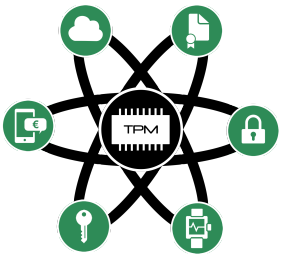

FutureTPM

D3.5

Final Report on the Design and Security of the QR TPM

Project number:	779391
Project acronym:	**FutureTPM**
Project title:	Future Proofing the Connected World: A Quantum-Resistant Trusted Platform Module
Project Start Date:	1st January, 2018
Duration:	36 months
Programme:	H2020-DS-LEIT-2017
Deliverable Type:	Report
Reference Number:	DS-LEIT-779391 / D3.5 / v1.1
Workpackage:	WP 3
Due Date:	31st December, 2020
Actual Submission Date:	8th February, 2021
Responsible Organisation:	SUR
Editor:	Georgios Fotiadis , José Moreira Kaitai Liang
Dissemination Level:	PU
Revision:	v1.1
Abstract:	In this report, we put forth the final models towards verifying the security properties of the remote attestation service, as leveraged in the context of one of the envisioned FutureTPM use cases; namely the Device Management where the focus is on the secure identification and management of network devices. The produced models are based on the *ideal functionalities* of TPM commands that have been defined through appropriate abstractions towards formally verifying the security properties of the executed protocols.
Keywords:	TPM Modelling & Abstraction, Formal Verification, Tamarin prover

The project FutureTPM has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779391.

Editor

Georgios Fotiadis (UL), José Moreira (UB)
Kaitai Liang (SURREY)

Contributors (ordered according to beneficiary numbers)

Kaitai Liang, Liqun Chen (SURREY)
José Moreira (UB)
Georgios Fotiadis (UL)
Roberto Sassu (HWDU)
Thanassis Giannetsos (DTU)

Disclaimer

Executive Summary

In this Deliverable, we complete the modelling and formal verification of one of the core services towards building chains of trust based on the secure identification and correct configuration state of deployed devices. More specifically, we formalize the notion of *secure remote attestation* and all its relevant functionalities such as the *creation of TPM keys, the Enhanced Authorization (EA) mechanism, and the management of sessions and Platform Configuration Registers (PCRs)*. We consider the modelling and formal verification of these TPM building blocks in the context of the Device Management use case as a starting point for the security modelling of the entire TPM platform towards supporting **trust-aware service graph chains with verifiable evidence on the integrity assurance and correctness of the comprised devices**. This break down of TPM services allows for a more effective verification process towards building a global picture of the entire TPM platform security modelling as a Root-of-Trust.

In the context of Device Management, the main objective is to establish a secure TLS communication channel between a Router and a Network Management System (NMS) as part of a network infrastructure. This communication channel is established in three phases where the creation of all necessary cryptographic keys is managed: In the first phase, the Router creates an Attestation Key (AK), by leveraging the attached TPM, which is certified by a Remote Attestation (RA) Server. In the second phase, the Router creates a TLS key, via the TPM, which is certified by the RA Server and signed by the NMS. Finally, the third phase is focused on the establishment of the secure communication channel between the Router and the NMS by leveraging the previously generated secret keys.

We present our models for all these phases, based on the **ideal functionalities** and **trusted platform command abstractions** that have been developed in Deliverables D3.3 and D3.4. The goal is to successfully verify specific security properties of interest in the Device Management model, using Tamarin lemmas. These security properties (as part of the overall *integrity, confidentiality, and secure measurements* requirements) include the necessary **sanity checks** proving that the model execute correctly, in that it reaches all possible branches, the **availability of keys and certificates** at all honest parties, the **freshness and secrecy of the created keys**, and the **authenticity of the messages and values** that each party receives. We also put forth a number of challenges that were encountered during this modelling and verification process and the actions taken in order to overcome them.

