)))); (* Msg 9 *)
.
 in (c, m10); (* Msg 10 *)
 let (hostU, EncryptFirm, N10, TS10, Sign10, Cert10)=m10 in
 let PK_oem = getID (Cert10) in
 let signCert10 = getSign(Cert10) in
.
 let valid10 =verifySign((tg18,PK_oem),signCert10,pk_ca) in
 if valid10 = validSign then
 let Res4 = verifySign((tg17,EncryptFirm, N10, TS10),Sign10,PK_oem) in
 if Res4=validSign then
.
 event beginComOEM_DT(hostU,dt_id);
.
 if N10=minus(N4_dt,one) then
.
 event endComDT_OEM(dt_id,hostU);
.
 out (c, (dt_id,req_progMode,EncryptFirm,N3_dt,ts11,MAC((tg19,req_-
progMode,EncryptFirm,N3_dt,ts11),SesK))); (* Msg 11
*)
.
 in (c, m12);
 let (=hostZ, ACK12, N12,TS12,MAC12)=m12 in
 let Res5 = verifyMAC((tg20,ACK12,N12,TS12),MAC12,SesK) in
 if Res5 = validMAC then
 if ACK12 = Ack then
 if N12=minus(N3_dt,one) then
```

.
event endComDT_ECU(dt_id,hostZ).
.
(* The process for the target ECU *)
.
```
let processECU =
 in (c, m7); (* Msg 7 *)
 let (hostT, REQ_PK, N7,TS7, MAC7)=m7 in
 let Res10=verifyMAC((tg11,REQ_PK, N7,TS7),MAC7,SesK) in
 if Res10 = validMAC then
 if REQ_PK = req_PK then
```
.
event beginComDT_ECU(hostT,ecu_id);
.
```
 new ts8;
 out (c, (ecu_id, encryptPK((tg12,pk_ecu),pk_-
sak),minus(N7,one),ts8,MAC((tg13,encryptPK((tg12,pk_ecu),pk_-
sak),minus(N7,one),ts8),SesK))); (* Msg 8
*)
```
.
```
 in (c, m11); (* Msg 11 *)
 let (=hostT,REQ_PROG,Encrypt11,N11,TS11,MAC11)=m11 in
 let Res11 = verifyMAC((tg19,REQ_PROG,Encrypt11,N11,TS11),MAC11, SesK) in
 if Res11 = validMAC then
 if REQ_PROG=req_progMode then
 let (=tg16, Firmware) = decryptSK(Encrypt11, sk_ecu) in
```
.
```
 out (c, (ecu_-
id,Ack,minus(N11,one),ts12,MAC((tg20,Ack,minus(N11,one),ts12),SesK))); (*
Msg 12 *)
```
event endComECU_DT(ecu_id, hostT).
.
(* The process for the OEM server *)
```
let processOEMserver =
 in (c, m9); (* Msg 9 *)
 let (hostL, Encrypt9, N9, TS9, REQ_FIRM, Sign9, Cert9)=m9 in
 let PK_DT = getID(Cert9) in
 let signCert9 = getSign(Cert9) in
```
.
```
 let valid9 = verifySign((tg15,PK_DT),signCert9, pk_ca) in
 if valid9 = validSign then
 let Res12 = verifySign((tg14,Encrypt9, N9, TS9, REQ_FIRM),Sign9,PK_DT) in
 if Res12 = validSign then
```
.
```
 if REQ_FIRM = req_firmware then
 let (=tg12,Pk_ECU) = decryptSK(Encrypt9, sk_sak) in
```
.
event beginComDT_OEM(hostL, oem_id);

```
.
 new ts10;
 new firmware1;
.
 out (c, (oem_id, encryptPK((tg16,firmware1),Pk_-
ECU),minus(N9,one),ts10,encryptSK(Hash((tg17,encryptPK((tg16,firmware1),Pk_-
ECU),minus(N9,one),ts10)),sk_oem),Cert(pk_oem,encryptSK(Hash((tg18,pk_-
oem)),sk_ca)))); (* Msg 10
*)
 event endComOEM_DT(oem_id, hostL).
.
```

3.7.6 Results for Verification of Authenticity (ProVerif)

The results for authenticity in the *Diagnosis* phase are presented in Table 24 whilst those corresponding to the *Download* phase are summarized in Table 25.

Table 24 Results for verification of Authenticity in the *Diagnosis* phase

Verification Scheme	ProVerif Model and queries for injective agreements
File model	pi_EVITA_FlashingDiagnosis_Authenticity_1
Processes in the model	*processDT, processCCU, processECU*
Injective agreement DT and CCU	*query evinj* : $endComDT_ECU(x1, x2)$ ==> *evinj* : $beginComDT_ECU(x1, x2)$.
Injective agreement CCU and DT	*query evinj* : $endComCCU_DT(x1, x2)$ ==> *evinj* : $beginComCCU_DT(x1, x2)$
Injective agreement CCU and ECU	*query evinj* : $endComCCU_ECU(x1, x2)$ ==> *evinj* : $beginComCCU_ECU(x1, x2)$
Injective agreement ECU and CCU	*query evinj* : $endComECU_CCU(x1, x2)$ ==> *evinj* : $beginComECU_CCU(x1, x2)$
Secrecy assumptions	*not attacker:sk_ca. not attacker:sk_dt. not attacker:sk_ccu. not attacker:sk_ecu. not attacker:PsSK_ccu. not attacker:SesK. not attacker:pk_ecu. not attacker:ccu_id.*
Nb. of phases	no phases were used
Nb. of rules for completion	200
Forcing completion	Tags were used
RESULT agreement DT and CCU	False. An attack trace is found.
RESULT agreement CCU and DT	False. An attack trace is found.
RESULT agreement CCU and ECU	False. An attack trace is found.
RESULT agreement ECU and CCU	False. An attack trace is found.
Observations	The model is not able to detect replayed messages. Consequently attack traces are found.

Table 25 Results for verification of Authenticity in the *Download* phase

Verification Scheme	ProVerif Model and queries for injective agreements
File model	pi_EVITA_FlashingDownload_Authenticity_1
Processes in the model	*processDT, processECU, processOEMserver*
Injective agreement DT and ECU	$query\ evinj\ :\ endComDT_ECU(x1,x2)\ ==>\ evinj:beginComDT_ECU(x1,x2)$
Injective agreement ECU and DT	$query\ evinj\ :\ endComECU_DT(x1,x2)\ ==>\ evinj:beginComECU_DT(x1,x2)$
Injective agreement DT and OEMserver	$query\ evinj\ :\ endComDT_OEM(x1,x2)\ ==>\ evinj:beginComDT_OEM(x1,x2)$
Injective agreement OEMserver and DT	$query\ evinj\ :\ endComOEM_DT(x1,x2)\ ==>\ evinj:beginComOEM_DT(x1,x2)$
Secrecy assumptions	*not attacker:sk_ca. not attacker:sk_dt. not attacker:sk_ecu. not attacker:sk_oem. not attacker:sk_sak. not attacker:SesK. not attacker:pk_ecu. not attacker:ecu_id. not attacker:firmware1.*
Nb. of phases	no phases were used
Nb. of rules for completion	200
Forcing completion	Tags were used
RESULT agreement DT and CCU	False. An attack trace is found.
RESULT agreement ECU and DT	False. An attack trace is found.
RESULT agreement DT and OEMserver	False. An attack trace is found.
RESULT agreement OEMserver and DT	False. An attack trace is found.
Observations	The model is not able to detect replayed messages. Consequently attack traces are found.

3.8 CAM-LDW Protocol

The CAM-LDW Protocol is based upon use cases 1 and 2 specified in EVITA D2.1 (See [14], *U.C.1-Safety Reaction: Active Brake* and *U.C.2-Local Danger Warning from other Cars*). Indeed, this protocol aims to secure Cooperative Awareness Messages (CAMs). The purpose of a CAM is to send a warning to other cars and drivers: CAM are used in case of critical situations. Hence, a CAM may contain information about Local Danger Warnings (LDW) like emergency braking, obstacles, car accidents, etc. CAMs can come from a specialized Road Side Unit (RSU) or directly from a vehicle. In agreement with EVITA D3.3 [18], CAMs we propose to model are initiated by vehicles. Last, the CAM-LDW Protocol is a relevant protocol with respect to EVITA goals. Figure 23 gives an overview of the protocol.

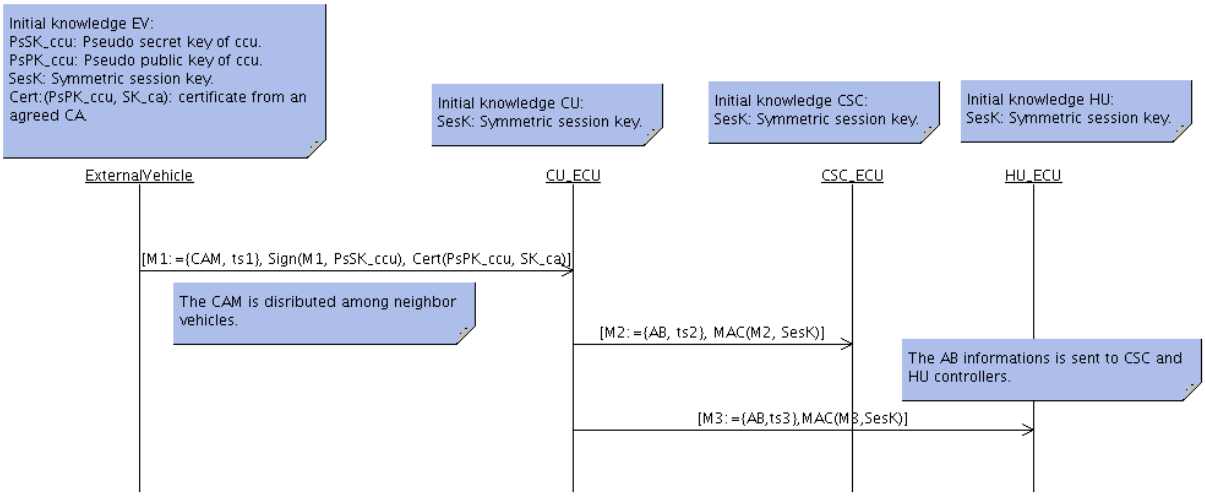


Figure 23 Sequence Diagram of CAM-LDW protocol

3.8.1 Protocol Description

The protocol involves four CEs: an External Vehicle (EV), an in-car Communications Unit (CU-ECU), an in-car Chassis and Safety Controller (CSC-ECU) and an in-car Head Unit (HU-ECU). The last three CE's are in the same vehicle. The protocol is triggered when EV broadcasts a CAM among neighbor vehicles. Thus, in a very first stage, EV performs an Active Brake (AB) maneuver because of an emergency situation. Right after, EV position, dynamics and time of the AB are retrieved in order to compose a CAM. To preserve privacy, and ensure authenticity and integrity, the message is signed with a short-term key pair $(PsSK\_ccu, PsPK\_ccu)$ of pseudo keys. Since the key pair is ignored by neighborhood cars, the message includes the certificate that binds $PsPK\_ccu$. Such certificate has been issued by an agreed Certification Authority (CA) for car-to-car communication. Right after reception by the target car, the CAM is validated by its CU-ECU. More precisely, the certificate is verified and the signature is authenticated. Right after and according to CAM contents, an algorithm is executed to determine plausibility of emergency brake (AB), the result of which is sent to CSC-ECU (Message 2). Since we assume that CU-ECU, CSC-ECU as well as HU-ECU share the symmetric session key $SesK$, Message 2 is time stamped and MAC protected with $SesK$. CSC-ECU receives Message 2 and in case of valid MAC, an algorithm to process AB is executed. In parallel with Message 2, a copy of plausibility check results is sent to HU-ECU (Message 3). This message is also time stamped and MAC protected with the $SesK$. After MAC validation, the HU-ECU displays the corresponding alert in the Human Machine Interface (HMI).

3.8.2 Targeted Security Properties

According to Security Requirements specification, EVITA D2.3 [17], our model targets the following properties:

1. *Authenticity_1* (see [17], page 25)
2. *Authenticity_2* (see [17], page 25)

3. *Authenticity_3* (see [17], page 25)
4. *Authenticity_4* (see [17], page 25)
5. *Authenticity_5* (see [17], page 25)
6. *Authenticity_6* (see [17], page 26)
7. *Authenticity_7* (see [17], page 26)
8. *Authenticity_8* (see [17], page 26)
9. *Authenticity_9* (see [17], page 26)
10. *Authenticity_10* (see [17], page 27)
11. *Authenticity_11* (see [17], page 27)
12. *Authenticity_101* (see [17], page 31)
13. *Authenticity_103* (see [17], page 32)
14. *Privacy_102* (see [17], page 40)
15. *Privacy_103* (see [17], page 41)
16. *Privacy_105* (see [17], page 41)

3.8.3 Model for Verification of Confidentiality (ProVerif)

CAM-LDW Protocol model for verification of confidentiality is composed by the following basic sections described in section 3.4.1: *Basic Blocks*, *Hypotheses*, *Queries*, *Variables*, *Processes* and *Main*.

1. Basic Blocks
   Since symmetric and asymmetric cryptography is used, the *Basic Blocks* include respective primitives. Additionally, structures to manipulate CAMs as well as related Active Brake information are included (e.g., `getDataX`). Moreover, functions like `pseudoCarID` and `pseudoDriverID` are meant to associate real and pseudo identities. Since such functions are not private, they are used by the attacker reasoning to determine privacy attacks on car and driver information.

```
 (* To decrypt/encrypt messages with a key *)
.
fun encrypt/2.
reduc decrypt(encrypt(x,k),k)=x.
.
 (* MACs in messages *)
.
fun MAC/2.
```

```
fun verifyMAC/3.
.
equation verifyMAC(m, MAC(m, k), k)=validMAC.
.
 (* Public key encryption/decryption *)
fun Pk/1.
.
fun encryptPK/2.
fun encryptSK/2.
.
reduc decryptSK(encryptPK(m,Pk(k)),k)=m.
reduc decryptPK(encryptSK(m,k),Pk(k))=m.
.
 (* Create/verify signatures *)
fun Hash/1.
fun verifySign/3.
.
equation verifySign(m,encryptSK(Hash(m),k),Pk(k))=validSign.
.
 (* Create/verify certificates *)
.
fun Cert/2.
.
reduc getID(Cert(id,sign))=id.
reduc getSign(Cert(id,sign))=sign.
.
 (* Related Operative Awareness Message functions *)
.
fun CAM/3.
fun AB/3.
.
reduc getAB(CAM(ps_carx, ps_drvy,AB(pos, dyn, ts)))=AB(pos,dyn,ts).
reduc getPsCar(CAM(ps_carx, ps_drvy,AB(pos, dyn, ts)))=ps_carx.
reduc getPsDrv(CAM(ps_carx, ps_drvy,AB(pos, dyn, ts)))=ps_drvy.
.
reduc getPosition(AB(pos, dyn, ts))=pos.
reduc getDynamics(AB(pos, dyn, ts))=dyn.
reduc getTime(AB(pos, dyn, ts))=ts.
.
 (* Association between real and pseudo identities *)
.
fun pseudoCarID/1.
fun pseudoDriverID/1.
.
```

2. Hypotheses
The model *Hypotheses* are compliant with confidential data which should be initially restricted for the attacker; apart from involved secret keys, we additionally assume that car and driver identifications are initially ignored by the attacker. Thus, it is

verified that car IDs -ev_ccu- and driver IDs -ev_driver- can not be derived from the initial knowledge of the attacker. Note that CAM contents like car position, dynamics and time of Active Brake are not considered as confidential.

```
 (* Initial secrecy assumptions ( the keys can not be derived from
clauses ) *)
.
not attacker:ev_ccu.
not attacker:ev_driver.
not attacker:SesK.
not attacker:PsSK_ccu.
not attacker:sk_ca.
```

.

3. Queries
   Queries model that confidential material is not delivered to the attacker as a result of the protocol exchanges. Indeed, we assume that if the attacker is able to derive `ev_ccu` or `ev_driver`, an attack on car's or driver's privacy can be achieved.

```
 (* Queries for Verification of Secrecy/Confidentiality *)
.
query attacker:ev_ccu.
query attacker:ev_driver.
query attacker:SesK.
query attacker:PsSK_ccu.
query attacker:sk_ca.
```

.

4. Variables
   Since the model was not tagged, the *Variables* section only includes channels declaration, as well as a few free variables and data structures. `c` represents the wireless medium, and `c1` and `c2` model in-car channels. Indeed, they represent an ECU internal bus for calling the Security Watch Dog module (SWD), and the main CAN bus, respectively.

```
 (* The channels in the model *)
free c.
private free c1.
free c2.
.
 (* Declaration of some variables that the attacker may know *)
.
data validMAC/0.
data validSign/0.
data true/0.
data cu_id/0.
free PsPK_ccu.
```

.

5. Processes
   The *Processes* section is composed by four processes, each one associated to a CE, namely `processEV`, `processCU_ECU`, `processCSC_ECU` and `processHU_ECU`. Additionally, three dummy processes are defined for representing SWD calls, Active Brake algorithm executions and display alert functions. Note that the key pair $(PsSK\_ccu, PsPK\_ccu)$ is internally generated in `processEV`. Moreover, the `processEV` uses pseudo car and driver identifications – `ps_ev_ccu` and `ps_ev_driver`-. These pseudo names are associated to the private in-car ID's using the blocks `pseudoCarID` and `pseudoDriverID`, respectively.

```
 (* The process for External Vehicle *)
.. let processEV =
 new ev_ccu;
 let ps_ev_ccu = pseudoCarID(ev_ccu) in
.
 new ev_driver;
 let ps_ev_driver = pseudoDriverID(ev_driver) in
.
 new PsSK_ccu;
 let PsPK_ccu = Pk(PsPK_ccu) in
.
 new position;
 new dynamics;
 new tsAB;
.
 new ts1;
 out (c, (CAM(ps_ev_ccu,ps_ev_-
driver,AB(position,dynamics,tsAB)),ts1,encryptSK(Hash((CAM(ps_ev_ccu,ps_-
ev_driver,
 AB(position,dynamics,tsAB)),ts1)),PsSK_ccu),Cert(PsPK_-
ccu,encryptSK(Hash(PsPK_ccu),sk_ca)))); (* Msg 1
*)
 0.
.
 (* The process for CU-ECU *)
let processCU_ECU =
 in (c, m1); (* Msg 1 *)
 let (Warning,TS1, Sign1, Cert1) = m1 in
 let PS_PK = getID(Cert1) in
 let signCert1 = getSign(Cert1) in
 let Res1 = verifySign(PS_PK, signCert1, pk_ca) in
 if Res1 = validSign then (
.
 let Res2 = verifySign((Warning,TS1),Sign1,PS_PK) in
 if Res2 = validSign then (
.
 (* Performing Plausibility Check *)
.
```

```
 let ActiveBrake = getAB(Warning) in
.
 let POS= getPosition(ActiveBrake) in
 let DYN = getDynamics(ActiveBrake) in
 let TS = getTime(ActiveBrake) in
.
 new ts2;
 out (c2, (cu_id, ActiveBrake, ts2, MAC((ActiveBrake,ts2),SesK))); (* Msg
2 *)
.
 new ts3;
 out (c2, (cu_id, ActiveBrake, ts3, MAC((ActiveBrake,ts3),SesK))) (* Msg 3
*)
.
 )
 else ( processSWD)
 )
 else ( processSWD ).
.
 (* The process for the CSC-ECU *)
.
let processCSC_ECU =
 in (c2, m2); (* Msg 2 *)
 let (CU2, AB2, TS2, MAC2) = m2 in
 let Res3 = verifyMAC((AB2, TS2), MAC2, SesK) in
 if Res3 = validMAC then (
 if CU2 = cu_id then
 (* Active Brake evaluation CSC *)
 let POS2 = getPosition(AB2) in
 let DYN2 = getDynamics(AB2) in
 let TS_2 = getTime(AB2) in
.
 executeAB
 )
 else (processSWD).
.
 (* The process for the HU-ECU *)
let processHU_ECU =
 in (c2, m3); (* Msg 3 *)
 let (CU3, AB3, TS3, MAC3) = m3 in
 let Res4 = verifyMAC((AB3, TS3), MAC3, SesK) in
 if Res4 = validMAC then (
 if CU3 = cu_id then
 (* Active Brake warning message *)
 let POS3 = getPosition(AB3) in
 let DYN3 = getDynamics(AB3) in
 let TS_3 = getTime(AB3) in
.
```

```
 displayAB
 )
 else (processSWD).
.
 (* Related process to the AB scenario *)
let processSWD =
 let Alert=true in
 out(c1, Alert).
.
let executeAB = 0.
.
let displayAB = 0.
.
```

6. Main
   The *Main* section includes secret keys used by more than one ECU: *SesK* and *sk_ca*. The basic block `Pk()` is used to associate *pk_ca* to its respective secret key. Later on, *pk_ca* is broadcasted in the channel `c`. Protocol processes are executed in parallel and infinitely replicated thus ensuring that data secrecy and authenticity is preserved no matter the number of protocol sessions.