Finally, we provide evidence that this “bottom-up” modelling approach, followed in the context of the remote attestation service, can be extended to other application domains with such strict security and privacy requirements as envisioned (for instance) by the other two FutureTPM use cases in the fields of Fintech and Assistive Healthcare. This is due to the fact that the set of TPM commands currently modelled constitute the common denominator considered in most scenarios leveraging the TPM as a decentralized Root-of-Trust, including also the advanced Direct Anonymous Attestation (DAA) protocol; as the trust anchor towards enhanced *privacy preservation* and *user-controlled anonymity and unlinkability*. **Overall, we believe that the produced models can provide the baseline for an extensible verification methodology that enables rigorous reasoning about the security properties of *Future TPMs***.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the main objectives of Work Package 3 (WP3) is the security modelling of the Trusted Platform Module (TPM) and the formal verification of its security properties. Recall from Deliverables D3.3 [10] and D3.4 [11] that the adopted methodology follows a "bottom-up" approach: The primary focus is on the modelling of TPM trust and security provided by a specific set of core TPM functionalities, leveraged by the majority of applications that make use of the TPM, before extending such models for reasoning about the TPM platform as a whole. In other words, we consider those **TPM functionalities that are crucial in establishing chains of trust in heterogeneous environments**—with the FutureTPM use cases standing as applications of particular interest. The intuition is that the provided models can serve as both a basis for reasoning about the **security of a wide set of applications and systems that make use of the TPM** and for reasoning about the **security of the TPM's mechanisms** themselves.

One such core functionality of interest is secure remote attestation, which can be performed via the advanced Direct Anonymous Attestation (DAA) protocol, or through the usage of a Privacy CA (PCA), as is the case of the currently standardized IBM remote attestation protocol [13]. Notably, secure remote attestation is one of the most popular services provided by the TPM and we have, therefore, chosen to base our modelling on this TPM functionality. Furthermore, in order to model this service, we also need to consider additional TPM processes, such as the creation of TPM keys, the Enhanced Authorization (EA) mechanism for authorizing key usage, the management of the Platform Configuration Registers (PCRs) and the creation and management of policy sessions.

In Deliverable D3.4 [11], we also described the reasoning behind selecting the device management use case as a starting point for applying our modelling methodology as it leverages an enhanced version of remote attestation functionalities towards the creation of secure TLS communication channels. The underpinnings of this use case mode of operation are described in detail in Deliverables D4.1 [8], D6.1 [7] and D6.3 [9].

Trusted Platform Command Abstractions. One of the most demanding parts in our approach is the modelling of the TPM platform and specifically, the modelling of the internal TPM commands. For this purpose, we have introduced in D3.3 [10] the notion of an *idealized functionality*. This is a model of a TPM command that captures the actions of the trusted platform module when the command is executed in such a way that it excludes the cryptographic operations carried out internally (e.g., hash functions, signature creation, encryption) and replaces them with non-cryptographic approaches. Such non-cryptographic mechanisms include the use of an equational theory, a Trusted Third Party (TTP) or involving private channels to model the communication between parties in a secure manner. We essentially developed a **trusted abstract platform model consisting of a specific set of formally-specified primitives sufficient to implement the**

core TPM functionalities beyond the core crypto operations. Such an abstraction modelling can enable the reasoning about and comparing different TPM services under various adversarial models and for different security guarantees, excluding any possible implications from the leveraged cryptographic primitives. For trusted platform module implementers, such a representation can be considered as a golden model for the expected system behaviour. From the perspective of formally verifying trusted hardware components, this model can provide a means of reasoning about security and privacy (of offered services) without being bogged down by the intricacies of various crypto primitives considered in the different platforms.

1.1 Methodology

The device management use case consists of three main entities: a (set of) Routers equipped with a TPM (each Router and TPM forms a Platform), a Remote Attestation (RA) Server, and a Network Management System (NMS), which is responsible for creating routing policies based on the integrity status of the routers. In order to receive the integrity status, the NMS establishes a secure TLS channel with the Router. Prior to that, the Router creates two different signing keys using the TPM, an Attestation Key (AK) and a TLS key. The former is certified by the RA Server while the later by the NMS (more precisely, it is certified by the RA Server and it is signed by the NMS). The AK certification is achieved through the use of a PCA-based protocol, namely the IBM remote attestation protocol [13], whereas the TLS certification is achieved by using the Subject Key Attestation Evidence (SKAE) certificate extension. The TLS key will be used by the Router towards establishing the secure communication with the NMS, leveraging the TLS1.3 protocol, and the AK will be used in order to sign produced quotes reflecting on the configuration integrity and correctness of the deployed routers. Consequently, the secure enrolment process (for each new Router) and its communication with the NMS, can be abstracted in the following three main phases:

1. The Router creates an AK, using the TPM and the RA Server certifies this AK via the IBM remote attestation protocol [13].
2. The Router creates a TLS key, using the TPM, and the RA Server together with the NMS certify this TLS key using the previously generated and certified AK.
3. The NMS is able to establish a secure TLS channel with the Router so that it can send attestation reports, including signed quotes, via a confidential channel.

In Deliverable D3.4 [11], we have presented the compiled model for the first phase; the creation of the AK and its certification by the RA Server. In addition, we have also presented the necessary model for the creation of the TLS key.

Continuing with the modelling process, there are two objectives for the remainder of this deliverable. The first is to present the finalized models for the second and third phases on the **TLS certification and the establishment of the TLS communication channel, respectively**. The second is to identify, model and formally verify the security properties for all the aforementioned three phases. The adopted methodology consists of the following steps:

Step 1: ***Determine the TPM commands.*** The first step is to identify the TPM commands that are used in the scenario we wish to model. The motivation is to present an abstract description of the TPM commands, or in other words, a high-level description of each TPM command that excludes the technical details presented in the TPM specification manual [24] which

are not relevant for the modelling. This will result in a better understanding of the core operations and actions of each TPM command and will allow us to model each command in such a way that our ideal functionalities are as close to the real commands as possible.

Step 2: ***Define the ideal functionalities.*** We describe the ideal functionality for each TPM command that is identified in Step 1, based on its abstract description. That is, we present a model for each TPM command which replaces the cryptographic operations that are carried out internally by the TPM with non-cryptographic approaches. In order to do this, we need to have a specific formal verification tool in mind and in our case, we have chosen the Stateful Applied Pi Calculus (SAPiC) tool. The set of the ideal functionalities constitutes the model for the TPM process.

Step 3: ***Model the remaining components.*** This is the step where we model the remaining entities and processes of the protocol. In our case, these are the **Router**, which interacts with the TPM, and the **RA Server plus the NMS** that will be modelled as a single process. This part of the model captures the interaction of the TPM and the Router with the outside world. It includes the process of creating the AK and TLS keys, their certification and the establishment of the secure channel between the Router and the NMS. Further, we have to highlight that we have treated the modelling of each phase independently, rather than considering one model that captures the intricacies of all as a whole ("bottom-up" approach). The reasoning behind this is to reduce the complexity of the compiled models, as well as to minimize the risk of running into unexpected behavior during the verification process, mainly non-termination issues as will be described in Section 4.

Step 4: ***Formal verification of security properties.*** The last task of our modelling process revolves around the actual verification of the target security properties that our model should satisfy. Such properties are modelled in the form of Tamarin lemmas [3] and the focus was on obtaining completely mechanized proofs; see Chapter 4 below. Examples of such security properties include the necessary sanity check lemmas that prove the correct execution and termination of the model (i.e., that it reaches all possible branches), the availability of the created keys and their corresponding AK and TLS certificates at all honest parties, the freshness and secrecy of the created TPM keys, and the authenticity of the messages and values that each party receives.