```
process
 new SesK;
.
 new sk_ca;
 let pk_ca = Pk(sk_ca) in
 out (c, pk_ca);
.
 ((!processEV)|(!processCU_ECU)|(!processCSC_ECU)|(!processHU_ECU))
.
```

3.8.4 Results for Verification of Confidentiality (ProVerif)

The results from verification of confidentiality in CAM-LDW Protocol are summarized in Table 26.

3.8.5 Model for Verification of Authenticity (ProVerif)

The model for verification of authenticity in CAM-LDW protocol is composed of usual sections (see 3.4.1), that is *Basic Blocks*, *Hypotheses*, *Queries*, *Variables*, *Processes* and *Main*.

The sections *Basic Blocks*, *Hypotheses*, *Variables* and *Main* are the same as in the previous model for verification of confidentiality (See section 3.8.3). Thus we only describe *Queries* and *Processes*.

1. Queries
   The *Queries* for verification of authenticity are based upon the concept of injective agreements explained in section 3.4.2. The protocol has three communication flows; between `processEV` and `processCU_ECU`, between `processCU_ECU` and

Table 26 Results for verification of Confidentiality in CAM-LDW Protocol

Verification Scheme	ProVerif Model and attacker queries
File model	pi_EVITA_CAM-LDWProtocol_Confidentiality_1
Processes in the model	*processEV, processCU_ECU, processCSC_ECU, processHU_ECU*
Property representation	*query attacker:ev_ccu. query attacker:ev_driver. query attacker:SesK. query attacker:PsSK_ccu. query attacker:sk_ca.*
Secrecy assumptions	*not attacker:ev_ccu. not attacker:ev_driver. not attacker:SesK. not attacker:PsSK_ccu. not attacker:sk_ca.*
Nb. of phases	No phases were used
Nb. of rules for completion	200
Forcing completion	No tags were used
RESULT *not attacker:ev_ccu*	True
RESULT *not attacker:ev_driver*	True
RESULT *not attacker:SesK*	True
RESULT *not attacker:PsSK_ccu*	True
RESULT *not attacker:sk_ca*	True
Observations	The protocol preserves secrecy of confidential data. Car and driver privacy are also ensured, since th disclosure of their real identity is not possible.

`processCSC_ECU` and between `processCU_ECU` and `processHU_ECU`. The authentication of these exchanges therefore relies on the following queries:

```
(* Queries for Verification of Authenticity *)
.
query evinj:endComEV_CU(x1,x2) ==> evinj:beginComEV_CU(x1,x2).
query evinj:endComCU_CSC(x1,x2) ==> evinj:beginComCU_CSC(x1,x2).
query evinj:endComCU_HU(x1,x2) ==> evinj:beginComCU_HU(x1,x2).
.
```

Thus for instance, the authentication of the first message in the protocol corresponds with the first query. The arguments `xi` for the the first query are `ps_ev_ccu` and `ps_ev_driver`, respectively. The arguments for the second and third query respectively correspond to `cu_id` and `ActiveBrake` values.

2. Processes
The section *Processes* is the same as in the verification of confidentiality, apart from authentication event which are located accordingly to queries:

```
(* The process for External Vehicle *)
.
```

```
let processEV =
 new ev_ccu;
 let ps_ev_ccu = pseudoCarID(ev_ccu) in
.
 new ev_driver;
 let ps_ev_driver = pseudoDriverID(ev_driver) in
.
 new PsSK_ccu;
 let PsPK_ccu = Pk(PsPK_ccu) in
.
 new position;
 new dynamics;
 new tsAB;
.
 new ts1;
.
 event beginComEV_CU(ps_ev_ccu, ps_ev_driver);
.
 out (c, (CAM(ps_ev_ccu,ps_ev_-
driver,AB(position,dynamics,tsAB)),ts1,encryptSK(Hash((CAM(ps_ev_ccu,ps_-
ev_driver,AB(position,dynamics,tsAB)),ts1)),PsSK_ccu),Cert(PsPK_-
ccu,encryptSK(Hash(PsPK_ccu),sk_ca)))); (* Msg 1
*)
 0.
.
 (* The process for CU-ECU *)
let processCU_ECU =
 in (c, m1); (* Msg 1 *)
 let (Warning,TS1, Sign1, Cert1) = m1 in
 let PS_PK = getID(Cert1) in
 let signCert1 = getSign(Cert1) in
 let Res1 = verifySign(PS_PK, signCert1, pk_ca) in
 if Res1 = validSign then (
.
 let Res2 = verifySign((Warning,TS1),Sign1,PS_PK) in
 if Res2 = validSign then (
.
 let CAR_ID = getPsCar(Warning) in
 let DRV_ID = getPsDrv(Warning) in
.
 event endComEV_CU(CAR_ID, DRV_ID);
.
 (* Performing Plausibility Check *)
 let ActiveBrake = getAB(Warning) in
.
 let POS= getPosition(ActiveBrake) in
 let DYN = getDynamics(ActiveBrake) in
 let TS = getTime(ActiveBrake) in
```

```
.
 event beginComCU_CSC(cu_id, ActiveBrake);
 event beginComCU_HU(cu_id, POS);
.
 new ts2;
 out (c2, (cu_id, ActiveBrake, ts2, MAC((ActiveBrake,ts2),SesK))); (* Msg
2 *)
.
 new ts3;
 out (c2, (cu_id, ActiveBrake, ts3, MAC((ActiveBrake,ts3),SesK))) (* Msg 3
*)
.
 )
 else ( processSWD)
 )
 else ( processSWD ).
.
 (* The process for the CSC-ECU *)
.
let processCSC_ECU =
 in (c2, m2); (* Msg 2 *)
 let (CU2, AB2, TS2, MAC2) = m2 in
 let Res3 = verifyMAC((AB2, TS2), MAC2, SesK) in
 if Res3 = validMAC then (
 if CU2 = cu_id then
 (* Active Brake evaluation CSC *)
 let POS2 = getPosition(AB2) in
 let DYN2 = getDynamics(AB2) in
 let TS_2 = getTime(AB2) in
.
 event endComCU_CSC(CU2, AB2);
.
 executeAB
 )
 else (processSWD).
.
 (* The process for the HU-ECU *)
.
let processHU_ECU =
 in (c2, m3); (* Msg 3 *)
 let (CU3, AB3, TS3, MAC3) = m3 in
 let Res4 = verifyMAC((AB3, TS3), MAC3, SesK) in
 if Res4 = validMAC then (
 if CU3 = cu_id then
 (* Active Brake warning message *)
 let POS3 = getPosition(AB3) in
 let DYN3 = getDynamics(AB3) in
 let TS_3 = getTime(AB3) in
```

```
.
 event endComCU_HU(CU3,AB3);
.
 displayAB
 )
 else (processSWD).
```

.

3.8.6 Results for Verification of Authenticity (ProVerif)

The results are presented in Table 27

Table 27 Results for verification of Authenticity in CAM-LDW protocol

Verification Scheme	ProVerif Model and queries for injective agreements
File model	pi_EVITA_CAM-LDWProtocol_Authenticity_1
Processes in the model	*processEV, processCU_ECU, processCSC_ECU, processHU_ECU*
Injective agreement EV and CU_-ECU	$query\ evinj\ :\ endComEV\_CU(x1,x2)\ ==>\ evinj : beginComEV\_CU(x1,x2)$
Injective agreement CU_ECU and CSC_ECU	$query\ evinj\ :\ endComCU\_CSC(x1,x2)\ ==>\ evinj : beginComCU\_CSC(x1,x2)$
Injective agreement CU_ECU and HU_ECU	$query\ evinj\ :\ endComCU\_HU(x1,x2)\ ==>\ evinj : beginComCU\_HU(x1,x2)$
Secrecy assumptions	*not attacker:ev_ccu. not attacker:ev_driver. not attacker:SesK. not attacker:PsSK_ccu. not attacker:sk_ca.*
Nb. of phases	no phases were used
Nb. of rules for completion	200
Forcing completion	No tags were used
RESULT agreement EV and CU_-ECU	False. An attack trace is found.
RESULT agreement CU_ECU and CSC_ECU	False. An attack trace is found.
RESULT agreement CU_ECU and HU_ECU	False. An attack trace is found.
Observations	The model is not able to detect replayed messages. Consequently attack traces are found.

3.9 Model Limitation in ProVerif

This section presents a limitation in ProVerif models for the verification of authenticity properties. This limitation concerns the analysis of replay attacks.

3.9.1 General Description

Attack traces have been identified during the verification of authenticity properties (see for example Table 24). However these traces do not correspond to attacks on EVITA protocols, but rather to a limitation of our modeling approach in ProVerif. Indeed, to our knowledge, apart from `phase`, ProVerif does not provide structures to represent time in processes. As a consequence of this limitation, there is no specific registers for time. Moreover, since ProVerif does not introduce any algebraic structure in variable domains, they can not be compared with respect to an order. Thus, time stamps validation with respect to relative or global time scales is missing.

As it was explained in section 3.4.2, authenticity is verified based upon an injective agreement between two CE's processes. Roughly speaking, the injective agreement holds if whenever the event `end(M)` is executed in $\texttt{processB}_\rho$, then at least one event `begin(M)` has been executed in process $\texttt{processA}_\rho$. Moreover, once executed events – `end_ex(M)` and `begin_ex(M)` – are associated in a protocol session ρ, they can not be reused in a different one.

Finally, the limitation appears in ProVerif models that represent exchanges of the form:

$$M := (d_1, d_2, \ldots, d_n, tsX), MAC(M, k) \text{ or}$$
$$M := (d_1, d_2, \ldots, d_n, tsX), Sign(M, k),$$

where tsX is the respective time stamp. Thus, whenever a $\texttt{processA}_\rho$ outputs such an exchange, an event `beginA_B(M)` is executed. If a $\texttt{processB}_\rho$ inputs the message, then an event `endA_B(M)` is executed. Nevertheless, due to time leakage representation, $\texttt{processB}_\rho$ is not able to determine whether M has been accepted in a previous protocol session or not. Knowing that, the attacker opens a new session by replication of $\texttt{processA}_\rho$ and $\texttt{processB}_\rho$, namely ρ_1. Then, the attacker simply assumes that $\texttt{processA}_{\rho_1}$ is not executed and afterwards replays M – and its authentication code – to $\texttt{processB}_{\rho_1}$. Thus, the event `endA_B(M)` is executed in $\texttt{processB}_{\rho_1}$ with no prior execution of event `beginA_B(M)`. Consequently, the following query is not satisfied for session ρ_1:

```
query evinj:endA_B(M) ==> evinj:beginA_B(M).
```

Since the attacker is able to replay time stamped messages, the injective agreement is never satisfied for those messages and consequently attack traces on authenticity are found.

3.9.2 Attack Trace Analysis

In the following lines we present an instance of an attack trace on authenticity for the CAM-LDW Protocol. Trace notation has been modified in order to simplify its reading. The query that leads to this trace is the following one:

```
query evinj:endComEV_CU(x1,x2) ==> evinj:beginComEV_CU(x1,x2).
```

Table 28 Attack Trace on authenticity for CAM-LDW Protocol

Attack Trace Sequence
1. The event $beginComEV\_CU(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791))$ (with environment $@sid\_232 = @sid\_1770$, $@occ55\_1034 = @occ\_cst()$) may be executed at {55}. So the message $M1_{s1} := (CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794)), ts1\_1795, encryptSK(Hash((CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794)), ts1\_1795)), PsSK\_ccu\_1796), Cert(Pk(PsSK\_ccu\_1796), encryptSK(Hash(Pk(PsSK\_ccu\_1796)), sk\_ca\_43[])))$ may be sent to the attacker at output {56}. attacker:$M1_{s1}$.
Description: `processEC` sends the message $M1_{s1}$ in session $s1$.
2. By 1, the attacker may know $M1_{s1}$. Using the 3th inverse of function 4-tuple the attacker may obtain $Cert(Pk(PsSK\_ccu\_1796), encryptSK(Hash(Pk(PsSK\_ccu\_1796)), sk\_ca\_43[]))$. attacker:$Cert(Pk(PsSK\_ccu\_1796), encryptSK(Hash(Pk(PsSK\_ccu\_1796)), sk\_ca\_43[]))$.
Description: $M1_{s1}$ is intercepted; the attacker obtains the certificate in $M1_{s1}$.
3. By 1, the attacker may know $M1_{s1}$. Using the 2th inverse of function 4-tuple the attacker may obtain $encryptSK(Hash((CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794)), ts1\_1795)), PsSK\_ccu\_1796)$. attacker:$encryptSK(Hash((CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794)), ts1\_1795)), PsSK\_ccu\_1796)$.
Description: The attacker obtains the signature of $M1_{s1}$.
4. By 1, the attacker may know $M1_{s1}$. Using the 1th inverse of function 4-tuple the attacker may obtain $ts1\_1795$. attacker:$ts1\_1795$.
Description: The attacker retrieves $ts1\_1795$
5. By 1, the attacker may know $M1_{s1}$. Using the 0th inverse of function 4-tuple the attacker may obtain $CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794))$. attacker:$CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794))$.
Description: The attacker begins to forge a new message with acquired knowledge.
6. By 5, the attacker may know $CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794))$. **6.1** By 4, the attacker may know $ts1\_1795$. **6.2** By 3, the attacker may know $encryptSK(Hash((CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794)), ts1\_1795)), PsSK\_ccu\_1796)$. **6.3** By 2, the attacker may know $Cert(Pk(PsSK\_ccu\_1796), encryptSK(Hash(Pk(PsSK\_ccu\_1796)), sk\_ca\_43[]))$.

Continued on next page

Table 28 Attack Trace on authenticity for CAM-LDW Protocol

Attack Trace Sequence
6.4 Using the function 4-tuple the attacker may obtain $M1'_{s1} = (CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794)), ts1\_1795, encryptSK(Hash((CAM(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791), AB(position\_1792, dynamics\_1793, tsAB\_1794)), ts1\_1795)), PsSK\_ccu\_1796), Cert(Pk(PsSK\_ccu\_1796), encryptSK(Hash(Pk(PsSK\_ccu\_1796)), sk\_ca\_43[])))$. attacker:$M1'_{s1}$.
Description: The attacker forges the new message $M1'_{s1}$. Indeed $M1'_{s1}$ is a copy of $M1_{s1}$ an can be used in other protocol sessions.
7. The message $M1'_{s1}$ that the attacker may have by 6 may be received at input {28}. So event $endComEV\_CU(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791))$ may be executed at {42} in session $endsid\_1787$. end:$endsid\_1787$, $endComEV\_CU(pseudoCarID(ev\_ccu\_1790), pseudoDriverID(ev\_driver\_1791))$.
Description: The attacker send the forged message $M1'_{s1}$ to `processCU_ECU`
A trace has been found. I am now trying to reconstruct a trace that falsifies injectivity.
Description: ProVerif shows the attack based upon facts 1-7.
I. $out(c, Pk(sk\_ca\_43\_12))$ at {2}
Description: Action in the main process.
II. $event(beginComEV\_CU(pseudoCarID(ev\_ccu\_84\_5), pseudoDriverID(ev\_driver\_86\_6)))$ at {55} in copy a_4
Description: The event `beginComEV_CU` is executed in `processEV` in session a_4
III. $out(c, M1_{s2} := (CAM(pseudoCarID(ev\_ccu\_84\_5), pseudoDriverID(ev\_driver\_86\_6), AB(position\_90\_7, dynamics\_91\_8, tsAB\_92\_9)), ts1\_93\_10, encryptSK(Hash((CAM(pseudoCarID(ev\_ccu\_84\_5), pseudoDriverID(ev\_driver\_86\_6), AB(position\_90\_7, dynamics\_91\_8, tsAB\_92\_9)), ts1\_93\_10)), PsSK\_ccu\_88\_11), Cert(Pk(PsSK\_ccu\_88\_11), encryptSK(Hash(Pk(PsSK\_ccu\_88\_11)), sk\_ca\_43\_12))))$ at {56} in copy a_4
Description: The message $M1_{s2}$, is sent by `processEV` in session a_4
IV. $in(c, M1_{s2})$ at {28} in copy a_3
Description: The message $M1_{s2}$ is input by `processCU_ECU` in session a_3
V. $event(endComEV\_CU(pseudoCarID(ev\_ccu\_84\_5), pseudoDriverID(ev\_driver\_86\_6)))$ at {42} in copy a_3
Description: Event `endComEV_CU` is executed by `processCU_ECU` in session a_3
VI. $event(beginComCU\_CSC(cu\_id(), AB(position\_90\_7, dynamics\_91\_8, tsAB\_92\_9)))$ at {47} in copy a_3
Description: Event `beginComCU_CSC` is executed by `processCU_ECU` in session a_3
VII. $event(beginComCU\_HU(cu\_id(), position\_90\_7))$ at {48} in copy a_3
Description: The event `beginComCU_HU` is executed by `processCU_ECU` in session a_3
VIII. $out(c2, (cu\_id(), AB(position\_90\_7, dynamics\_91\_8, tsAB\_92\_9), ts2\_82\_15, MAC((AB(position\_90\_7, dynamics\_91\_8, tsAB\_92\_9), ts2\_82\_15), SesK\_42\_13)))$ at {49} in copy a_3
Description: Active Brake information is sent by `processCU_ECU` to `processCSC_ECU` in session a_3

Continued on next page

Table 28 Attack Trace on authenticity for CAM-LDW Protocol

Attack Trace Sequence
IX. $out(c2,\ (cu\_id(),\ AB(position\_90\_7,\ dynamics\_91\_8,\ tsAB\_92\_9),\ ts3\_83\_16,\ MAC((AB(position\_90\_7,\ dynamics\_91\_8,\ tsAB\_92\_9),\ ts3\_83\_16),\ SesK\_42\_13)))$ at {50} in copy a_3
Description: Active Brake information is sent by `processCU_ECU` to `processHU_ECU` in session a_3
X. $in(c,\ M1_{s2})$ at {28} in copy sid_1957_14
Description: The attacker opens the session sid_1957_14 and replays the message $M1_{s2}$) to `processCU_ECU`.
XI. $event(endComEV\_CU(pseudoCarID(ev\_ccu\_84\_5), pseudoDriverID(ev\_driver\_86\_6)))$ at {42} in copy sid_1957_14
Description: Event `endComEV_CU` is executed by `processCU_ECU` in session sid_1957_14.
XII. The event $endComEV\_CU(pseudoCarID(ev\_ccu\_84\_5), pseudoDriverID(ev\_driver\_86\_6))$ is executed in session sid_1957_14 and in session a_3. A trace has been found.
Description: ProVerif informs that event `endComEV_CU` has been executed in two different sessions.
RESULT $evinj : endComEV\_CU(x1\_935,\ x2\_936) ==> evinj : beginComEV\_CU(x1\_935,\ x2\_936)$ is false. RESULT (but $ev : endComEV\_CU(x1\_1749,\ x2\_1750) ==> ev : beginComEV\_CU(x1\_1749,\ x2\_1750)$ is true.)
Description: Event `endComEV_CU` does not match with any event `beginComEV_CU` in session sid_1957_14. Therefore the injective agreement is not held.

Continued on next page

3.9.3 Conclusions

Based upon presented traces, we can conclude that the attacker is able to replay messages whenever message authenticity depends on time stamps. This is a strong limitation of our model, and therefore, we intend to use a model of time in order to more easily verify authenticity in EVITA protocols, i.e., verify authenticity properties without having to verify attack traces by hand to check whether they correspond to reply attacks due to a lack of your modeling approach, or not.

3.10 Summary of proofs

The magnified view verification approach targets the formal verification of EVITA cryptographic protocols. For that purpose, three complementary techniques – TURTLE, ProVerif, AVATAR – have been defined, and then used for three different protocols: Keying with Key Master, Remote Flashing Update and CAM-LDW. TURTLE offers a high level language (UML) for system modeling. Even if its Dolev-Yao based attacker model can be used for a wide range of security properties, it was mainly used for integrity, and also for freshness proof thanks to the timed model of TURTLE.

In the case of privacy, confidentiality and also authenticity properties, the ProVerif approach provides a strong and formal framework relying on horn clauses resolution.

Nevertheless, ProVerif modeling and results evaluation require strong skills in pi-calculus, and in ProVerif queries. Additionaly, ProVerif requires to make a separate model from the one of the system design, which might be error-prone. At last, the AVATAR profile is conceived for taking the best of the two first approaches: a timed-model, and integration of the model with the model of the system design.

Only one security flaw was identified with the verification process we applied, with respect to a set of assumptions listed at the beginning of this section. Obviously, extending attacker models, properties, and verification technique capabilities might lead to the identification of other security flaws.

4 Compositional Verification Results

In this section we will demostrate the application of a SeBB-based formal verification process to an exemplary deployment of the EVITA architecture to a single of EVITA's use cases. After introducing the necessary formal concepts, we will present the exemplary instantiation of the EVITA architecture and protocols regarding the requirement to be verified here. Then an analysis of the EVITA security architecture is performed and its security relevant properties are extracted and formalized. In this process also additional properties that were not yet foreseen (e.g. properties of secure software) will be identified. Afterwards the secure onboard protocols will be discussed and their corresponding SeBBs formalized. Finally the actual proof will be attempted and necessary trust relations required for the actual success of the proof will be extracted.

Please note that this section will not perform an exhaustive analysis of a complete architecture utilizing the EVITA results. This would require the provision of a real deployment that needs to be defined for a real vehicle in order to proof the relevant security properties. Rather, this section demonstrates an appropriate verification approach for deployment scenarios in the future. We will deeply investigate the properties of a sensor, a keymaster, and a subsequent application ECU, as well as their communication with respect to their HSMs, HSM-integration, software, key distribution, firmware update and secure boot behavior. However these results can be applied similarly to the rest of the use case as well as the refined requirements of other use cases and security requirements. [13] gives an overview over a set of refined and consolidated requirements of more use cases and security requirements that demonstrate their similarities.

4.1 Necessary Formal Concepts

The work presented in this deliverable is based on the Security Modelling Framework SeMF and the concept of SeMF's Security Building Blocks (SeBBs). These concepts have already been explained in [13].

This section will introduce all additional formal concepts needed besides those that have already been explained in the above mentioned deliverables of the EVITA project.

4.1.1 Semantics

Systems' indexing During the rest of this text, there will be several occasions in which systems are indexed multiple times, due to expressing e.g. A's trust in B's trust in C's trusted system. To improve readability, instead of $S_{C_{B_A}}$ we will write $S_{C \leftarrow B \leftarrow A}$ which corresponds to $trust(A, trust(B, trust(C, \ldots)))$.

4.1.2 SeMF-Definitions

Limited Precede Sometimes a precede property cannot be expressed solely by a single trigger action. It may be the case that not one but two or more actions together imply precedence by another action.

Definition 1 (Limited-Precede)**.** *Let B be a set of sequences of actions, then limited precede denoted by limited-precede(a, b & c) holds if for all $\omega \in B$ with $b, c \in alph(\omega)$ it holds that $a \in alph(\omega)$.*

Phases When verifying the security of an Evita system, what can or cannot happen within software boot cycles is an important aspect to be considered. In order to model these boot cycles, we use the definition of a *phase* provided in [12]. A phase $V \subset \Sigma^*$ is a prefix closed language consisting only of words which, as long as they are not maximal in V, show the same continuation behavior within V as within B.

Definition 2. *Let $B \subseteq \Sigma^*$ be a system. A prefix closed language $V \subset \Sigma^*$ is a phase in B if the following holds:*

1. $V \cap \Sigma \neq \emptyset$
2. $\forall \omega \in B$ *with* $\omega = uv$ *and* $v \in V \setminus (max(V) \cup \{\varepsilon\})$ *holds:* $\omega^{-1}(B) \cap \Sigma = v^{-1}(V) \cap \Sigma$

Thus a phase as defined above is essentially a part of the system behaviour that is closed with respect to concatenation. In analogy to the maximal words of a phase V which are those $v \in V$ for which exists $\omega, u \in B$ with $\omega = uv$ such that for all $a \in \Sigma$ with $\omega a \in B$ holds $va \notin V$, we define the minimal words of a phase as all $v \in V$ with $|alph(v)| = 1$.

A phase can be a very complex construct. However, in many cases phases are of interest that can be defined by their starting and ending actions. Since an action can occur more than once in a word, it is not sufficient to identify the starting and terminating actions for determining where a particular phase starts and where it ends. The following definition takes this into account.

In the following, we denote the number of occurences of a set of actions $\Gamma \in \Sigma$ in a word ω by $card(\Gamma, \omega)$. If Γ consists of only one action a, we simply say $card(a, \omega)$.

Definition 3. *Let $s_1, \ldots, s_k, t_1, \ldots, t_l \in \Sigma$ be actions. Then $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$ (with $j_1, \ldots, j_l \in \mathbb{N}$) defines a phase in B that starts with actions $s_1, \ldots, s_k$ and terminates with actions $t_1, \ldots, t_l$ in the following sense:*

- *For all $\omega \in B$ for which exists $s_j \in \{s_1, \ldots, s_k\}$ such that $\omega s_j \in B$ follows $s_j \in V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$ (i.e. s_j is minimal in $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$).*
- *For all $\omega \in B$ for which exists $t_i(j_i) \in \{t_1(j_1), \ldots, t_l(j_l)\}$ and $u, v \in \Sigma^*$ with $\omega = uvt_i$, $vt_i \in V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$, and $card(t_i, vt_i) = j_i$ follows that vt_i is maximal in $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$.*

In the above definition, the starting action(s) need to be fixed so that starting from these the terminating actions occuring in the phase can be counted in order to identify those ones that actually terminate the phase. In the following definition, we conversely fix the termination action(s) and count the number of occurences of the starting action(s) backwards in the phase to identify the actual start(s) of the phase:

Definition 4. *Let $s_1, \ldots, s_k, t_1, \ldots, t_l \in \Sigma$ be actions. Then $V(\{s_1(i_1), \ldots, s_k(i_k)\}, \{t_1, \ldots, t_l\}) \subseteq B$ (with $i_1, \ldots, i_l \in \mathbb{N}$) defines a phase in B that starts with actions $s_1, \ldots, s_k$ and terminates with actions $t_1, \ldots, t_l$ in the following sense:*

- *For all $\omega \in B$ for which exists $s_j(i_j) \in \{s_1(i_1), \ldots, s_k(i_k)\}$ such that $\omega s_j \in B$ follows $s_j \in V(\{s_1(i_1), \ldots, s_k(i_k)\}, \{t_1, \ldots, t_l\})$ (i.e. s_j is minimal in $V(\{s_1(i_1), \ldots, s_k(i_k)\}, \{t_1, \ldots, t_l\})$) and $card(s_j, v) = i_j$ for all maximal words $v \in V(\{s_1(i_1), \ldots, s_k(i_k)\}, \{t_1, \ldots, t_l\})$.*
- *For all $\omega \in B$ for which exists $t_i \in \{t_1, \ldots, t_l\}$ and $u, v \in \Sigma^*$ with $\omega = uvt_i$, $vt_i \in V(\{s_1(i_1), \ldots, s_k(i_k)\}, \{t_1, \ldots, t_l\})$, follows that vt_i is maximal in $V(\{s_1(i_1), \ldots, s_k(i_k)\}, \{t_1, \ldots, t_l\})$.*

In some cases we are not interested in how often each of the ending actions occurs within the phase but we want to fix the number of occurences of any of them.

Definition 5. *Let $s_1, \ldots, s_k, t_1, \ldots, t_l \in \Sigma$ be actions. Then $V(\{s_1, \ldots, s_k\}, \{t_1, \ldots, t_l\}(j)) \subseteq B$ (with $j \in \mathbb{N}$) defines a phase in B that starts with actions $s_1, \ldots, s_k$ and terminates with actions $t_1, \ldots, t_l$ in the following sense:*

- *For all $\omega \in B$ for which exists $s_j \in \{s_1, \ldots, s_k\}$ such that $\omega s_j \in B$ follows $s_j \in V(\{s_1, \ldots, s_k\}, \{t_1, \ldots, t_l\}(j))$ (i.e. s_j is minimal in $V(\{s_1, \ldots, s_k\}, \{t_1, \ldots, t_l\}(j))$).*
- *For all $\omega \in B$ for which exists $t_i \in \{t_1, \ldots, t_l\}$ and $u, v \in \Sigma^*$ with $\omega = uvt_i$, $vt_i \in V(\{s_1, \ldots, s_k\}, \{t_1, \ldots, t_l\}(j))$, and $card(\{t_1, \ldots, t_l\}, vt_i) = j$ follows that vt_i is maximal in $V(\{s_1, \ldots, s_k\}, \{t_1, \ldots, t_l\}(j))$.*

While in many cases we are able to identify the last action(s) of a phase, in some cases we may know only the first action(s) that occur outside the phase. The next definition allows to specify a phase using these actions.

Definition 6. *Let $s_1, \ldots, s_k, t_1, \ldots, t_l \in \Sigma$ be actions, and let $p_1, \ldots, p_l$ be parameters with values* in *and* ex. *Then $V(\{s_1, \ldots, s_k\}, \{t_1(j_1, p_1), \ldots, t_l(j_l, p_l)\}) \subseteq B$ (with $j_1, \ldots, j_l \in \mathbb{N}$) defines a phase in B that starts with actions $s_1, \ldots, s_k$ and terminates with actions $t_1, \ldots, t_l$ in the following sense:*

- *For all $\omega \in B$ for which exists $s_j \in \{s_1, \ldots, s_k\}$ such that $\omega s_j \in B$ follows $s_j \in V(\{s_1, \ldots, s_k\}, \{t_1(j_1, p_1), \ldots, t_l(j_l, p_l)\})$ (i.e. s_j is minimal in $V(\{s_1, \ldots, s_k\}, \{t_1(j_1, p_1), \ldots, t_l(j_l, p_l)\})$.*
- *For all $\omega \in B$ for which exists $t_i(j_i, in) \in \{t_1(j_1, p_1), \ldots, t_l(j_l, p_l)\}$ and $u, v \in \Sigma^*$ with $\omega = uvt_i$, $vt_i \in V(\{s_1, \ldots, s_k\}, \{t_1(j_1, p_1), \ldots, t_l(j_l, p_l)\})$, and $card(t_i, vt_i) = j_i$ follows that vt_i is maximal in $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$.*
- *For all $\omega \in B$ for which exists $t_i(j_i, ex) \in \{t_1(j_1, p_1), \ldots, t_l(j_l, p_l)\}$ and $u, v \in \Sigma^*$ with $\omega = uvt_i$, $v \in V(\{s_1, \ldots, s_k\}, \{t_1(j_1, p_1), \ldots, t_l(j_l, p_l)\})$, and $card(t_i, v) = j_i$ follows that v is maximal in $V(\{s_1, \ldots, s_k\}, \{t_1(j_1), \ldots, t_l(j_l)\})$.*

In other words, if the number of occurrences of a terminating action is accompanied by the parameter *in*, the action is part of the phase. If it is accompanied by *ex*, it is the first action after the phase's termination.

4.1.3 SeMF-Properties

We now introduce additional SeMF Properties in terms of predicates that were not already introduced in [13].

Definition 7. *Let V be a phase within B, the behaviour of a system S, then precede-within-phase(a, b, V) is fulfilled, iff for all $v \in V$ with $b \in alph(v)$ also $a \in alph(v)$.*

Definition 8. *Let B be the behaviour of a system S, then not-happens(a) is fulfilled, iff for all $\omega \in B$ it holds that $a \notin alph(\omega)$.*

Definition 9. *Let V be a phase within B, the behaviour of a system S, then not-happens-within-phase(a, V) is fulfilled, iff for all $v \in V$ it holds that $a \notin alph(v)$.*

Software Properties A software property is a new concept added to the set of properties in SeMF that describes certain (relevant) behavioral properties of a certain software. Formally this is interpreted such that, when a software based agent P (e.g. an ECU) is booted with a certain software (formalized by $boot(P, sw_i)$), this agent will behave according to the software property until it is shutdown. Note that for reasons of simplicity and because of the monolithic nature of the embedded firmwares used in ECUs we abstract from the case of an agent's software configuration changing during runtime after a certain software stack sw_i has been booted.

Definition 10 (Software Property)**.** *Some software sw_i has a certain property* prop *if for all $P \in \mathbb{P}$,* prop *holds in $V(\{boot(P, sw_i)\}, \{shutdown(P)\}(1))$, denoted by sw-prop$(sw_i, prop)$.*

4.1.4 F-SeBBs

This section will introduce the necessary F-SeBBs for the subsequent proof. F-SeBBs are relations among SeMF properties that are founded in their formal definitions.

Implication from not-happens to precede within a phase (SeBB.4.1.4.1)
External Property: $\forall x \in \Sigma : precede\text{-}within\text{-}phase(x, a, V)$
Internal Property: $not\text{-}happens\text{-}within\text{-}phase(a, V)$

Transitivity of Precede (SeBB.4.1.4.2)
External Property: $precede(a, c)$
Internal Property: $precede(a, b) \wedge precede(b, c)$

Transitivity of Precede within a phase (SeBB.4.1.4.3)
External Property:
$precede\text{-}within\text{-}phase(a, c, V)$
Internal Property:
$precede\text{-}within\text{-}phase(a, b, V) \wedge precede\text{-}within\text{-}phase(b, c, V)$

Unification of phase-starts for precede (SeBB.4.1.4.4)
External Property:
$precede\text{-}within\text{-}phase(a, b, V(s_1 \cup s_2, t))$
Internal Property:
$precede\text{-}within\text{-}phase(a, b, V(s_1, t)) \wedge precede\text{-}within\text{-}phase(a, b, V(s_2, t))$

Unification of phase-starts for not-happens (SeBB.4.1.4.5)
External Property:
$not\text{-}happens\text{-}within\text{-}phase(a, V(s_1 \cup s_2, t))$
Internal Property:
$not\text{-}happens\text{-}within\text{-}phase(a, V(s_1, t)) \wedge not\text{-}happens\text{-}within\text{-}phase(a, V(s_2, t))$

Chaining of phases for not-happens (SeBB.4.1.4.6)
External Property:
$not\text{-}happens\text{-}within\text{-}phase(a, V(s_1, t_2(n)))$
Internal Property:
$not\text{-}happens\text{-}within\text{-}phase(a, V(s_1, t_1(m))) \wedge not\text{-}happens\text{-}within\text{-}phase(a, V(t_1, t_2(n)))$

Implications of precede to not-happens (SeBB.4.1.4.7)
External Property:
$not\text{-}happens(b)$
Internal Property:
$not\text{-}happens(a) \wedge precede(a, b)$

Implications of not-happens to precede-within-phase's start (SeBB.4.1.4.8)
External Property: $\forall a, b \subseteq \Sigma :$ *precede-within-phase*$(a, b, V(s, t))$
Internal Property: *not-happens*(s)

Implication of not-happens to not-happens-within-phase's start (SeBB.4.1.4.9)
External Property: $\forall a \subseteq \Sigma :$ *not-happens-within-phase*$(a, V(s, t))$
Internal Property: *not-happens*(s)

Implications of phased not-happens to precede (SeBB.4.1.4.10)
External Property: *precede*(a, b)
Internal Property: *not-happens-within-phase*$(b, V(\emptyset, a(1)))$

Chaining to cyclic phases for precede (SeBB.4.1.4.11)
External Property: *precede-within-phase*$(a, b, V(s, s(2)))$
Internal Property: *precede-within-phase*$(a, b, V(s, t(1))) \wedge$ *precede-within-phase*$(a, b, V(t, s(1))) \wedge$ $s \cap t = \emptyset$

Resolution of cyclic phases to overall precede (SeBB.4.1.4.12)
External Property: *precede*(a, b)
Internal Property: *precede-within-phase*$(a, b, V(\emptyset, s(1))) \wedge$ *precede-within-phase*$(a, b, V(s, s(2)))$

Specialization of Precede's Trigger actions within a phase (SeBB.4.1.4.13)
External Property: $$precede\text{-}within\text{-}phase(a, b_1, V)$$
Internal Property: $$precede\text{-}within\text{-}phase(a, b_1 \cup b_2, V)$$

Specialization of Preceding actions using not-happens within a phase (SeBB.4.1.4.14)
External Property: $$precede\text{-}within\text{-}phase(a_1, b, V)$$
Internal Property: $$precede\text{-}within\text{-}phase(a_1 \cup a_2, b, V) \wedge not\text{-}happens\text{-}within\text{-}phase(a_2, V)$$

Specialization of Limited Precede using precede (SeBB.4.1.4.15)
External Property: $$precede(a, c)$$
Internal Property: $$limited\text{-}precede(a, b\&c) \wedge precede(b, c)$$

Specialization of Limited Precede using precede within a phase (SeBB.4.1.4.16)
External Property: $$precede\text{-}within\text{-}phase(a, c, V)$$
Internal Property: $$limited\text{-}precede\text{-}within\text{-}phase(a, b\&c, V) \wedge precede\text{-}within\text{-}phase(b, c, V)$$

Specialization of nothappens to a phase (SeBB.4.1.4.17)
External Property: $$\forall V : not\text{-}happens\text{-}within\text{-}phase(a, V)$$
Internal Property: $$not\text{-}happens(a)$$

Specialization of precede under not-happens	(SeBB.4.1.4.18)
External Property:	
$precede(a, c)$	
Internal Property:	
$precede(a \cup b, c) \wedge not\text{-}happens(b)$	

Narrowing of limited-precedewith not-happens	(SeBB.4.1.4.19)
External Property:	
$limited\text{-}precede(a\&b_1, c)$	
Internal Property:	
$limited\text{-}precede(a\&b_1 \cup b_2, c) \wedge not\text{-}happens(b_2)$	

Specialization of triggers in limited-precede	(SeBB.4.1.4.20)
External Property:	
$limited\text{-}precede(a, b_1\&c)$	
Internal Property:	
$limited\text{-}precede(a, b_1 \cup b_2\&c) \wedge not\text{-}happens(b_2)$	

True Implication	(SeBB.4.1.4.21)
External Property:	
b	
Internal Property:	
$(a \rightarrow b) \wedge b$	

Relations between not-happens and not-precede	(SeBB.4.1.4.22)
External Property:	
$not\text{-}precede(a, \{b \mid b \in \Sigma\}$	
Internal Property:	
$not\text{-}happens(a)$	

4.2 System-Model, Agents and Topology

Agents An EVITA system consists of finitely many vehicles $Vehicle_1, \ldots, Vehicle_k$ ($k \in \mathbb{N}$). Each vehicle in turn consists of a sensor ECU, an application ECU, an ECU for the communication control unit CCU, an ECU for the human-machine interface HMI, and a key master ECU:

$$Vehicle_i = \{ECU_{Sensor_i}, ECU_{Appl_i}, ECU_{CCU_i}, ECU_{HMI_i}, ECU_{KeyMaster_i}\} \ (i = 1, \ldots, k)$$

Each ECU again is composed of a CPU and an HSM, i.e. for $x \in \{Sensor, Appl, CCU, HMI, KeyMaster\}$, we have $ECU_{x_i} = \{CCU_{x_i}, HMI_{x_i}\}$ $(i = 1, \ldots, k)$.
Further, each vehicle has a driver, so $\{Driver_1, \ldots, Driver_k\} \subseteq \mathbb{P}$ are agents as well. Finally, the ECU manufacturer is an agent: $Manufacturer \in \mathbb{P}$.

For validation purposes, we will use a small example system composed of two vehicles $Vehicle_1$ and $Vehicle_2$, their drivers $Driver_1, Driver_2$, and the manufacturer, it graphical representation is depicted in Figure 24.

The Figure 24 demonstrates the topological view we assume for the example system. Regarding the actions of each of the agents, we will assume, that each of the CPUs can send and receive on to and from its topological attached neighbours. The set of actions of the HSMs consists of the actions from D3.2. And for each of these actions of the HSM, the CPU may call the HSM to perform a certain action and wil receive the corresponding results. These will be denoted by the prefixes $cmd\_$ and $ret\_$.

Further, we will assume the ECUs to communicate using the EVITA protocols from [18]. More details on this will be found in the subsequent sections.

During the formalization of properties from [19, 18] some flaws were found that needed to be corrected before attempting to model or verify the system. The following list gives a summary. More details follow in the subsequent sections:

- *Secure Storage* added at all ECU for KeyHandle-Lookups, Group-Lookups, ... Unfortunately the Protocol specifications do not mention or foresee where to store the key handles generated by the HSM during key Import. This was added to the functionality of secure storage here.

- *Consolidation of Key_Export, Key_Import and KeyDistributionProtocol* was necessary, as the HSM-Specification forsees two transportation keys for encryption and integrity, whereas the Protocol only uses one transportation key. As only the confidentiality of the transported key comes into play during the proofs, the second (integrity) transportation key is omitted for the rest of the document.

- *Distiction between senders and receivers of a Group* to allow the KeyMaster to differentiate between senders and receivers for a group during KeyDistributionProtocol. This will become an important part in the following functional proof.

4.3 Security Properties

4.3.1 General Properties

In this section we will present general properties that are used to express (usually physical) restrictions of the presented system that are however not imposed by the specific Evita architecture and protocols.

Unavailability of ECUs prior to or in between boot cycles A rather obvious property of all computer systems – and therefore of the ECUs consisting of CPU and HSM – is that they will not perform any action prior to the first boot as well as in between shutdown and boot. Please note for ease of understanding, we will define the following properties very general. However, Prop.4.3.3.3 is a contradiction to the generality of

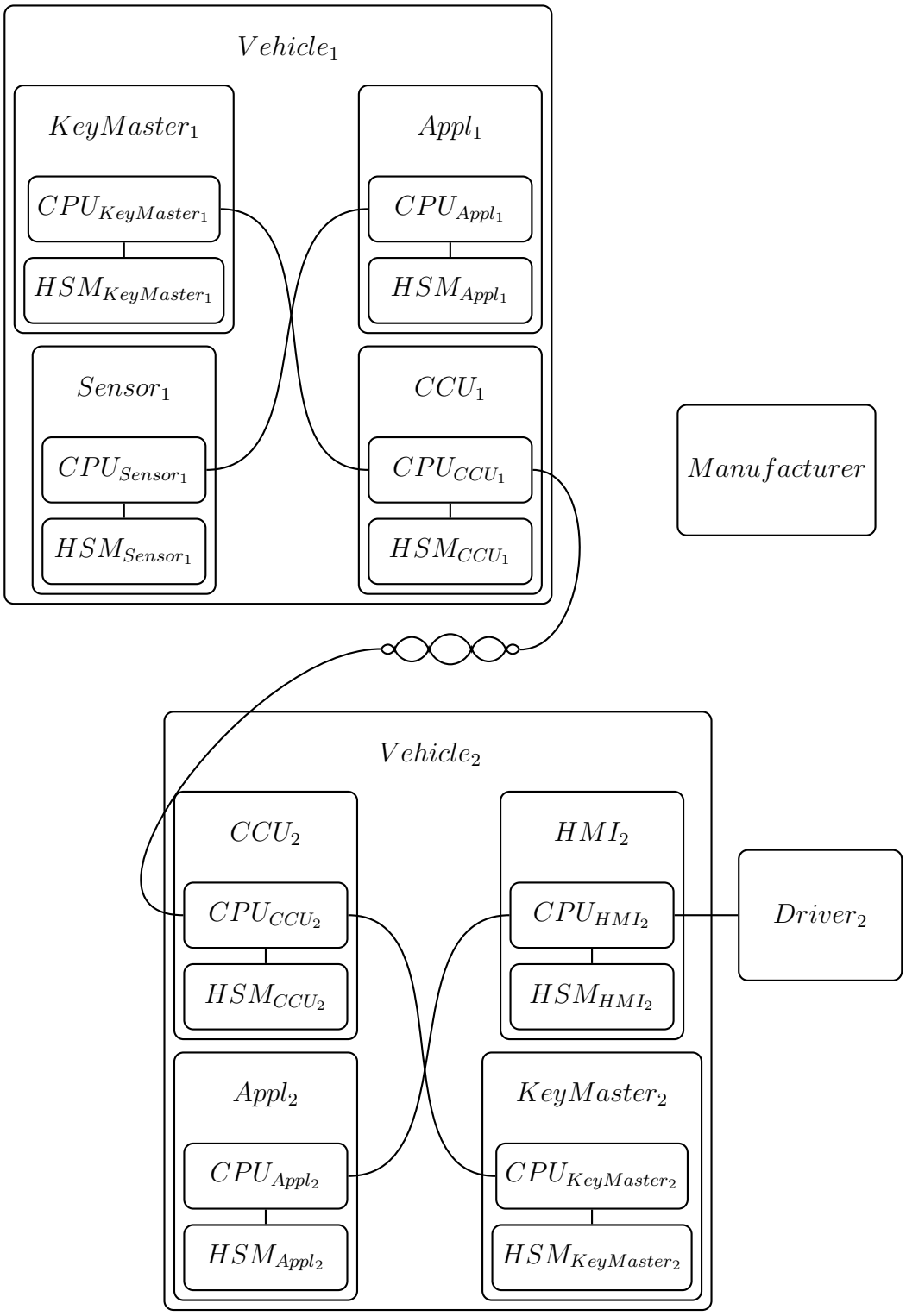


Figure 24 System model of verification target

unavailability of actions. the following definitions We therefore assume the manufacturer

to set the first reference value during production allready and therefore instantiations do not concern the $preset\_ECR$ action.

Unavailability prior to first boot: (Prop.4.3.1.1)

$$\begin{aligned} &\textit{not-happens-within-phase}(\Sigma_{/CPU_i} \cup \Sigma_{/HSM_i} \setminus \{boot(CPU_i, \ldots), \\ &\qquad preset\_ECR(HSM_i, \ldots)\}, \\ &\quad V(\emptyset, boot(CPU_i, \ldots)(1))) \end{aligned}$$

Unavailability between shutdown and boot: (Prop.4.3.1.2)

$$\begin{aligned} &\textit{not-happens-within-phase}(\Sigma_{/CPU_i} \cup \Sigma_{/HSM_i} \setminus \{shutdown(CPU_i, \ldots), boot(CPU_i, \ldots)\}, \\ &\quad V(shutdown(CPU_i), boot(CPU_i, \ldots)(1))) \end{aligned}$$

Key Usage For the following definitions, recall that the predicate $conf(\mathcal{A}(p), p, X)$ denotes that only the agents $P \in X$ ($X \subseteq \mathbb{P}$) are allowed to know the parameter p, and that $\mathcal{A}(p)$ denotes those actions that allow an agent monitoring the system to possibly gain knowledge about p.

The usage of a key by an agent $P \in \mathbb{P}$ implies that (besides a not further defined group of agents $X \subseteq \mathbb{P}$ allowed to know the key), P itself knows the key. Accordingly, if an agent does not know a key, it can not use it. (This applies to systems as well as phases within a system.):

Confidentiality restricts usage: (Prop.4.3.1.3)

$$\forall P \in \mathbb{P} : conf(\mathcal{A}(k), k, .., \mathbb{X} \setminus \{P\}) \rightarrow \textit{not-happens}(\mathcal{A}_{/P}(K))$$

If an action a does not involve a key within its parameters, hence does not add to agents' knowledge about this key's value, the concatenation of this action a to any sequence of actions will not extend the knowledge about k of any agent in the system. This assumption requires the system to be formalized in an appropriate way, in particular actions adding to agents' knowledge about parameters to be confidential must contain these parameters. Of course this does not imply that agents necessarily learn from actions involving this parameter. This has to be decided for each specific system separately.

For the following definition, recall that the homomorphism μ keeps exactly those actions that can add to agents' knowledge about the parameter to be confidential. Here μ_k denotes the homomorphism keeping all actions that add to knowledge about the key k.

Only actions involving a Key parameter add to agents' knowledge about the Key's value: (Prop.4.3.1.4)

$$\forall \omega \in B, \forall R \in \mathbb{P}, \forall a \in \Sigma \setminus \mathcal{A}(k) : \mu_k(\lambda_R^{-1}(\lambda_R(\omega)) \cap W_R) = \mu_k(\lambda_R^{-1}(\lambda_R(\omega a)) \cap W_R)$$

KeyHandle Usage When a key is generated or imported by an HSM, it may be securely stored in its non-volatile memory or directly used and stored in cache for further use within the current session. We regard both the non-volatile storage and the cache as secure storage, hence the following property holds:

Key Handles originate from secure storage: (Prop.4.3.1.5)

$$\textit{precede-within-phase}(SecStorRead(CPU_i, StorID, Handle), cmd\_*(CPU_i, Handle, \ldots), V(boot(CPU_i, sw_x, \ldots), shutdown(CPU_i)(1)))$$

4.3.2 Evita Software Properties

This section presents requirements against secure software on Evita architectures. These requirements are necessary for the verification of the system's design. As such they can be considered input for the Tasks T4200 – the basic driver – as well as T4400 – the verification of the secure software.

As the Evita specification so far only considers the ECU on-chip design, a functionally not further defined middleware and the on-board protocols, the requirements specified in this section are results of our formal validation process.

Synchronized HSM-Access from Software The connection between the CPU and the HSM is asynchronous and non-blocking. Further it does not include the notion of command-return sessions or associations – neither cryptographically secured nor non-secured. This leads to possible race-conditions during concurrent access to the HSM which can be security relevant. For example during parallel Key_Import at the KeyMaster within the Key-Distribution Protocol, a malicious ECU might smuggle its own key into a different group. Accordingly, every software in Evita is required to synchronize the access to the HSM at all times, formalized as:

Synchronized Access to ECU: (Prop.4.3.2.1)

$$\textit{not-happens-within-phase}(cmd\_*(CPU_i, \ldots), V(cmd\_*(CPU_i, \ldots)(ex), ret\_*(CPU_i, \ldots)(1, in)))$$

Non-Malicious Sensor Behaviour

Operational Mode Whenever the sensor's software commands the HSM to produce a signature using the key that was stored for the purpose of signing sensing data, the data being signed must have been sensed before:

Sign sensed data with sensing key: (Prop.4.3.2.2)

$$\textit{sw-prop}(sw_{Sensor}, \textit{limited-precede}(sense(P, data_{Sensor}), SecStorRead(P, \text{“}StorID_{Sensor}\text{”}, Handle)\& cmd\_Hash\_mac\_sign(P, data_{Sensor}, Handle))$$

This property can be specialized to the case of CPU_{Sensor_1}:

Sensor signs sensed data with sensing key: (Prop.4.3.2.3)

$$\begin{aligned}&\textit{sw-prop}(sw_{Sensor}, \textit{limited-precede}(\textit{sense}(CPU_{Sensor_1}, data_{Sensor}),\\ &\qquad SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle)\ \&\\ &\qquad cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, data_{Sensor}, Handle))\end{aligned}$$

Whenever the sensor's software commands the HSM to sign a message containing of a keyExport-Blob together with the group identifier for the group of sensors with the pre-shared key with the keymaster, this exported key inside this blob must be the key generated for signing of sensor data.

Distribution of $Handle_{SessK_{Sensor}}$ under the label of “Gid_{Sensor}”: (Prop.4.3.2.4)

$$\begin{aligned}&\textit{sw-prop}(sw_{Sensor},\textit{limited-precede}(SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle_a)\\ &\quad \&\ ret\_Key\_Export(CPU_{Sensor_1}, exportBlob(K_c),\\ &\qquad cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, \ldots)),\\ &\quad SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{PSK}\text{”}, Handle_b)\\ &\quad \&\ cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, m(exportBlob(K_c), \text{“}Gid_{Sensor}\text{”}),\\ &\qquad Handle_b)))\end{aligned}$$

This is part of the necessary constraints within the key distribution protocol, that ensures that the key being distributed under the label of the sensor-data-signing-key is actually the sensor-data-signing key and no other.

Key Distribution It is clear, that a Sensor will attempt to exchange a shared session key with the application cpus that come afterwards. This key (even though not further specified) will be called $SessK_{Sensor}$ and the corresponding key handle $Handle_{SessK_{Sensor}}$ in the following. For this special key, several constraints against the behavior of the sensor exist.

First of all, the sensor may only use such a key, that was created locally by the sensor's HSM.

“$StorID_{Sensor}$”'s key created locally: (Prop.4.3.2.5)

$$\begin{aligned}&\textit{sw-prop}(sw_{Sensor}, \textit{precede}(ret\_create\_Random\_Key(P, Handle_{SessK_{Sensor}}, \ldots),\\ &\qquad SecStorWrite(P, \text{“}StorID_{Sensor}\text{”}, Handle_{SessK_{Sensor}})))\end{aligned}$$

Further this session key may only be stored in the secure storage under the label “$StorID_{Sensor}$”. This is important as we will see in Section 4.4.1 because messages of the transport protocol are not further typed and the key therefore acts as typisation authenticator as well.

Only $Handle_{SessK_{Sensor}}$ is written to “$StorID_{Sensor}$”: (Prop.4.3.2.6)

$$\textit{sw-prop}(sw_{Sensor}, \textit{not-happens}(\{SecStorWrite(P, StorID, Handle_{SessK_{Sensor}}) \mid StorID \neq \text{“}StorID_{Sensor}\text{”}\}))$$

Finally, of course the Sensor may not store any other key handle than the one of the shared secret under the corresponding storage identifier. Not that a storage identifier may be anything from a memory address in the Non-Volotile Memory, over a filename to a database entry. This solely depends on the implementation.

$Handle_{SessK_{Sensor}}$ *is written only to* “$StorID_{Sensor}$”: (Prop.4.3.2.7)

$$\textit{sw-prop}(sw_{Sensor}, \textit{not-happens}(\{SecStorWrite(P, \text{“}StorID_{Sensor}\text{”}, Handle) \mid Handle \neq Handle_{SessK_{Sensor}}\}))$$

Whenever the sensor's software commands the HSM to export a key that was stored for the purpose of signing sensing data, the transport key used for the encrypted export must originate from the storage designated to the key PSK that the HSMpre-shares with the key master:

Only export SessK to keyMaster: (Prop.4.3.2.8)

$$\textit{sw-prop}(sw_{Sensor}, \textit{limited-precede}(SecStorRead(P, \text{“}StorID_{PSK}\text{”}, Handle_b), SecStorRead(P, \text{“}StorID_{Sensor}\text{”}, Handle_a) \& cmd\_Key\_Export(P, Handle_a, Handle_b)))$$

Finally it is very important for the Sensor's software to set the correct transportable useflags. This way other HSMs are bound to using the SessK only for verification and not for signing:

Creation-Command of SessK with validation only transportation: (Prop.4.3.2.9)

$$\textit{sw-prop}(sw_{Sensor}, \textit{not-happens}(\{cmd\_create\_Random\_Key(CPU_{Sensor_1}, useflags) \mid sign_{transp} \in useflags\})$$

Non-Malicious KeyMaster Behaviour This paragraph includes the properties required from a trustworthy software for the keymaster $sw_{KeyMaster}$.

KeyDistribution The keymaster's software must value the keyhandles stored in secure storage for the redistribution of keys.

Keymaster Gid compliance: (Prop.4.3.2.10)

$$\begin{aligned}\textit{sw-prop}(sw_{KeyMaster}, \textit{limited-precede}(&SecStorRead(P, \text{“}Gid_{Sensor}\text{-}Recvr\text{”}, Handle_a),\\ &SecStorRead(P, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_b)\\ \&\ &ret\_Key\_Import(P, Handle_c, cmd\_Key\_Import(\\ &P, exportBlob, Handle_b))\\ \&\ &cmd\_Key\_Export(P, Handle_c, Handle_a)))\end{aligned}$$

The keymaster's software must check the MACs of importblobs before continuing with the key distribution in terms of importing the key.

Keymaster MAC check required: (Prop.4.3.2.11)

$$\begin{aligned}
&sw\text{-}prop(sw_{KeyMaster}, limited\text{-}precede(\\
&\quad SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender'', Handle_a)\\
&\quad \& \; ret_Hash_mac_verify(CPU_{KeyMaster_1}, cmd_Hash_mac_verify(\\
&\quad\quad CPU_{KeyMaster_1}, m(exportBlob, \text{“}Gid_{Sensor}''), Handle_a)),\\
&\quad SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender'', Handle_b)\\
&\quad \& \; cmd_Key_Import(CPU_{KeyMaster_1}, exportBlob, Handle_b)))
\end{aligned}$$

Further Agents Similar properties can be found for softwares of all the subsequent agent in the functional path. However when it comes to the processing of the data for example, more specifics from the functional domain must be taken into account, such as the set of measurement values that actually make a warning necessary. This input is processed within the security proof.

4.3.3 Evita Hardware Properties

[19] uses the assumption that the HSM is "trustworthy". This means that the HSM has certain properties related to its internal behaviour but also related to the connection between the Application CPU and the protected and separated HSM. This section will list and formalize those properties of the HSM that are relevant for the validation of the properties and use cases provided in this document.

Secure Boot [19] provides the necessary HSM commands in order to perform a secure boot functionality. However an actual secure boot depends as much on the HSM commands as on the behaviour of the pre-bios / boot-rom code. Therefore we will refine the secure boot functionality further. This is input for WP5000, the design of the Evita prototype, and more specifically for any attempt to deploy the Evita architecture into a final on-chip solution.

The Evita specification requires a secure boot process to be in place that ensures the trustworthiness of the respective platform in the following way: Any program code for the platform that shall be flashed is accompanied by a reference value. Successful validation against this reference value is required before every boot of the software. There are however various different ways in (i) what part of the software is to be verified and (ii) how to react in cases of a missmatch, that are not further specified within D3.2.

Possible solutions to (i) include the verification of the complete firmware image, i.e. the wholeness of executable code against the reference value in the very first step, or the verification of a firmware loader that would validate the loaded firmware by means of certificates and a public key carried within the firmware loader. Also each of the components within the boot chain may validate the subsequent loaded software component by means of certificates and public keys. Targeting (ii) leaves the question open of whether

the boot / software load process would interrupt in case of a missmatch of validations or whether this would only be marked e.g. within one of the ECR registers.

As this decision is to be made by an actual implementation and deployment in WP4000 / WP5000, no concrete properties can be deduced for the moment. However for the validation attempt in this document, we will assume that a software component will only be loaded in case of successful validation and that the ECU will shutdown momentarily otherwise. This decision regarding (ii) has to be made as it is an important assumption for the correctness of the system. Regarding the question (i) of how the validation is performed, we will abstract from this concreteness and assume that once the system is running, its components can be traced to the reference ECR value stored within the HSM. As a result, whenever a software is booted on an ECU whose ecr values do not match the ones set as ECR reference $ECR\text{-}ref$, no further actions are performed by the ECU except finishing this boot action and subsequently performing the shutdown:

Secure Boot: (Prop.4.3.3.1)

$$\begin{aligned} &\textit{not-happens-within-phase}(\Sigma_{/CPU_i} \cup \Sigma_{/HSM_i} \setminus \{boot(CPU_i, \ldots), shutdown(CPU_i, \ldots)\}, \\ &\qquad V(\{boot(CPU_i, sw_x, ECR\text{-}ref) | ecr(sw_x) \neq ECR\text{-}ref\}, \\ &\qquad\quad shutdown(CPU_i)(1))) \end{aligned}$$

Further, to investigate the validity of the reference ECR, the action of setting a reference ECR is to be secured. This will be further investigated in Section 4.4's paragraph on the Secure Firmware Update Protocols.

D3.2 foresees that a preset_ECR command is used to set the reference value that will be checked during the following boot cycle. The corresponding formal property is:

Boot-Reference-Setting: (Prop.4.3.3.2)

$$\begin{aligned} &\textit{precede-within-phase}(preset\_ECR(HSM_i, ECR\text{-}ref_i, \ldots), boot(CPU_i, sw_x, ECR\text{-}ref_i), \\ &\qquad V(preset\_ECR(HSM_i, ECR\text{-}ref_y, \ldots)(1), boot(CPU_i, sw_x, ECR\text{-}ref_i))) \end{aligned}$$

An important requirement that needs to be satisfied by the Evita system is that the first setting of PCR reference values is performed in a secure environment. Thus for reasons of simplicity we will assume the first *preset_ECR* command that sets these values to happen during production even before the first *boot* happens. Although this may neglect the actual sequence of actions during production, it appropriately represents the usage for our deployment scenario that ignores disregards the production phase.

1st Boot has Reference: (Prop.4.3.3.3)

$$\begin{aligned} &\textit{not-happens-within-phase}(boot(CPU_i, \ldots), \\ &\qquad V(\emptyset, preset\_ECR(HSM_i, \ldots))) \end{aligned}$$

Applying SeBB.4.1.4.10 on Prop.4.3.3.3 implies

1st Boot has Reference II: (Prop.4.3.3.4)

$$precede(preset\_ECR(HSM_i, \ldots), boot(CPU_i, \ldots))$$

Prop.4.3.3.4 can be combined with Prop.4.3.3.2 to conclude:

Every Boot has Reference: (Prop.4.3.3.5)

$$precede(preset\_ECR(HSM_i, ECR\text{-}ref_i, \ldots), boot(CPU_i, sw_x, ECR\text{-}ref_i))$$

KeyUsage Restrictions Another approach to ensure the software integrity of an ECU,known from the TPM specification, is to restrict the usage of a key to a predefined set of ECR values that represent non-malicious versions of the software. In consequence the HSM will not use this key if the actual ECR values do not match the ones assigned to the key, i.e. if the actual running software corresponds to different ECR values, hence must be assumed to be malicious. This property can be formalized as:

KeyUsage-Restriction: (Prop.4.3.3.6)

$$\begin{aligned} &not\text{-}happens\text{-}within\text{-}phase(\{Hash\_mac\_sign(HSM_i, SS(ECR\text{-}ref), \ldots) \\ &\qquad |ECR\text{-}ref \neq ecr(sw_x)\}, \\ &\qquad V(boot(CPU_i, sw_x), shutdown(CPU_i)(1))) \end{aligned}$$

However the ECR values the key is bound to are set during its creation. Similar to other key information such as the integrity of useflags, these ECR values must not be altered from key creation to usage:

KeyUsage-Reference-Setting: (Prop.4.3.3.7)

$$\begin{aligned} &precede(create\_Random\_Key(HSM_i, ECR\text{-}ref, SS(ECR\text{-}ref), \ldots), \\ &\qquad Hash\_mac\_sign(HSM_i, SS(ECR\text{-}ref), \ldots)) \end{aligned}$$

KeyUsage SecureBoot Hybrid It is also be possible to restrict the usage of the HSM (except for the extendECR and compareECR functions) to a non-malicious booted system.

KeyUsage-SecureBoot-Restriction: (Prop.4.3.3.8)

$$\begin{aligned} &not\text{-}happens\text{-}within\text{-}phase(\Sigma_{/HSM_i} \setminus \{extendECR(HSM_i, \ldots), compareECR(HSM_i, \ldots)\}, \\ &\qquad V(\{boot(CPU_i, sw_x, ECR\text{-}ref) | ecr(sw_x) \neq ECR\text{-}ref\}, \\ &\qquad\quad shutdown(CPU_i)(1))) \end{aligned}$$

Key Handling Regarding the handling and generation of keys, D3.2 makes several assumptions. First, it includes Random Number Generators in every HSM that can generate unpredictable and unguessable keys. This property may be formalized as:

Unpredictable / unobservable Key Generation: (Prop.4.3.3.9)

$$conf\text{-}within\text{-}phase(\{create\_Random\_Key(HSM_i, K_j, \ldots)\}, K_j, \ldots, \{HSM_i\}, V(\emptyset, create\_Random\_Key(HSM_i, K_j, \ldots)\}))$$

Further, the HSM is supposed to be constructed in such a way that it is impossible to retrieve a key from it except by the use of the command Key_Export. (Note that we omit the command for creation of Diffie-Helman keys as they are not used within our example use case. They may however pose another way to make a key visible to the counter-part.) This property can be formalized as:

Non-disclosure of Keys (execpt for Key_Export): (Prop.4.3.3.10)

$$\forall X \subseteq \mathbb{P}, \forall \mathbb{A} \subseteq \mathcal{A}_{/HSM_i}(K_j) \setminus Key\_Export(HSM_i, K_j, \ldots) : conf\text{-}within\text{-}phase(\mathcal{A}(K_j), K_j, \ldots, \mathbb{X}, V(\emptyset, \mathbb{A}(n, ex)) \rightarrow conf\text{-}within\text{-}phase(\mathcal{A}(K_j), K_j, \ldots, \mathbb{X}, V(\emptyset, \mathbb{A}(n, in)))$$

Unobservability of Keys: (Prop.4.3.3.11)

$$\forall \mathbb{X} \subseteq \mathbb{P}, \mathbb{A} \subseteq \mathcal{A}_{/\mathbb{P}\setminus\mathbb{X}}(K_j) : conf\text{-}within\text{-}phase(\mathcal{A}(K_j), K_j, \ldots, \mathbb{X}, V(\emptyset, \mathbb{A}(n, ex))) \rightarrow conf\text{-}within\text{-}phase(\mathcal{A}(K_j), K_j, \ldots, \mathbb{X}, V(\emptyset, \mathbb{A}(n, in)))$$

Flag Handling As a result of preliminary investigations, the HSM was extended by a concept of useflags. These flags that are part of a key's public information describe for which purposes an HSM may use a key, i.e. that a shared secret may be used for MAC generation and verification or for verification purposes only. In the following we describe the formalization of the properties that originate from these use flags.

An HSM will never perform a MAC generation, if the key does not include a sign use flag:

HSMs respect sign-useflag: (Prop.4.3.3.12)

$$not\text{-}happens(\bigcup_{\forall HSM_i \in \mathbb{P}_{/HSM}} \{Hash\_mac\_sign(HSM_i, SS(useflags \setminus \{sign\}), \ldots)\}$$

An HSM will not alter the use flags of a key. So the use flags of a key will always stay the same as they were during creation or import of the key:

HSMs do not change useflags: (Prop.4.3.3.13)

$$precede(\{create\_Random\_Key(HSM, SS(useflags), \ldots),\\ Key\_Import(HSM, SS(useflags), \ldots\},\\ \mathcal{A}_{/HSM}(SS(useflags)))$$

An HSM will only export those use flags that were marked as transportable. This holds in particular for sign flags

Only transport-flagged sign-useflags are ex-/imported: (Prop.4.3.3.14)

$$precede(Key\_Export(HSM_i, SS(useflags \cup \{sign_{transp}\}), \ldots),\\ Key\_Import(HSM_j, SS(useflags \cup \{sign\}), \ldots))$$

as well as for verify flags

Only transport-flagged verify-useflags are ex-/imported: (Prop.4.3.3.15)

$$precede(Key\_Export(HSM_i, SS(useflags \cup \{verify_{transp}\}), \ldots),\\ Key\_Import(HSM_j, SS(useflags \cup \{verify\}), \ldots))$$

HSM-CPU Connection
The HSM and the Application CPU of an Evita ECU are directly connected in such a way that messages between those two cannot be intercepted, replayed, altered or spoofed. Therefore an action performed by an HSM will always refer to the command most recently received from the Application CPU within the same boot-cycle and the Application CPU will allways receive the result of the action most recently performed by the HSM within the same boot-cycle. These properties can be formalized as follows:

Commands originate from same ECU in bootcycle: (Prop.4.3.3.16)

$$\forall action \in \Sigma_{/HSM_i} :\\ precede\text{-}within\text{-}phase(cmd\_action(CPU_i, \ldots), action(HSM_i, \ldots),\\ V(boot(CPU_i, \ldots), boot(CPU_i, \ldots)(2)))$$

Returns originate from same ECU in bootcycle: (Prop.4.3.3.17)

$$\forall action \in \Sigma_{/HSM_i} :\\ precede\text{-}within\text{-}phase(action(HSM_i, \ldots), ret\_action(CPU_i, \ldots),\\ V(boot(CPU_i, \ldots), boot(CPU_i, \ldots)(2)))$$

cmd_create_Random_Key: (Prop.4.3.3.18)

$$\begin{aligned}&\textit{precede-within-phase}(cmd\_create\_Random\_Key(CPU_i, useflags),\\ &\qquad create\_Random\_Key(HSM_i, K(useflags), \ldots),\\ &\quad V(cmd\_*(CPU_i, \ldots)(1, in),\\ &\qquad create\_Random\_Key(HSM_i, K(useflags), \ldots)))\end{aligned}$$

ret_create_Random_Key: (Prop.4.3.3.19)

$$\begin{aligned}&\textit{precede-within-phase}(create\_Random\_Key(HSM_i, \ldots, Handle),\\ &\qquad ret\_create\_Random\_Key(CPU_i, Handle),\\ &\quad V(\Sigma_{/HSM_i}(1, in),\\ &\qquad ret\_create\_Random\_Key(CPU_i, Handle)))\end{aligned}$$

cmd_Key_Export: (Prop.4.3.3.20)

$$\begin{aligned}&\textit{precede-within-phase}(\\ &\quad cmd\_Key\_Export(CPU_i, Handle_j, useflags_k, Handle_l),\\ &\qquad Key\_Export(HSM_i, Handle_j, useflags_k, Handle_l, enc(K_l, K_j(useflags_k))),\\ &\qquad\quad V(cmd\_*(CPU_i, \ldots)(1, in),\\ &\qquad Key\_Export(HSM_i, Handle_j, useflags_k, Handle_l, enc(K_l, K_j(useflags_k)))))\end{aligned}$$

ret_Key_Export: (Prop.4.3.3.21)

$$\begin{aligned}&\textit{precede-within-phase}(\\ &\quad Key\_Export(HSM_i, Handle_j, useflags_k, Handle_l, enc(K_l, K_j(useflags_k))),\\ &\quad ret\_Key\_Export(CPU_i, enc(K_l, K_j(useflags_k))),\\ &\quad V(\Sigma_{/HSM_i}(1, in), ret\_Key\_Export(CPU_i, enc(K_l, K_j(useflags_k)))))\end{aligned}$$

cmd_Key_Import: (Prop.4.3.3.22)

$$\begin{aligned}&\textit{precede-within-phase}(\\ &\quad cmd\_Key\_Import(CPU_i, enc(K_l, K_j(useflags_k)), Handle_l),\\ &\qquad Key\_Import(HSM_i, enc(K_l, K_j(useflags_k)), Handle_l, K_l, K_j(useflags_k), Handle_j),\\ &\qquad\quad V(cmd\_*(CPU_i, \ldots),\\ &\qquad Key\_Import(HSM_i, enc(K_l, K_j(useflags_k)), Handle_l, K_l, K_j(useflags_k), Handle_j)))\end{aligned}$$

ret_Key_Import: (Prop.4.3.3.23)

precede-within-phase(

$Key\_Import(HSM_i, enc(K_l, K_j(useflags_k)), Handle_l, K_l, K_j(useflags_k), Handle_j)$,

$ret\_Key\_Import(CPU_i, Handle_j)$,

$V(\Sigma_{/HSM_i}(1, in), ret\_Key\_Import(CPU_i, Handle_j)))$

cmd_Hash_mac_sign: (Prop.4.3.3.24)

precede-within-phase(

$cmd\_Hash\_mac\_sign(CPU_i, data, Handle_l)$,

$Hash\_mac\_sign(HSM_i, data, Handle_l, K_l, mac(K_l, data))$,

$V(cmd\_*(CPU_i, \ldots)$,

$Hash\_mac\_sign(HSM_i, data, Handle_l, K_l, mac(K_l, data))))$

ret_Hash_mac_sign: (Prop.4.3.3.25)

precede-within-phase(

$Hash\_mac\_sign(HSM_i, data, Handle_l, K_l, mac(K_l, data))$,

$ret\_Hash\_mac\_sign(CPU_i, mac(K_l, data))$,

$V(\Sigma_{/HSM_i}(1, in), ret\_Hash\_mac\_sign(CPU_i, mac(K_l, data))))$

cmd_Hash_mac_verify: (Prop.4.3.3.26)

precede-within-phase(

$cmd\_Hash\_mac\_verify(CPU_i, data, mac(K_l, data), Handle_l)$,

$Hash\_mac\_verify(HSM_i, data, Handle_l, K_l, mac(K_l, data))$,

$V(cmd\_*(CPU_i, \ldots)$,

$Hash\_mac\_verify(HSM_i, data, Handle_l, K_l, mac(K_l, data))))$

ret_Hash_mac_verify: (Prop.4.3.3.27)

precede-within-phase(

$Hash\_mac\_verify(HSM_i, data, Handle_l, K_l, mac(K_l, data))$,

$ret\_Hash\_mac\_verify(CPU_i, mac(K_l, data))$,

$V(\Sigma_{/HSM_i}(1, in), ret\_Hash\_mac\_verify(CPU_i)))$

4.4 Protocols

4.4.1 Transport Protocol

The transport protocol allows for the confidential, authentic and timestamped transmission of data from one node within a vehicle to the next. In order to achieve this, it implements encryption and an HMAC algorithm that optionally includes a time-field. We can encapsulate the property of the HMAC as the following SeBB that was already presented in [13].

Transport Protocol's HMAC Precede Property (SeBB.4.4.1.1)
External Property: $$precede(\{Hash\_mac\_sign(P, m, SS, \ldots) \mid P \in \mathbb{X}\}, Hash\_mac\_verify(HSM_j, m, SS, \ldots))$$
Internal Property: $$conf(\mathcal{A}(SS), SS, \ldots, \mathbb{X})$$

When it comes to HMACs and Signature in general, however, there is an important issue to consider that is often neglected unintentionally in formal verification. This issue is concerned with the typing of the message that is being sent – or more practically is about the following question: Is there a tag identifying the type of message inside the signed payload.

Typing ouside of Secured Payload If the typing information, i.e. the protocol identifier, is not included in the signed payload, it is *only* the data sent to the protocol handler that is signed. For the example of the Evita deployment in this document, this would essentially look like:

$$send(P, (\text{“}sensordata'', (data)_{K_{Sensor}}))$$

with $(data)_K$ denoting data signed using key K.

The problem with this approach is that if the key is used to sign several different types of data, an attacker can use one signed piece of information and pretend it to be a different one. For example, two messages by the same (multi purpose-) sensor such as:

$$send(P, (\text{“}wheelspeed'', (int\ speed)_{K_{Sensor}}))$$

and

$$send(P, (\text{“}temperature'', (int\ temp)_{K_{Sensor}}))$$

can be merged in such a way that the attacker may inject a message of the form:

$$send(P, (\text{“}wheelspeed'', (int\ temp)_{K_{Sensor}}))$$

In a practical case, the attacker could be able to inject messages that report the wheels to turn with 20 rotations per minute, whilst in reality the wheels do not turn at all but the outside temperature is 20 degrees. Such attacks can be very harmful in cases of a vehicle standing at the traffic lights as well as a vehicle's ABS brake system.

In such a case, the key used for the signature must not only identify the sender of a message but also the type of data within the message, and the typing tag alongside serves for unsecured signalling purposes only and should not be considered reliable. The problem

becomes even more important in cases of stream-channels over a network, as there does not exist a length-field which leads to security vulnerabilities e.g. in the Needham-Schroeder protocol.

Typing inside of Secured Payload The attack described in the previous paragraph is not possible if the data is structured in such a way that a typing tag is provided alongside the payload within the signature. This could look as follows:

$$send(P, (\text{“}sensordata'', data)_{K_{Sensor}})$$

In this application of signatures or MACs, the key only serves as reliable identifier of the sender of a message, but is not needed for typing of the message's content. This approache has been taken in the specification of the Trusted Platform Module by the Trusted Computing Group. Here each TPM_Quote signature includes the character array char[]"QUOT" whilst each TPM_Sign signature includes the character array char[]"SIGN".

Implications for Evita As the Transport Protocol of Evita performes typing by using the Message Identifier field outside of the secured payload, the typing must be implicitely enforced through the used key. Note however that this does not restrict the usage of variable typing within one *data*'s data-type, such as

```
data := {OBJ_TYPE tag; OBJ_DATA object;}
```

as long as this data-type itself is fixed.

Accordingly we must assume for our proofs that for each type of *data* being transmitted a different MAC key is used, even if the group of recipients is the same, in order to prohibit the above mentioned attack. This is input for the Tasks of WP5000 of Evita where a prototype as well as any deployment of the Evita protocols will be developed.

4.4.2 Key Distribution Protocols

The primary purpose of the key distribution protocols is to provide a confidential shared secret for encryption/decryption and MACs between a set of Evita HSMs, e.g.:

$conf(\mathcal{A}(SS_{1,2}), SS_{1,2}, \ldots, \{HSM_1, HSM_2\})$

Important to note in this context is that this includes (i) the confidentiality to agents from the group of HSMs $\mathbb{P}_{/HSM}$ and the confidentiality to the HSMs corresponding to ECUs 1 and 2 in the example. The identification of which ECUs exactly that is becomes very important considering that the compromise of a single ECU shall not infect the whole system of ECUs within a vehicle.

A more in-depth investigation of the key distribution protocols however reveal additional requirements, namely the integrity of use flags associated with the shared keys. Not every member of a group of ECUs shall be able to use a key for generation of MACs. Most members only are allowed to use the key for MAC validation purposes. The Evita HSM provides the concept of use flags and use transportations flags for this as a result of the investigations. In the following we will formalize the properties provided by of one of the key distribution protocols in more depth.

Keying using asymmetric encryption In this document we will focus on the analysis of the symmetric key distribution protocol as introduced in [18]. A symmetric approach will most likely be used in every final deployment of Evita as the price for asymmetric cryptography in every ECU (esp. small sensors and actuators) is too high to afford. Also the symmetric approach is the more complex scenario, hence the more challenging to investigate.

Keying for group communication: symmetric approach This protocol uses the KeyMaster (KM) as the central point of key distribution. An ECU_1 that wants to distribute a session key SS to a group of other ECUs $\{ECU_2, \ldots, ECU_k\}$ creates a new session key SS with the useflag $\{sign, verify\} \in useflags$. It encrypts the session key with its own key PSK_1 pre-shared with KM and sends it to KM. KM decrypts with PSK_1 and encrypts again, using each of the pre-shared keys PSK_i it shares with the other ECUs. These are the essential steps of the protocol. For more information we refer the reader to [18]

The Key Distribution Symmetric Protocol thereby serves for two aspects of key handling. First, it ensures the confideniality of the Shared Secret to only those ECUs who are in the targeted group. Second, a more detailed analysis reveals the more complex details of this process. First, the HSM will only export those use-flags that were marked as transportable during creation – note that this involves 3 different export targets that we do not further investigate for now. As the key-creating ECU is the only ECU that will be sender in this group, it would be recommendable to create session keys that have only the verify-useflag marked as transportable $\{verify_{transp}\} \in useflags, \{sign_{transp}\} \notin useflags$. Therefore as long as a key stays confidential for trustworthy Evita HSMs, those flags will be respected and therefore no actions involving a sign-flag performed.

Key Distribution Symmetric: Authenticity of UseFlags (SeBB.4.4.2.1)
External Property:
$not\text{-}happens(\{\mathcal{A}_{/HSM_j}(SS(useflags \cup \{sign\}), \ldots) \mid HSM_j \in (\mathbb{P}_{/HSM} \setminus \{HSM_i\})\})$
Internal Property:
$not\text{-}happens(create\_Random\_Key(HSM_i, SS(useflags \cup \{sign_{transp}\}), \ldots))$ $\wedge\, conf(\mathcal{A}(SS), SS, \ldots, \mathbb{P}_{/HSM})$

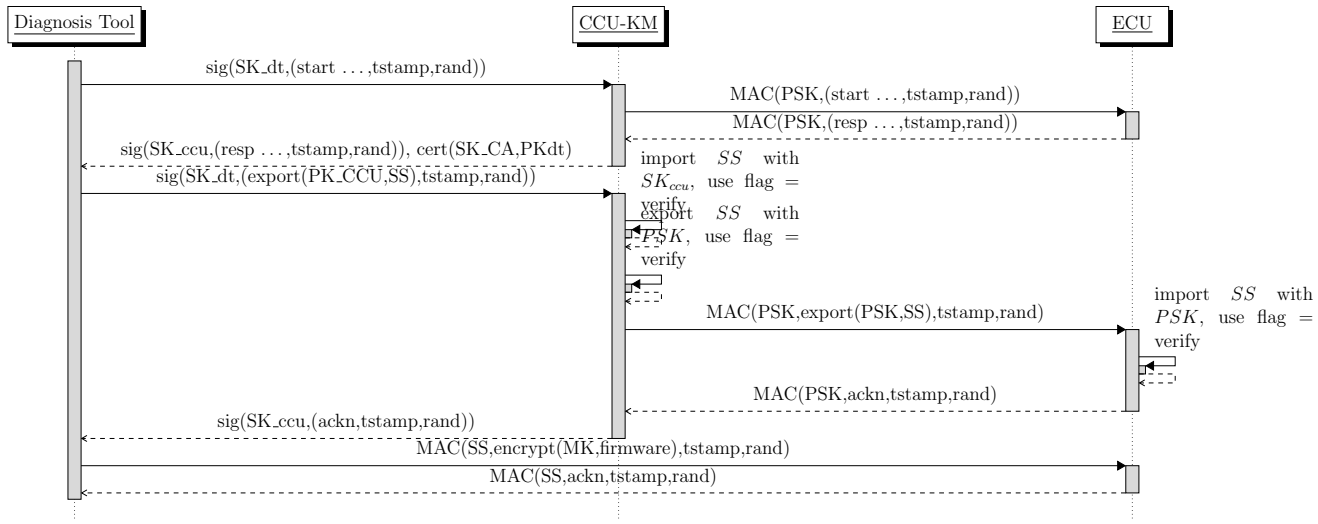
Regarding the confidentiality, the important action within the symmetric key distribution protocol is the export of a key from one HSM using a certain transport key. When this action is performed, the owner of the transport key gains (potential) knowledge of the shared secret that is being exported. The corresponding use flag may therefore be

formalized as follows:

Key distribution symmetric: Confidentiality of SS on KeyExport (SeBB.4.4.2.2)
External Property: $$conf\text{-}within\text{-}phase(\mathcal{A}(SS), SS, \ldots, \mathbb{X} \cup \mathbb{Y}, V(\emptyset, Key\_Export(P, SS, PSK, \ldots)(n, in)))$$
Internal Property: $$conf\text{-}within\text{-}phase(\mathcal{A}(SS), SS, \ldots, \mathbb{X}, V(\emptyset, Key\_Export(P, SS, PSK, \ldots)(n, ex))) \wedge conf(\mathcal{A}(PSK), PSK, \ldots, \mathbb{X} \cup \mathbb{Y})$$

4.4.3 Secure Firmware Update Protocols

The protocol is performed by four parties: OMV, the manufacturer of the new firmware, a Diagnosis Tool, a CCU that seems to be the Key Master of the ECU that shall be flashed, and the ECU. The OVM uses its key KM to encrypt or sign the new firmware. Here KM stands for MVK which is part of every ECU's HSM and is either a symmetric key or a key pair in which case the OMV uses its private key part for signature generation and the ECU uses the public counter part for verifying the signature on the firmware.



A first analysis of this protocol reviels however that the primary focus is the confidentiality of the provided firmware and a simple check of authenticity by the software agent. Without further investigation we will therefore disregard it for the rest of this work.

Feedback was given to the corresponding tasks that the important property that a firmware update protocol in combination with secure boot have to perform is the trustworthy setting of pcr-reference values. Formally this may be expressed such that the setting of the preset_ECR command may only happen with software that is conform to

the stereotype of trustworthy softwares preseted above:

Presetting of Secure Software References Only: (Prop.4.4.3.1)

$$\textit{not-happens}(\{\textit{cmd\_preset\_ECR}(CPU_i, h(sw_j)) \mid i \neq j\})$$

(where sw_j represents a trustworthy software for an agent j.)

4.5 Functional Proof

This section will perform an actual first attempt to verify the security requirement within the sample deployment outline above. We will first outline the Assumptions made against the Pre-Deployment phase, such as shared secrets and group storages.

Then we will investigate the first piece of the functional chain of the use case, namely the internal behavior of the sensor. This will include an in-depth investigation of the software as well as key distribution and secure storage w.r.t. this element of the functional chain.

Then we will investigate the communication between the sensor and the functionally following application ECU. This includes investigations of secure storage and several aspects of the key distribution protocol.

After this we will outline the necessary subsequent proof for the application ECUs internal behaviour as well as the other components of both vehicles up the driver's display. Note that these parts will only directly conclude the final properties of these proof parts rather than perform them completely. Parallels to the to be presented partial proofs will be outlined to demonstrate the similarity of approach.

Finally the partial proofs along the functional path will be combined to the overall safety property of the exemplary system. A discussion on the verify property opposed to the security property that shall be derived will be presented in the following section.

4.5.1 Assumptions against Pre-Deployment

Pre-Shared Secrets with KeyMaster At beginning of the system we will assume that there exist pre shared secrets between each ECU of a vehicle and it's keymaster. These pre shared keys are confidential between the ECUs' HSMs and the key master's HSM:

Confidentiality of PSKs: (Prop.4.5.1.1)

$$conf(\mathcal{A}(PSK_i), PSK_i, \ldots, \{HSM_i, HSM_{KeyMaster_1}\})$$

KeyHanles at ECUs' Secure Storage We further assume the keyhandle of the pre-shared secret to be returned from secure storage under the storage-label of the psk:

Authenticity of PSK-Handles: (Prop.4.5.1.2)

$$\textit{not-happens}(\{SecStorRead(CPU_i, \text{“}StorID_{PSK}\text{”}, Handle) | Handle \neq Handle_{PSK_i}\})$$

Further we assume the handles of the psk only to be returned from secure storage under the storage label of the psk and no other storage label:

Authenticity of PSK-Storage: (Prop.4.5.1.3)

$$\textit{not-happens}(\{\textit{SecStorRead}(CPU_i, StorID, Handle_{PSK_i}) | StorID \neq \text{“}StorID_{PSK}\text{”}\})$$

Group Definitions at KeyMaster's Secure Storage We assume that the GroupIDs within the Secure Storage of the KeyMaster was correctly initiated. This means that the KeyMaster will only hold the Handle of the Sensor's PSK in the group of allowed senders for this group, and that it will only hold the Handle of the Application's PSK in the group of allowed receivers for the group.

Only $Handle_{PSK_{Sensor}}$ *in* “Gid_{Sensor}-*Sender*”: (Prop.4.5.1.4)

$$\textit{not-happens}(\{\textit{SecStorRead}(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle) \mid Handle \neq Handle_{PSK_{Sensor}}\})$$

Only $Handle_{PSK_{Appl}}$ *in* “Gid_{Sensor}-*Recvr*”: (Prop.4.5.1.5)

$$\textit{not-happens}(\{\textit{SecStorRead}(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Recvr\text{”}, HandleList) \mid HandleList \neq \{Handle_{PSK_{Appl}}\}\}))$$

4.5.2 Behaviour of the Sensor of the Sending Vehicle

Application of Firmware Update Protocol As we have mentioned before the most important property of firmware updates is that only the trustworthy software is set as reference value during firmware update Prop.4.4.3.1. For the sensor, this instaciates to

Only sw_{Sensor} *is preset as secure boot reference:* (Prop.4.5.2.1)

$$\textit{not-happens}(\{\textit{preset\_ECR}(HSM_{Sensor_1}, ECR\text{-}ref) | ECR\text{-}ref \neq sw_{Sensor}\})$$

Application of Secure Boot For the example used in this deliverable we assumed a secure boot to be in place that will only allow a software to boot, whose reference values are stored in the HSM before. In other words, any attempt to boot a non-trustworthy software will result in the HSM to deny any operation. In case of the sensor, this property Prop.4.3.3.1 can be specialized to

Secure Boot of the Sensor: (Prop.4.5.2.2)

$$\begin{aligned}\textit{not-happens-within-phase}(&\Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}} \\ &\setminus \{\textit{boot}(CPU_{Sensor_1}, \ldots), \textit{shutdown}(CPU_{Sensor_1}, \ldots)\}, \\ &V(\{\textit{boot}(CPU_{Sensor_1}, sw_x, ECR\text{-}ref) | ecr(sw_x) \neq ECR\text{-}ref\}, \\ &\quad \textit{shutdown}(CPU_{Sensor_1})(1)))\end{aligned}$$

Secure Boot of the Sensor's Software: (Prop.4.5.2.3)

$$\begin{aligned}&\textit{not-happens-within-phase}(\Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}} \\ &\qquad \setminus \{boot(CPU_{Sensor_1}, \ldots), shutdown(CPU_{Sensor_1}, \ldots)\}, \\ &\qquad V(\{boot(CPU_{Sensor_1}, sw_x, ECR\text{-}ref) | sw_x \neq sw_{Sensor}\}, \\ &\qquad\quad shutdown(CPU_{Sensor_1})(1)))\end{aligned}$$

Further we know, that inbetween boot cycles – between shutdown and the next boot – the Sensor cannot perform any action Prop.4.3.1.2. For the sensor, this means

Unavailability of Sensor between shutdown and boot: (Prop.4.5.2.4)

$$\begin{aligned}&\textit{not-happens-within-phase}(\Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}} \setminus \\ &\qquad \{shutdown(CPU_{Sensor_1}, \ldots), boot(CPU_{Sensor_1}, \ldots)\}, \\ &\qquad V(shutdown(CPU_{Sensor_1}), boot(CPU_{Sensor_1}, \ldots)(1)))\end{aligned}$$

We also know, that the only action that can happen before the first boot occures is the setting of a reference ECR value during production Prop.4.3.1.1 for the sensor

Unavailability prior to first boot: (Prop.4.5.2.5)

$$\begin{aligned}&\textit{not-happens-within-phase}(\Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}} \setminus \{boot(CPU_{Sensor_1}, \ldots), \\ &\qquad preset\_ECR(HSM_{Sensor_1}, \ldots)\}, \\ &\qquad V(\emptyset, boot(CPU_{Sensor_1}, \ldots)(1)))\end{aligned}$$

So the above properties state that (i) under untrustworthy software the sensor cannot act malicious because it does not boot Prop.4.5.2.3, that (ii) inbetween boot cycles the sensor does not act malicious as it is turned off Prop.4.5.2.4 and that (iii) before the first boot the sensor does not act malicious as it is turned off Prop.4.5.2.5. The only not yet defined behaviour is the trustworthy software's sw_{Sensor}.

Accordingly, for every *not-happens* software property of the Sensor's trusted software $sw\text{-}prop(sw_{Sensor}, not\text{-}happens(a))$ with $a \notin \{boot(CPU_{Sensor_1}, \ldots), shutdown\ (CPU_{Sensor_1}, \ldots), preset\_ECR(HSM_i, \ldots)\}$ we can utilize SeBB.4.1.4.5 to extend it by the phases from Prop.4.5.2.3 and derive $not\text{-}happens\text{-}within\text{-}phase(a, V(\{boot(CPU_{Sensor_1}, sw_x, ECR\text{-}ref), shutdown(CPU_{Sensor_1})(1)))$. Using SeBB.4.1.4.6 on this property together with Prop.4.5.2.4 and Prop.4.5.2.5 it is easy to conclude by induction that $not\text{-}happens\text{-}within\text{-}phase(a, V(\emptyset, shutdown(CPU_{Sensor_1}, \ldots)(n)))$ with $n \in \mathbb{N}$ and therefore to conclude $not\text{-}happens(a)$. Following this proof we can create a new general SeBB for the application of secure boot in the sensor:

Application of Sensor's secure boot for nothappens (SeBB.4.5.2.1)
External Property: $$not\text{-}happens(a)$$
Internal Property: $$sw\text{-}prop(sw_{Sensor}, not\text{-}happens(a))$$ $$for\ all\ a \in (\Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}}) \setminus \{boot(CPU_{Sensor_1}, \ldots), preset\_ECR(HSM_{Sensor_1}, \ldots), shutdown(CPU_{Sensor_1})\}$$

Further the properties Prop.4.5.2.3, Prop.4.5.2.4 and Prop.4.5.2.5 let us conclude according to SeBB.4.1.4.7 the corresponding precede properties for the phases where the sensor is turned off or attempted to boot with a malicious software:

Secure Boot of the Sensor's Software II: (Prop.4.5.2.6)

$$precede\text{-}within\text{-}phase(\Sigma, \Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}} \setminus \{boot(CPU_{Sensor_1}, \ldots), shutdown(CPU_{Sensor_1}, \ldots)\}, V(\{boot(CPU_{Sensor_1}, sw_x, ECR\text{-}ref) | sw_x \neq sw_{Sensor}\}, shutdown(CPU_{Sensor_1})(1)))$$

Unavailability of Sensor between shutdown and boot: (Prop.4.5.2.7)

$$precede\text{-}within\text{-}phase(\Sigma, \Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}} \setminus \{shutdown(CPU_{Sensor_1}, \ldots), boot(CPU_{Sensor_1}, \ldots)\}, V(shutdown(CPU_{Sensor_1}), boot(CPU_{Sensor_1}, \ldots)(1)))$$

Unavailability prior to first boot: (Prop.4.5.2.8)

$$precede\text{-}within\text{-}phase(\Sigma, \Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}} \setminus \{boot(CPU_{Sensor_1}, \ldots), preset\_ECR(HSM_{Sensor_1}, \ldots)\}, V(\emptyset, boot(CPU_{Sensor_1}, \ldots)(1)))$$

Accordingly, for every *precede* software property of the Sensor's trusted software $sw\text{-}prop(sw_{Sensor}, precede(a, b))$ with $b \notin \{boot(CPU_{Sensor_1}, \ldots), shutdown(CPU_{Sensor_1}, \ldots),$

$preset\_ECR(HSM_i, \ldots)\}$ we can construct the respective SeBB:

Application of Sensor's secure boot for precede (SeBB.4.5.2.2)
External Property: $precede(a, b)$
Internal Property: $sw\text{-}prop(sw_{Sensor}, precede(a, b))$ $for\ all\ b \in (\Sigma_{/CPU_{Sensor_1}} \cup \Sigma_{/HSM_{Sensor_1}}) \setminus \{boot(CPU_{Sensor_1}, \ldots), preset\_ECR(HSM_{Sensor_1}, \ldots), shutdown(CPU_{Sensor_1})\}$

Note that these SeBBs do not represent properties of the architecture but rather the combination of properties and assumptions and therefore represent a unision of several proof steps that can be applied multiple times in the following.

Application of scheduled HSM-Access As it was mentioned above, the asynchronous, non-blocking, session-less interface between CPU and HSM can lead to race-conditions that enable an attacker with good timing e.g. to impersonate another ECU. This can be chountered by scheduling a synchronized access to the ECU in software Prop.4.3.2.1.

The challenge solved by this is that a return value from the HSM can be associated with a command given to the HSM, such that it is known, which parameters were used during the calling of the HSM. It is easy to conclude from a software that performs only one command at a time Prop.4.3.2.1 to that the software can therefore be assured of the command, that a return value refers to:

Scheduled HSM-Access: (Prop.4.5.2.9)

$$precede(cmd\_X(params), ret\_X(\ldots, cmd\_X(params)))$$

Application of Key Distribution Protcol I – Correct key distribution We know that (triggered by policy and application management) at some point in time a key is created that is designated to be the session key between the sensor and the application ECU, namely $SessK_{Sensor}$. This happens durig an action in the sensor's HSM $create\_Random\_Key(HSM_{Sensor_1}, SessK_{Sensor}, Handle_{SessK_{Sensor}})$. A keyhandle is created alongside and returned to the sensor's CPU $ret\_create\_Random\_Key(CPU_{Sensor_1}, Handle_{SessK_{Sensor}}, \ldots)$. The fact that assigns this key and key handle the role of the $SessK_{Sensor}$ is the secure storage of its handle under the label of "$StorID_{Sensor}$". The software properties that are related to this very first part of key distribution constitute that a key saved as "$StorID_{Sensor}$" must have been created by the sensor itself Prop.4.3.2.5 and that this keyhandle must not be stored under any other label Prop.4.3.2.6 neither any other keyhandle stored under this label Prop.4.3.2.7. Thanks to the secure boot of the sensor SeBB.4.5.2.1 and SeBB.4.5.2.2, these properties extend to the whole lifetime of the system:

"$StorID_{Sensor}$"*'s key created locally:* (Prop.4.5.2.10)

$$precede(ret\_create\_Random\_Key(CPU_{Sensor_1}, Handle_{SessK_{Sensor}}, \ldots),$$
$$SecStorWrite(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle_{SessK_{Sensor}})))$$

Only $Handle_{SessK_{Sensor}}$ *is written to* “$StorID_{Sensor}$”: (Prop.4.5.2.11)

$$not\text{-}happens(\{SecStorWrite(CPU_{Sensor_1}, StorID, Handle_{SessK_{Sensor}}) \mid StorID \neq \text{“}StorID_{Sensor}\text{”}\}))$$

$Handle_{SessK_{Sensor}}$ *is written only to* “$StorID_{Sensor}$”: (Prop.4.5.2.12)

$$not\text{-}happens(\{SecStorWrite(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle) \mid Handle \neq Handle_{SessK_{Sensor}}\}))$$

Using the fact that the secure storage holds integrity and authenticity of stored data Prop.4.5.2.11 and Prop.4.5.2.12 we can conclude SeBB.4.1.4.7, that no other than $SessK_{Sensor}$'s key handle will be read from storage under the id of “$StorID_{Sensor}$”:

$Handle_{SessK_{Sensor}}$ *is read only from* “$StorID_{Sensor}$”: (Prop.4.5.2.13)

$$not\text{-}happens(\{SecStorRead(CPU_{Sensor_1}, StorID, Handle_{SessK_{Sensor}}) \mid storeid \neq Handle_{SessK_{Sensor}}\})$$

Only $Handle_{SessK_{Sensor}}$ *is read from* “$StorID_{Sensor}$”: (Prop.4.5.2.14)

$$not\text{-}happens(\{SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle_a) \mid a \neq SessK_{Sensor}\})$$

The trustworthy sensor software has a property Prop.4.3.2.4 that allows it only to distribute the actual $SessK_{Sensor}$ as “Gid_{Sensor}”. Thanks to the secure boot of the sensor SeBB.4.5.2.2 this property extends to the whole system lifetime:

Distribution of $Handle_{SessK_{Sensor}}$ *under the label of* “Gid_{Sensor}”: (Prop.4.5.2.15)

$$limited\text{-}precede(SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle_a)$$
$$\& \; ret\_Key\_Export(CPU_{Sensor_1}, exportBlob(K_c),$$
$$cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, \ldots)),$$
$$SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{PSK}\text{”}, Handle_b)$$
$$\& \; cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, m(exportBlob(K_c), \text{“}Gid_{Sensor}\text{”}),$$
$$Handle_b))$$

As it is known from assumption Prop.4.5.1.2 that only the keyhandle $Handle_{PSK_{Sensor}}$ is returned from secure storage under the label of “$StorID_{PSK}$” we can specialize this

property to this keyhandle SeBB.4.1.4.20.

Distribution of $Handle_{SessK_{Sensor}}$ *under the label of "*Gid_{Sensor}*" with* $Handle_{PSK_{Sensor}}$*:* (Prop.4.5.2.16)

$$\begin{aligned}&\textit{limited-precede}(SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle_a)\\ &\quad \& \; ret\_Key\_Export(CPU_{Sensor_1}, exportBlob(K_c),\\ &\qquad cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, \ldots)),\\ &\quad SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{PSK}\text{”}, Handle_{PSK_{Sensor}})\\ &\quad \& \; cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, m(exportBlob(K_c), \text{“}Gid_{Sensor}\text{”}),\\ &\qquad Handle_{PSK_{Sensor}}))\end{aligned}$$

Because it is known that each key handle used must originate from the secure storage Prop.4.3.1.5 it is easy to conclude that the export using $Handle_{PSK_{Sensor}}$ is always preceded by a *SecStorRead*. We can therefore narrow down the trigger accordingly SeBB.4.1.4.15

Distribution of $Handle_{SessK_{Sensor}}$ *under the label of "*Gid_{Sensor}*" simplified:* (Prop.4.5.2.17)

$$\begin{aligned}&\textit{limited-precede}(SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle_a)\\ &\quad \& \; ret\_Key\_Export(CPU_{Sensor_1}, exportBlob(K_c),\\ &\qquad cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, \ldots)),\\ &\quad cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, m(exportBlob(K_c), \text{“}Gid_{Sensor}\text{”}),\\ &\qquad Handle_{PSK_{Sensor}}))\end{aligned}$$

Due to the properties of the HSM and the scheduled access of the HSM by software we know that an exported KeyBlob will contain the exact key that was commanded to be exported

Correctness of exported key: (Prop.4.5.2.18)

$$\begin{aligned}&\textit{not-happens}(\{ret\_Key\_Export(CPU_{Sensor_1}, exportBlob(K_c),\\ &\qquad cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, \ldots))|a \neq c\})\end{aligned}$$

This can be used to further narrow down the software property from above, namely that the key inside the exported blob corresponds to the keyhandle that was read from secure storage SeBB.4.1.4.19

Distribution of $SessK_{Sensor}$ *under the label of "*Gid_{Sensor}*":* (Prop.4.5.2.19)

$$\begin{aligned}&\textit{limited-precede}(SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle_a)\\ &\quad \& \; ret\_Key\_Export(CPU_{Sensor_1}, exportBlob(K_a),\\ &\qquad cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, \ldots)),\\ &\quad cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}),\\ &\qquad Handle_{PSK_{Sensor}}))\end{aligned}$$

Since we know from Prop.4.5.2.14 that only $Handle_{SessK_{Sensor}}$ is returned from secure storage we can easily conclude, SeBB.4.1.4.7 the the sensor will issue the MAC-generation for the exported key blob only, when the blob also contains the actual session key $SessK_{Sensor}$.

Distribution only of $SessK_{Sensor}$ *under the label of "*Gid_{Sensor}*" from CPU:* (Prop.4.5.2.20)

$$\textit{not-happens}(\{cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}), Handle_{PSK_{Sensor}})) \mid a \neq SessK_{Sensor}\})$$

Because of the authentic connection between the CPU and the HSM of an ECU Prop.4.3.3.24 we may again further conclude that also no $Hash\_mac\_sign$ will be executed with a key different from $SessK_{Sensor}$.

Distribution only of $SessK_{Sensor}$ *under the label of "*Gid_{Sensor}*" from HSM:* (Prop.4.5.2.21)

$$\textit{not-happens}(\{Hash\_mac\_sign(HSM_{Sensor_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}), PSK_{Sensor})) \mid a \neq SessK_{Sensor}\})$$

So we can proof, that only $SessK_{Sensor}$ will be signed and send from the sensor to the keymaster for use in the key distribution of the session key for "Gid_{Sensor}".

Application of Key Distribution Protcol II – Confidentiality of Distributed Key With the start of the system, there are no session keys distributed. Therefore $Sensor_1$ will start by generating a session key for communication with $Appl_1$. After this, the creation of this key $SessK_{Sensor}$, according to Prop.4.3.3.9 it will be confidential to the creator only:

Confidential creation of $SessK_{Sensor}$*:* (Prop.4.5.2.22)

$$conf\text{-}within\text{-}phase(\mathcal{A}(SessK_{Sensor}), SessK_{Sensor}, \ldots, \{HSM_{Sensor_1}\}, V(\emptyset, create\_Random\_Key(HSM_{Sensor_1}, SessK_{Sensor}, Handle_{SessK_{Sensor}})(1, in)))$$

It is a property of the HSM, that a key cannot be observed from outside of the HSM Prop.4.3.3.10 and that the only way for any other agent to gain knowledge of an HSM's key is through an explicite keyExport by the HSM Prop.4.3.3.11. Accordingly the confedientiality of the newly created key extends up to the first keyExport command of the HSM that created the key. In this case, the sensor's HSM – being the creator of the key Prop.4.5.2.22 – will be the only agent with knowledge about the key up to excluding its first keyExport operation through using SeBB.4.1.4.21.

Confidentiality of $SessK_{Sensor}$ *up to* Key_Export *I:* (Prop.4.5.2.23)

$$conf\text{-}within\text{-}phase(\mathcal{A}(SessK_{Sensor}), SessK_{Sensor}, \ldots, \{HSM_{Sensor_1}\}, V(\emptyset, Key\_Export(HSM_{Sensor_1}, SessK_{Sensor}, \ldots)(1, ex)))$$

We know, that sw_{Sensor} will only use key handles that are read from secure storage when issuing a command to the HSM Prop.4.3.1.5. This hold expecially for cmd_Key_Export for both keys, the key to be exported as well as the key being exported. Using the fact that only sw_{Sensor} is booted on the sensor SeBB.4.5.2.2 we can therefore conclude that the key handles sent to the HSM allways originate from secure storage.

ExportHandle from Secure Storage: (Prop.4.5.2.24)

$$precede(SecStorRead(CPU_{Sensor_1}, StorID, Handle_a), \\ cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, Handle_b))$$

For the specialized case of $Handle_a = Handle_{SessK_{Sensor}}$ we know that $Handle_{SessK_{Sensor}}$ is only returned from secure storage under "$StorID_{Sensor}$" Prop.4.5.2.13. We may therefore conclude with SeBB.4.1.4.7

ExportHandle from Secure Storage II: (Prop.4.5.2.25)

$$precede(SecStorRead(CPU_{Sensor_1}, "StorID_{Sensor}", Handle_{SessK_{Sensor}}), \\ cmd\_Key\_Export(CPU_{Sensor_1}, Handle_{SessK_{Sensor}}, Handle_b))$$

Further we know the trustworthy sensor software sw_{Sensor} will export the session key only to the keymaster Prop.4.3.2.8. Because we know that only this software is booted on the sensor, we can refine it SeBB.4.5.2.2 to

Export SessK only to keyMaster: (Prop.4.5.2.26)

$$limited\text{-}precede(SecStorRead(CPU_{Sensor_1}, "StorID_{PSK}", Handle_b), \\ SecStorRead(CPU_{Sensor_1}, "StorID_{Sensor}", Handle_a)\& \\ cmd\_Key\_Export(CPU_{Sensor_1}, Handle_a, Handle_b)))$$

Because we know, that only $Handle_{SessK_{Sensor}}$ is returned from secure storage under "$StorID_{Sensor}$" Prop.4.5.2.14 we can specialize SeBB.4.1.4.15 this property to

Export SessK only to keyMaster II: (Prop.4.5.2.27)

$$limited\text{-}precede(SecStorRead(CPU_{Sensor_1}, "StorID_{PSK}", Handle_b), \\ SecStorRead(CPU_{Sensor_1}, "StorID_{Sensor}", Handle_{SessK_{Sensor}})\& \\ cmd\_Key\_Export(CPU_{Sensor_1}, Handle_{SessK_{Sensor}}, Handle_b)))$$

In combination with the fact, that the $Handle_{SessK_{Sensor}}$ allways originates from $SecStorRead$ Prop.4.5.2.25 we can conclude, that $Handle_{SessK_{Sensor}}$ shall always be exported only with the keyhandle that is stored under "$StorID_{PSK}$".

Export $Handle_{SessK_{Sensor}}$ only with "$StorID_{PSK}$": (Prop.4.5.2.28)

$$precede(SecStorRead(CPU_{Sensor_1}, "StorID_{PSK}", Handle), \\ cmd\_Key\_Export(CPU_{Sensor_1}, Handle_{SessK_{Sensor}}, Handle)))$$

From Prop.4.5.1.2 we know that only $Handle_{PSK_{Sensor}}$ is returned from secure storage under "$StorID_{PSK}$". We may therefore conclude SeBB.4.1.4.7 with Prop.4.5.2.28 that a keyExport command concerning $Handle_{SessK_{Sensor}}$ will use $Handle_{PSK_{Sensor}}$.

Export $Handle_{SessK_{Sensor}}$ *only with* $Handle_{PSK_{Sensor}}$*:* (Prop.4.5.2.29)

$$\begin{aligned} &\textit{not-happens}(\{cmd\_Key\_Export(CPU_{Sensor_1}, Handle_{SessK_{Sensor}}, Handle) \\ &\qquad | Handle \neq Handle_{PSK_{Sensor}}\}) \end{aligned}$$

And further we can conclude SeBB.4.1.4.7 using the authentic connection between sensor's CPU and sensor's HSM Prop.4.3.3.20 that an export only happens using PSK_{Sensor}.

$SessK_{Sensor}$ *only exported with* PSK_{Sensor}*:* (Prop.4.5.2.30)

$$\begin{aligned} &\textit{not-happens}(\{Key\_Export(CPU_{Sensor_1}, SessK_{Sensor}, SS, \ldots) \\ &\qquad | SS \neq PSK_{Sensor}\}) \end{aligned}$$

Because PSK_{Sensor} is confidential for the sensor and the keymaster Prop.4.5.1.1 and $SessK_{Sensor}$ is only known to the sensor up to the first keyExport Prop.4.5.2.23 we can apply SeBB.4.4.2.2 to conclude that after the first keyExport $SessK_{Sensor}$ is confidential to the sensor and the keymaster only

Confidentiality of $SessK_{Sensor}$ *until first* Key_Export*:* (Prop.4.5.2.31)

$$\begin{aligned} &conf\text{-}within\text{-}phase(\mathcal{A}(SessK_{Sensor}), SessK_{Sensor}, \ldots, \{HSM_{Sensor_1}, HSM_{KeyMaster_1}\}, \\ &\qquad V(\emptyset, \{Key\_Export(HSM_{Sensor_1}, SessK_{Sensor}, \ldots)(1, in)\})) \end{aligned}$$

Integration of Sensor Properties From the assumptions against trustworthy software we know that keyhandles originate from secure storage Prop.4.3.1.5. This is especially true for signing of $data_{Sensor}$ with $Handle_{SessK_{Sensor}}$. Because we have a secure boot in place for the sensor, we can conclude SeBB.4.5.2.2 that the sensor will read its $Handle_{SessK_{Sensor}}$ from secure storage.

$Handle_{SessK_{Sensor}}$ *from secure storage:* (Prop.4.5.2.32)

$$\begin{aligned} &precede(SecStorRead(CPU_{Sensor_1}, StorID, Handle_{SessK_{Sensor}}), \\ &\qquad cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, data_{Sensor}, Handle_{SessK_{Sensor}}, \ldots)) \end{aligned}$$

However, we know that $Handle_{SessK_{Sensor}}$ is only stored under "$StorID_{Sensor}$" Prop.4.5.2.13 we can further conclude SeBB.4.1.4.14

$Handle_{SessK_{Sensor}}$ *from "*$StorID_{Sensor}$*":* (Prop.4.5.2.33)

$$\begin{aligned} &precede(SecStorRead(CPU_i, \text{“}StorID_{Sensor}\text{”}, Handle_{SessK_{Sensor}}), \\ &\qquad cmd\_Hash\_mac\_sign(CPU_i, data_{Sensor}, Handle_{SessK_{Sensor}}, \ldots)) \end{aligned}$$

We know that a trustworthy sensor software will only sign those data with a secure storages session key handle, that it actually measured Prop.4.3.2.3. Because of the secure boot, we can conclude SeBB.4.5.2.2 that this is always the case for the sensor.

Sensor CPU signs only $data_{Sensor}$ *I:* (Prop.4.5.2.34)

$$\begin{aligned} &\textit{limited-precede}(sense(CPU_{Sensor_1}, data_{Sensor}),\\ &\qquad SecStorRead(CPU_{Sensor_1}, \text{“}StorID_{Sensor}\text{”}, Handle)\ \&\\ &\qquad cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, data_{Sensor}, Handle)) \end{aligned}$$

Combining this property with the fact, that the keyhandle used for signing $Handle_{SessK_{Sensor}}$ only originates from secure storage under the corresponding label Prop.4.5.2.33, we can conclude SeBB.4.1.4.15 that data signed by $Handle_{SessK_{Sensor}}$ has always been measured before

Sensor CPU signs only $data_{Sensor}$ *II:* (Prop.4.5.2.35)

$$\begin{aligned} &precede(sense(CPU_{Sensor_1}, data_{Sensor}),\\ &\qquad cmd\_Hash\_mac\_sign(CPU_{Sensor_1}, data_{Sensor}, Handle_{SessK_{Sensor}}, \ldots)) \end{aligned}$$

From property Prop.4.3.3.24 we know that the HSM will only perform those signatures that it is commanded to by its CPU. We may therefore conclude SeBB.4.1.4.2 that only $data_{Sensor}$ will ever be signed with $SessK_{Sensor}$.

Sensor signs only $data_{Sensor}$*:* (Prop.4.5.2.36)

$$\begin{aligned} &precede(sense(CPU_{Sensor_1}, data_{Sensor}),\\ &\qquad Hash\_mac\_sign(HSM_{Sensor_1}, data_{Sensor}, SessK_{Sensor}, \ldots)) \end{aligned}$$

4.5.3 KeyMaster between Sensor and Application ECU

Secure Boot The secure Boot of the KeyMaster can be similarly derived as the Sensor's. Starting from Prop.4.3.3.1 it can be derived:

Application of KeyMasters's secure boot for nothappens (SeBB.4.5.3.1)
External Property: $not\text{-}happens(a)$
Internal Property: $sw\text{-}prop(sw_{KeyMaster}, not\text{-}happens(a))$ $for\ all\ a \in (\Sigma_{/CPU_{KeyMaster_1}} \cup \Sigma_{/HSM_{KeyMaster_1}}) \setminus \{boot(CPU_{KeyMaster_1}, \ldots),$ $preset\_ECR(HSM_{KeyMaster_1}, \ldots), shutdown(CPU_{KeyMaster_1})\}$

Application of KeyMasters's secure boot for precede (SeBB.4.5.3.2)
External Property: $$precede(a, b)$$
Internal Property: $$sw\text{-}prop(sw_{KeyMaster}, precede(a, b))$$ $$for\ all\ b \in (\Sigma_{/CPU_{KeyMaster_1}} \cup \Sigma_{/HSM_{KeyMaster_1}}) \setminus \{boot(CPU_{KeyMaster_1}, \ldots), preset\_ECR(HSM_{KeyMaster_1}, \ldots), shutdown(CPU_{KeyMaster_1})\}$$

Application of Key Distribution Protcol III – Correctness of distributed Key The confidentiality of the PSK_{Sensor} between sensor and key master Prop.4.5.1.1 lets us conclude that whenever a message signed by this key is verified at the HSM of the keymaster, either the keymaster or the sensor must have send it, according to the transport protocols HMAC-sebb SeBB.4.4.1.1.

Origin of $SessK_{Sensor}$ distribution message: (Prop.4.5.3.1)

$$precede(\{Hash\_mac\_sign(HSM_{Sensor_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}), PSK_{Sensor}), Hash\_mac\_sign(HSM_{KeyMaster_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}), PSK_{Sensor})\}, Hash\_mac\_verify(HSM_{KeyMaster_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}), PSK_{Sensor}))$$

Because $Handle_{PSK_{Sensor}}$ is not part of the target-group the the keymaster distributes keys to Prop.4.5.1.5, it will not issue a sign command for message containing “Gid_{Sensor}”.

Keymaster does not sign a “Gid_{Sensor}” message with PSK_{Sensor}: (Prop.4.5.3.2)

$$not\text{-}happens(Hash\_mac\_sign(HSM_{KeyMaster_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}), PSK_{Sensor}))$$

Further we know that from Prop.4.5.2.21 that the sensor's HSM will only do so with the $SessK_{Sensor}$ being part of the exportBlob. Accordingly we may conclude SeBB.4.1.4.7 together with Prop.4.5.3.1 that a verification at the keymaster for a “Gid_{Sensor}”-marked message will only contain $SessK_{Sensor}$ in the *exportBlob*:

Distribution only of $SessK_{Sensor}$ under the label of “Gid_{Sensor}” to HSM: (Prop.4.5.3.3)

$$not\text{-}happens(\{Hash\_mac\_verify(HSM_{KeyMaster_1}, m(exportBlob(K_a), \text{“}Gid_{Sensor}\text{”}), PSK_{Sensor})) \mid a \neq SessK_{Sensor}\})$$

And accordingly we can further conclude that only for $SessK_{Sensor}$ the HSM resturs a valid MAC because of scheduled HSM access Prop.4.5.2.9:

Distribution only of $SessK_{Sensor}$ under the label of “Gid_{Sensor}”: (Prop.4.5.3.4)

$$\textit{not-happens}(\{\textit{ret\_Hash\_mac\_verify}(CPU_{KeyMaster_1}, \textit{cmd\_Hash\_mac\_verify}(CPU_{KeyMaster_1}, m(\textit{exportBlob}(K_a), \text{“}Gid_{Sensor}\text{”}), Handle_{PSK_{Sensor}})) \mid a \neq SessK_{Sensor}\})$$

From the Software Propety for the KeyMaster Prop.4.3.2.11 together with Secure Boot SeBB.4.5.3.2 we ca conclude that it will only import a Key from the senders of “Gid_{Sensor}” if the message containing the *exportBlob* together with the “Gid_{Sensor}” had a correct MAC:

KeyMaster Behaviour on Import I: (Prop.4.5.3.5)

$$\begin{aligned}&\textit{limited-precede}(\textit{SecStorRead}(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_a)\\ &\quad \& \textit{ret\_Hash\_mac\_verify}(CPU_{KeyMaster_1}, \textit{cmd\_Hash\_mac\_verify}(\\ &\qquad CPU_{KeyMaster_1}, m(\textit{exportBlob}, \text{“}Gid_{Sensor}\text{”}), Handle_a)),\\ &\qquad \textit{SecStorRead}(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_b)\\ &\quad \& \textit{cmd\_Key\_Import}(CPU_{KeyMaster_1}, \textit{exportBlob}, Handle_b))\end{aligned}$$

Through the knowledge about the KeyMasters Secure Storage initialization Prop.4.5.1.4 we can specialize this to $Handle_{PSK_{Sensor}}$:

KeyMaster Behaviour on Import II: (Prop.4.5.3.6)

$$\begin{aligned}&\textit{limited-precede}(\textit{SecStorRead}(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_a)\\ &\quad \& \textit{ret\_Hash\_mac\_verify}(CPU_{KeyMaster_1}, \textit{cmd\_Hash\_mac\_verify}(\\ &\qquad CPU_{KeyMaster_1}, m(\textit{exportBlob}, \text{“}Gid_{Sensor}\text{”}), Handle_a)),\\ &\qquad \textit{SecStorRead}(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_{PSK_{Sensor}})\\ &\quad \& \textit{cmd\_Key\_Import}(CPU_{KeyMaster_1}, \textit{exportBlob}, Handle_{PSK_{Sensor}}))\end{aligned}$$

Because we know that every keyhandle comes from secure storage Prop.4.3.1.5 we can conclude SeBB.4.1.4.15 that every import with $Handle_{PSK_{Sensor}}$ must have a valid MAC:

KeyMaster Behaviour on Import III: (Prop.4.5.3.7)

$$\begin{aligned}&\textit{limited-precede}(\textit{SecStorRead}(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_a)\\ &\quad \& \textit{ret\_Hash\_mac\_verify}(CPU_{KeyMaster_1}, \textit{cmd\_Hash\_mac\_verify}(\\ &\qquad CPU_{KeyMaster_1}, m(\textit{exportBlob}, \text{“}Gid_{Sensor}\text{”}), Handle_a)),\\ &\qquad \textit{cmd\_Key\_Import}(CPU_{KeyMaster_1}, \textit{exportBlob}, Handle_{PSK_{Sensor}}))\end{aligned}$$

Again we can use the knowledge about the initialization of secure Group Definition Storage at the KeyMaster Prop.4.5.1.4 to conclude that this MAC must have be validated using $Handle_{PSK_{Sensor}}$.

KeyMaster Behaviour on Import IV: (Prop.4.5.3.8)

$$\begin{aligned}&\textit{limited-precede}(SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_{PSK_{Sensor}})\\ &\quad \& \; ret\_Hash\_mac\_verify(CPU_{KeyMaster_1}, cmd\_Hash\_mac\_verify(\\ &\qquad CPU_{KeyMaster_1}, m(exportBlob, \text{“}Gid_{Sensor}\text{”}), Handle_{PSK_{Sensor}})),\\ &\quad cmd\_Key\_Import(CPU_{KeyMaster_1}, exportBlob, Handle_{PSK_{Sensor}}))\end{aligned}$$

And as this keyHandle must also originates from secure storage Prop.4.3.1.5 we may further simplify this property SeBB.4.1.4.15 to state that every $exportBlob$ being imported using $Handle_{PSK_{Sensor}}$ has a valid MAC verified using $Handle_{PSK_{Sensor}}$:

KeyMaster Behaviour on Import V: (Prop.4.5.3.9)

$$\begin{aligned}&precede(ret\_Hash\_mac\_verify(CPU_{KeyMaster_1}, cmd\_Hash\_mac\_verify(\\ &\qquad CPU_{KeyMaster_1}, m(exportBlob, \text{“}Gid_{Sensor}\text{”}), Handle_{PSK_{Sensor}})),\\ &\quad cmd\_Key\_Import(CPU_{KeyMaster_1}, exportBlob, Handle_{PSK_{Sensor}}))\end{aligned}$$

We remember from Prop.4.5.3.4 that only $SessK_{Sensor}$'s $exportBlob$ will ever carry a valid MAC with PSK_{Sensor}. Therefore it is easy to conclude SeBB.4.1.4.7 that a keyImport command with $Handle_{PSK_{Sensor}}$ will only occure for $SessK_{Sensor}$'s $exportBlob$:

KeyMaster Behaviour on Import VI: (Prop.4.5.3.10)

$$\begin{aligned}&\textit{not-happens}(\{cmd\_Key\_Import(CPU_{KeyMaster_1}, exportBlob(K_a), Handle_{PSK_{Sensor}})\\ &\qquad \mid a \neq SessK_{Sensor}\})\end{aligned}$$

and further because of scheduled HSM access Prop.4.5.2.9 to conclude SeBB.4.1.4.7 that an actual keyImport and its return with $Handle_{PSK_{Sensor}}$ will only occure for $SessK_{Sensor}$'s $exportBlob$:

KeyMaster Behaviour on Import: (Prop.4.5.3.11)

$$\begin{aligned}&\textit{not-happens}(\{ret\_Key\_Import(CPU_{KeyMaster_1}, Handle_{SessK_{Sensor}},\\ &\qquad cmd\_Key\_Import(exportBlob(K_a), Handle_{PSK_{Sensor}}))\\ &\qquad \mid a \neq SessK_{Sensor}\})\end{aligned}$$

Application of Key Distribution Protocol IV – Confidentiality of Distributed Key We know that until after the first keyExport of $SessK_{Sensor}$ at the sensor $SessK_{Sensor}$ is confidential to the sensor and the key master. It is a property of the HSM, that a key cannot be observed from outside of the HSMs Prop.4.3.3.10 and that the only way for any other agent to gain knowledge of the HSMs' key is through an explicite keyExport by one of the HSM Prop.4.3.3.11. Accordingly we can use SeBB.4.1.4.21 to conclude the confidentiality of the key up to excluding the second export of the sensor or the first export of the key master.

Confidentiality of $SessK_{Sensor}$ up to Key_Export: (Prop.4.5.3.12)

$$conf\text{-}within\text{-}phase(\mathcal{A}(SessK_{Sensor}), SessK_{Sensor}, \ldots, \{HSM_{Sensor_1}, HSM_{KeyMaster_1}\},$$
$$V(\emptyset, \{Key\_Export(HSM_{Sensor_1}, SessK_{Sensor}, \ldots)(2, ex),$$
$$Key\_Export(HSM_{KeyMaster_1}, SessK_{Sensor}, \ldots)(1, ex)\}))$$

For the sensor we know that an export of $SessK_{Sensor}$ will only be perfomed using PSK_{Sensor} Prop.4.5.2.30. Therefore the inductive application of SeBB.4.4.2.2 and the steps above will lead us to the case, that $SessK_{Sensor}$ is confidential to the sensor and the keymaster up to the keymaster's first export.

Confidentiality of $SessK_{Sensor}$ up to Keymaster Key_Export: (Prop.4.5.3.13)

$$conf\text{-}within\text{-}phase(\mathcal{A}(SessK_{Sensor}), SessK_{Sensor}, \{HSM_{Sensor_1}, HSM_{KeyMaster_1}\},$$
$$V(\emptyset, Key\_Export(HSM_{KeyMaster_1}, SessK_{Sensor}, \ldots)(1, ex)))$$

We know for the trustworthy KeyMaster software, that it will only use the GroupReceivers' key handles from secure storage when it exports a key that was imported with one of the GroupSenders' key handles Prop.4.3.2.10. Because we know that the keymaster will only boot the trustworthy keymaster software $sw_{KeyMaster}$ we can conclude SeBB.4.5.3.2

Keymasters Gid compliance I: (Prop.4.5.3.14)

$$limited\text{-}precede(SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Recvr\text{”}, Handle_a),$$
$$SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_b)$$
$$\& \; ret\_Key\_Import(CPU_{KeyMaster_1}, Handle_c, cmd\_Key\_Import($$
$$CPU_{KeyMaster_1}, exportBlob(SessK_d), Handle_b))$$
$$\& \; cmd\_Key\_Export(CPU_{KeyMaster_1}, Handle_c, Handle_a))$$

As we assume only $Handle_{PSK_{Sensor}}$ to be in the group “Gid_{Sensor}-Sender” Prop.4.5.1.4 we can conclude SeBB.4.1.4.20:

Keymasters Gid compliance II: (Prop.4.5.3.15)

$$limited\text{-}precede(SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Recvr\text{”}, Handle_a),$$
$$SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender\text{”}, Handle_{PSK_{Sensor}})$$
$$\& \; ret\_Key\_Import(CPU_{KeyMaster_1}, Handle_c, cmd\_Key\_Import($$
$$CPU_{KeyMaster_1}, exportBlob(SessK_c), Handle_{PSK_{Sensor}}))$$
$$\& \; cmd\_Key\_Export(CPU_{KeyMaster_1}, Handle_c, Handle_a))$$

Using the fact that a keyImport return for $Handle_{PSK_{Sensor}}$ will only happen for $SessK_{Sensor}$ Prop.4.5.3.11 leads us to

Keymasters Gid compliance III: (Prop.4.5.3.16)

$$\begin{aligned}
&\textit{limited-precede}(SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Recvr'', Handle_a),\\
&\qquad SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Sender'', Handle_{PSK_{Sensor}})\\
&\quad \& \; ret_Key_Import(CPU_{KeyMaster_1}, Handle_{SessK_{Sensor}}, cmd_Key_Import(\\
&\qquad CPU_{KeyMaster_1}, exportBlob(SessK_{Sensor}), Handle_{PSK_{Sensor}}))\\
&\quad \& \; cmd_Key_Export(CPU_{KeyMaster_1}, Handle_{SessK_{Sensor}}, Handle_a))
\end{aligned}$$

It is easy to see, that a CPU can only issue a keyExport command for a key, that was created or imported before:

Export of Created or Imported Keys only: (Prop.4.5.3.17)

$$\begin{aligned}
&precede(\{ret_Key_Import(CPU_{KeyMaster_1}, Handle),\\
&\qquad ret_create_Random_Key(CPU_{KeyMaster_1}, Handle)\},\\
&\quad cmd_Key_Export(CPU_{KeyMaster_1}, Handle))
\end{aligned}$$

It is further the case that a key can only be created once. Therefore the KeyMaster cannot have created $SessK_{Sensor}$:

$SessK_{Sensor}$ *not by KeyMaster:* (Prop.4.5.3.18)

$$not\text{-}happens(ret\_create\_Random\_Key(CPU_{KeyMaster_1}, SessK_{Sensor}))$$

From these two properties it is easy to conclude SeBB.4.1.4.18, that the keyMaster must have imported $SessK_{Sensor}$:

$SessK_{Sensor}$ *imported by KeyMaster:* (Prop.4.5.3.19)

$$\begin{aligned}
&precede(ret_Key_Import(CPU_{KeyMaster_1}, Handle_{SessK_{Sensor}}),\\
&\qquad cmd_Key_Export(CPU_{KeyMaster_1}, Handle_{SessK_{Sensor}}))
\end{aligned}$$

This property Prop.4.5.3.19 and the fact that key handles originate from secure storage Prop.4.3.1.5 can be used to simplify Prop.4.5.3.16 such that $Handle_{SessK_{Sensor}}$ will only be exported with those target keyhandles that originate from secure storage as valid receivers for the group “Gid_{Sensor}”:

Keymasters Gid compliance: (Prop.4.5.3.20)

$$\begin{aligned}
&precede(SecStorRead(CPU_{KeyMaster_1}, \text{“}Gid_{Sensor}\text{-}Recvr'', \{Handle_a, \ldots\}),\\
&\qquad cmd_Key_Export(CPU_{KeyMaster_1}, Handle_{SessK_{Sensor}}, Handle_a))
\end{aligned}$$

From secure storage initialization we know, that only $Handle_{PSK_{Appl}}$ is in the group of receivers for “Gid_{Sensor}” Prop.4.5.1.5

Export with $Handle_{PSK_{Appl}}$ *only:* (Prop.4.5.3.21)

$$\begin{aligned}
&not\text{-}happens(\{cmd_Key_Export(CPU_{KeyMaster_1}, Handle_{SessK_{Sensor}}, Handle)\\
&\qquad \mid Handle \neq Handle_{PSK_{Appl}}\}))
\end{aligned}$$

And with the properties of the HSM-CPU connection Prop.4.3.3.20 we know that the export will only happen with PSK_{Appl}:

Export with PSK_{Appl} *only:* (Prop.4.5.3.22)
$$not\text{-}happens(\{Key\_Export(CPU_{KeyMaster_1}, SessK_{Sensor}, PSK) \mid PSK \neq PSK_{Appl}\}))$$

Using the knowledge about the key distribution protocol SeBB.4.4.2.2, the export by the keymaster with PSK_{Appl} only and the knowledge that $SessK_{Sensor}$ was only confidential for HSM_{Sensor_1} and $HSM_{KeyMaster_1}$ up to the keyMasters keyExport Prop.4.5.3.12 we can conclude that after the keyMaster's keyExport $SessK_{Sensor}$ in confidential to those three HSMs only:

Confidentiality of $SessK_{Sensor}$ *up to* Key_Export *IV:* (Prop.4.5.3.23)
$$conf\text{-}within\text{-}phase(\mathcal{A}(SessK_{Sensor}), SessK_{Sensor}, \{HSM_{Sensor_1}, HSM_{Appl_1}, HSM_{KeyMaster_1}\}, V(\emptyset, Key\_Export(HSM_{KeyMaster_1}, SessK_{Sensor}, \ldots)(1, in)))$$

For the Application ECU we know that it can be shown through secure boot and a trustworthy software that no command for a keyExport for a keyHandle will be given, that came from secure storage under the label of "$StorID_{Sensor}$" similarly to the Sensor and the KeyMaster:

Application ECU's Trustworthy software I: (Prop.4.5.3.24)
$$not\text{-}precede(SecStorRead(CPU_{Appl_1}, "StorID_{Sensor}", Handle)), cmd\_Key\_Export(CPU_{Appl_1}, Handle, \ldots))$$

Combined with knowledge that every keyHandle originates from secure storage Prop.4.3.1.5 we man conclude that the Application ECU will never command a keyExport for $Handle_{SessK_{Sensor}}$:

Application ECU's Trustworthy software II: (Prop.4.5.3.25)
$$not\text{-}happens(cmd\_Key\_Export(CPU_{Appl_1}, Handle_{SessK_{Sensor}}, \ldots))$$

and further due to Prop.4.3.3.20 we can conclude SeBB.4.1.4.7 that $SessK_{Sensor}$ will never be exported by HSM_{Appl_1}:

Application ECU's Trustworthy software III: (Prop.4.5.3.26)
$$not\text{-}happens(Key\_Export(HSM_{Appl_1}, SessK_{Sensor}, \ldots))$$

The Unobservability of keys in $HSM_{Sensor_1}, HSM_{KeyMaster_1}, HSM_{Appl_1}$ Prop.4.3.3.11, the non-disclosure of keys by $HSM_{Sensor_1}, HSM_{KeyMaster_1}, HSM_{Appl_1}$ Prop.4.3.3.10 and the fact that the key is only exported from HSM_{Sensor_1} to $HSM_{KeyMaster_1}$ and from $HSM_{KeyMaster_1}$ to HSM_{Appl_1} and not exported by HSM_{Appl_1} at all Prop.4.5.3.25 let's us finally conclude that the key is confidential over the whole system lifetime from those three HSM only:

Final Confidentiality Property of Key Distribution: (Prop.4.5.3.27)
$$conf(\mathcal{A}(SessK_{Sensor}), SessK_{Sensor}, \{HSM_{Sensor_1}, HSM_{Appl_1}, HSM_{KeyMaster_1}\})$$

Application of Key Distribution Protocol for UseFlags From the sensor's software property we can conclude that it will never create $SessK_{Sensor}$ with a $sign_{transp}$ that would allow receivers of the keyBlob to make signatures with it:

Creation of SessK with validation only transportation: (Prop.4.5.3.28)

$$\begin{aligned} &\textit{not-happens}(\{create\_Random\_Key(HSM_{Sensor_1}, SessK_{Sensor}(useflags)) \\ &\quad | sign_{transp} \in useflags\}) \end{aligned}$$

Using the knowledge about the behaviour of HSM SeBB.4.4.2.1 we conclude from this Prop.4.5.3.28 and the knowledge of the keys confidentiality only to HSM Prop.4.5.3.27 that only HSM_{Sensor_1} will ever perform signature operations with this key:

Export of SessK with validation based on transportation: (Prop.4.5.3.29)

$$\begin{aligned} &\textit{not-happens}(\{Hash\_mac\_sign(HMI_i, SessK_{Sensor}(useflags)) \\ &\quad | sign \in useflags, HMI_i \neq HSM_{Sensor_1}\}) \end{aligned}$$

Intagration of Communication Properties Finally all these properties can be integrated.

We know of the Confidentiality of $SessK_{Sensor}$ to $\{HSM_{Sensor_1}, HSM_{Appl_1}, HSM_{KeyMaster_1}\}$ Prop.4.5.3.27. According to the properties of the transport protocol SeBB.4.4.1.1 this means that only these three agent can have generated a MAC whenever anyone – i.e. HSM_{Appl_1} – performs a verification of a signature:

Only the possible signature sources: (Prop.4.5.3.30)

$$\begin{aligned} &\textit{precede}(\{Hash\_mac\_sign(HSM_{Sensor_1}, data, SessK_{Sensor}), \\ &\quad Hash\_mac\_sign(HSM_{Appl_1}, data, SessK_{Sensor}), \\ &\quad Hash\_mac\_sign(HSM_{KeyMaster_1}, data, SessK_{Sensor})\}, \\ &\quad Hash\_mac\_verify(HSM_{Appl_1}, SessK_{Sensor}, data)) \end{aligned}$$

From Prop.4.5.3.29 we can directly derive that neighter the App-HSM:

App-HSM does not sign: (Prop.4.5.3.31)

$$\textit{not-happens}(Hash\_mac\_sign(HSM_{Appl_1}, data, SessK_{Sensor}))$$

nor the KeyMaster HSM will produce MAC with $SessK_{Sensor}$:

KM-HSM does not sign: (Prop.4.5.3.32)

$$\textit{not-happens}(Hash\_mac\_sign(HSM_{KeyMaster_1}, data, SessK_{Sensor}))$$

Using SeBB.4.1.4.18 we can use there three properties Prop.4.5.3.30,Prop.4.5.3.31 and Prop.4.5.3.32 to conclude that only HSM_{Sensor_1} can have signed a message with $SessK_{Sensor}$ whenever HSM_{Appl_1} verifies it with $SessK_{Sensor}$:

Final Property of Sensor–Application-Communication: (Prop.4.5.3.33)

$$\begin{aligned} &\textit{precede}(Hash\_mac\_sign(HSM_{Sensor_1}, data_{Sensor}, SessK_{Sensor}, \ldots), \\ &\quad Hash\_mac\_verify(HSM_{Appl_1}, data_{Sensor}, SessK_{Sensor}, \ldots)) \end{aligned}$$

4.5.4 Behaviour of the Application ECU

The subsequent proof for this example system will in general follow the same priciples. For the APP-ECU's software we will need the property that only after having verified the MAC of data with a certain key who's keyHandle lies in secure storage, will the ECU use this data to issue a warning:

Application ECU's functional behaviour: (Prop.4.5.4.1)

$$precede(Hash\_mac\_verify(HSM_{Appl_1}, data_{Sensor}, SessK_{Sensor}, \ldots)),\\ process(HSM_{CCU_1}, data_{Sensor}, warning)$$

4.5.5 Behaviour and Communication of remaining Vehicle 1

For the remainder of Vehicle 1 it can be shown similarly that keys are distributed in a secure manner through the keymaster. If the corresponding group and PSK identifieres are deployed correctly to the ECUs' secure storages and the behaviour of the ECUs' software – being input to the secure boot – is correct.

This let's us conclude that only those warnings are signed (and subsequently send out) that actually originate from a processing of sensor data at the application ECU:

Remainder of Vehicle 1: (Prop.4.5.5.1)

$$precede(processHSM_{CCU_1}, data_{Sensor}, warning),\\ ECCsign(HSM_{CCU_1}, warning, PrivK_a, sig(PrivK_a, warning)))$$

4.5.6 Communication between Vehicle 1 and Vehicle 2

The communication between vehicle 1 and vehicle 2 is out of scope of the on-board architecture's protocols. Rather protocols from Projects SeveCom or SimTD will have to be used here. However the key used for verification of the ECC signature but be confidential to the HSM at the senders side and thereby identifying the sender as being a trustworthy vehicle that uses an Evita Architecture. This kind of signalling the trustworthyness will be the challenge for the integration of in-car, car2car and inter-manufacturer certification. For now, we assume an appropriate mechanism to be in place:

Communication among Vehicles 1 and 2: (Prop.4.5.6.1)

$$precede(ECCsign(HSM_{CCU_1}, warning, PrivK_a, sig(PrivK_a, warning)),\\ ECCverify(HSM_{CCU_2}, warning, sig(PrivK_a, warning), PubK_a))$$

4.5.7 Behaviour and Communication of Vehicle 2

For the internal behaviour of vehicle 2 it can similarly to vehicle one be shown that the HMI will only show warnings that were successfully verify at the CCU before. The key used for this shall be the Public Key of a trustworthy vehicle 1.

Internal Behaviour of Vehicle 2: (Prop.4.5.7.1)

$$precede(ECCverify(HSM_{CCU_2}, warning, sig(PrivK_a, warning), PubK_a), show(CPU_{HMI_2}, warning))$$

4.5.8 Integration of Partial Proofs

The derived properties covering parts of the functional path Prop.4.5.2.36, Prop.4.5.3.33, Prop.4.5.4.1, Prop.4.5.5.1, Prop.4.5.6.1 and Prop.4.5.7.1 can be combined using the transitivity of precede SeBB.4.1.4.2 to conclude that each warning on the HMI of Vehicle_2 is based on sensor data measured by the sensor of vehicle_1:

Final Property of 1st Proof Attempt: (Prop.4.5.8.1)

$$precede(sense(CPU_{Sensor_1}, data_{Sensor}), show(CPU_{HMI_2}, warning))$$

4.6 Analysis and Trust Reasoning

We have seen above that the functional property oy a *warning* to originate from actual $data_{Sensor}$ can be fulfilled under the assumptions of our example deployment. However as it was also already outlined in [13], this is not yet sufficient to fulfill the goals of the security requirements elicitation in [17]. ather, the assurance to the driver that this functional property is actually fulfilled is still missing.

Within our example deployment we can preliminarily reason about the chain of trust assumptions, assertions and assurances informally. Starting from the HMI the introduction of a PSK with the KeyMaster is a trust of this agent. However, it relies on the assertion by the manufacturer through the act of importing the PSK and its keyhandle into the HMI. A deeper investigation on this trust assertion would rely heavily on the way that these two agents are acquainted by the manufacturer. The same applies to all the other PSKs within both the vehicles.

Another challenge that is heavily deployment dependant is the assertions among the vehicles regarding the behaviours of the included ECUs as well as the assertions' assertions within the other vehicles. This is one of the major challenges to be handled in case of a deployment in the field. The certificates used in car2car up to date do not carry information about the underlying platforms. These PKI structures however would have to carry information about all this. It should also be noted that though trust (assertion) levels may be introduced in these certificates, the decision of whether or not to display warnings to subsequent vehicles is binary. Accordingly for each level introduced in the assertion phase among vehicles and manufacturers a decision algorithm has to be introduced as well, as it seems infeasable to display to the driver a level of confidence next to a warning message.

Remaining attacks would however include the possibility for an attacker to exchange the actuator in a functional chain directly (i.e. the HMI for this example). These kinds of evil-maid attacks are known from Trusted Computing and Attacks on BitLocker already. However in a Car scenario, the driver usually always has an authentication token (the key) that could also be used in reverse.

5 Conclusions

In this document we demonstrated the application of two complementary verification approaches.

In the *magnified view* three techniques have been used for the proof of security properties in EVITA cryptographic protocols:

- An approach based on the TURTLE profile, and relying on usual model-checking techniques for achieving the proofs.
- An approach based on the ProVerif environment, and relying on Horn clauses resolution.
- An approach based on a new profile named AVATAR, developed in the scope of EVITA, and aiming at taking the best of the two first approaches, i.e., using a high-level language (SysML) and relying on a powerful toolkit dedicated top the proof of security properties (ProVerif).

Using those approaches, attacks – or limitations – of protocols were found during the definition phase of those protocols (i.e., before protocols as defined in [18] were published). More specifically, the modeling phase made it possible to identify flaws regarding DoS attacks. For other security properties (e.g., confidentiality, authenticity), performed proofs make it possible to have a stronger trust in EVITA cryptographic protocols studied in this document. Indeed, even if flaws on authenticity properties were identified. But the analysis of traces leading to attacks are related only to the fact that the model does not contain the modeling of time stamping, and so, replay attacks can be performed.

During the *global compositional*'s functional validate we have see many assumptions that in part may have been unknown or underestimated in their importance. Central to those where

- *Scheduling of HSM Access* to avoid race-condition introduced impersonations.
- *Secure Storage*'s importance to almost every action because of the storage of key handles.
- *Software Properties* for their functional meaning but also regarding e.g. correct setting transportflags for key creation.

The example used here only covered a part of an exemplary deployment. For a real deployment it would have to be redone. However, for this kind of proofs a lot of overlaps will occur such that effort can be saved (see also the consolidation step in [13]).

In summary it was shown, however, that the EVITA architecture and protocols are capable to provide a basis for future real deployments of secure in-vehicle platforms.

References

[1] The CADP toolkit. http://www.inrialpes.fr/vasy/cadp.

[2] TTool, the TURTLE toolkit. http://labsoc.comelec.enst.fr/ttoolindexhtml.

[3] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 33–44, Portland, Oregon, January 2002. ACM Press.

[4] M. Abadi and A.D. Gordon. A calculus fo cryptographic protocols, the spi calculus. In *Information and Computation*, volume 148, pages 1–70. Academic Press, Inc., January 1999.

[5] L. Apvrille et al. TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. In *IEEE transactions on Software Engineering*, volume 30, pages 473–487, Jul 2004.

[6] L. Apvrille et al. A UML-based environment for system design space exploration. In *13th IEEE International Conference on Electronics, Circuits and Systems (ICECS'2006)*, Nice, France, Dec 2006.

[7] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.

[8] B. Blanchet. From secrecy to authenticity in security protocols. In M. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, Madrid, Spain, September 2002. Springer Verlag.

[9] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

[10] B. Blanchet. Proverif automatic cryptographic protocol verifier user manual. Technical report, CNRS, Département d'Informatique École Normale Supérieure, Paris, July 2010.

[11] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. *Theoretical Computer Science*, 333(1-2):67–90, March 2005. Special issue FoSSaCS'03.

[12] R. Grimm and P. Ochsenschläger. Binding Cooperation, A Formal Model for Electronic Commerce. *Computer Networks*, 37:171–193, 2001.

[13] C. Jouvray, A. Kung, M. Sall, A. Fuchs, S. Gürgens, and R. Rieke. Security and trust model. Technical Report Deliverable D3.1.2, EVITA Project, 2009.

[14] E. Kelling, M. Friedewald, T. Leimbach, M. Menzel, P. Säger, H. Seudié, and B. Weyl. Specification and evaluation of e-security relevant use cases. Technical Report Deliverable D2.1, EVITA Project, 2009.

[15] D. Knorreck, L. Apvrille, and P. de Saqui-Sannes. TEPE: A SysML language for timed-constrained property modeling and formal verification. November 2010.

[16] Information processing systems – Open Systems Interconnection – LOTOS: A formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807, 1989.

[17] A. Ruddle, D. Ward, B. Weyl, S. Idrees, Y. Roudier, M. Friedewald, T. Leimbach, A. Fuchs, S. Gürgens, O. Henniger, R. Rieke, M. Ritscher, H. Broberg, L. Apvrille, R. Pacalet, and G. Pedroza. Security requirements for automotive on-board networks based on dark-side scenarios. Technical Report Deliverable D2.3, EVITA Project, 2009.

[18] H. Schweppe, M. S. Idrees, Y. Roudier, B. Weyl, R. El Khayari, O. Henniger, D. Scheuermann, G. Pedroza, L. Apvrille, H. Seudié, H. Platzdasch, and M. Sall. Secure on-board protocols specification. Technical Report Deliverable D3.3, EVITA Project, 2010.

[19] H. Seudié, J. Shokrollahi, B. Weyl, A. Keil, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. El Khayari, O. Henniger, D. Scheuermann, L. Apvrille, and G. Pedroza. Secure on-board architecture specification. Technical Report Deliverable D3.2, EVITA Project, 2010.