It is important to highlight that such a "bottom-up" modelling approach is certainly a general process that is applicable for modelling any TPM-based scenario and it is not solely specific to the device management scenario, or the use cases of this project. We argue that such a verification methodology, based on the use of trusted abstract platform models and idealized functionalities, is more than just a set of proofs of correctness of specific services (e.g., secure remote attestation) but it can also enable the security modelling of the TPM as a whole merging various functionalities offered by the different abstraction layers. The trusted abstract platform model can serve as a specification of primitives of TPM operation, and is designed to be extensible towards additional features (as presented in Chapter 5) and additional guarantees against sophisticated attackers. Based on our findings, we also posit open issues and challenges towards this generalization, and discuss possible ways to address them, so as this type of formally verified security modelling can act as an enabler for the further enactment of trusted computing technologies.

1.2 Structure of the Report

In Chapter 2 we provide a high-level description of the device management use case and the three phases to be modelled. In Chapter 3 we present our modelling choices and the complete models in SAPiC for these three phases: AK creation and certification, TLS key creation and certification and establishment of TLS communication. Chapter 4 deals with the description of the security properties (Tamarin lemmas) and their formal verification, using Tamarin. Finally, in Chapter 5 we discuss the extension of our model to the two additional use cases of the project and beyond and present ideas on how our modelling efforts can be extended for future work. Finally, we present the conclusions in Chapter 6.

Chapter 2

FutureTPM Device Management Use Case

The purpose of Chapter 2 is to present an overview of the device management use case in order to obtain suitable abstractions of the messages exchanged between the involved parties and to have a clear view of the functionalities that we wish to model. Recall that the devices encountered in the ecosystem of the device management scenario are a set of Routers, the RA Server and the NMS. The use case assumes that each Router is equipped with a (quantum-resistant) TPM which is used in order to perform specific cryptographic tasks, such as creating keys, quotes and managing Certificate Signing Requests (CSRs). Further, each Router also contains a series of components and libraries, e.g., the Zero Touch Provisioning (ZTP) Agent and the RA Client. The ZTP Agent is responsible for initiating certification of the AK and the TLS keys and it is also responsible for responding to remote attestation requests by the NMS. The RA Client is the component that interacts with the TPM. On the other hand, the RA Server contains the RA Lib, which is a library that is responsible for the enrolment of each Router and for the verification of the CSRs and quotes that are received from the Router. In what follows, we will abstract all those implementation details in our discussion.

Recall also that the scenario that we intend to model consists of three phases: the AK creation and certification, the TLS key creation and certification and the establishment of the secure communication channel between the Router and the NMS. We give an abstract description of three phases in the following sections.

Notation. We introduce the following notation that will be used throughout this chapter. Given a key pair $k = (k_{priv}, k_{pub})$, we denote by $cert_P(k_{pub})$ the certificate for the public key k_{pub}, signed by the entity P, with private signing key P_{priv}:

$$cert_P(k_{pub}) = (k_{pub}, sign(k_{pub}, P_{priv})),$$

where $sign()$ is a signature algorithm. We note that k_{pub} can also be a tuple containing some additional information, e.g., $\langle k_{pub}, \text{info} \rangle$. Following the TPM specification, we model the name of a key k as the hash of its public part: $k_{name} = H(k_{pub})$, for some hash function $H()$. In addition, we denote $fqdn_P$ the Fully Qualified Domain Name (FQDN) that is a unique identifier for party P.

We also make the following assumptions: A endorsement key (EK) pair with handle ek_h is already loaded into the TPM, which we denote as $EK = (ek_{priv}, ek_{pub})$. The EK corresponds to an asymmetric encryption key pair and is trusted by the servers. Further, we consider that the NMS and the RA Server are equipped with a pair of signing keys, (NMS_{priv}, NMS_{pub}) and (RA_{priv}, RA_{pub}) respectively. Finally, we assume that the RA Server's public key is certified by the NMS, hence, $cert_{NMS}(\langle RA_{pub}, fqdn_{NMS} \rangle)$ is already created and publicly available.

2.1 Certification of the AK

The process of creating and certifying the AK is carried out between the Router, the TPM, the RA Server and the NMS. The interaction of these parties and the exchanged messages are depicted in Figure 2.1. For a more complete analysis, we refer the reader to Deliverable D6.3 [9], in which the device management demonstrator is described.

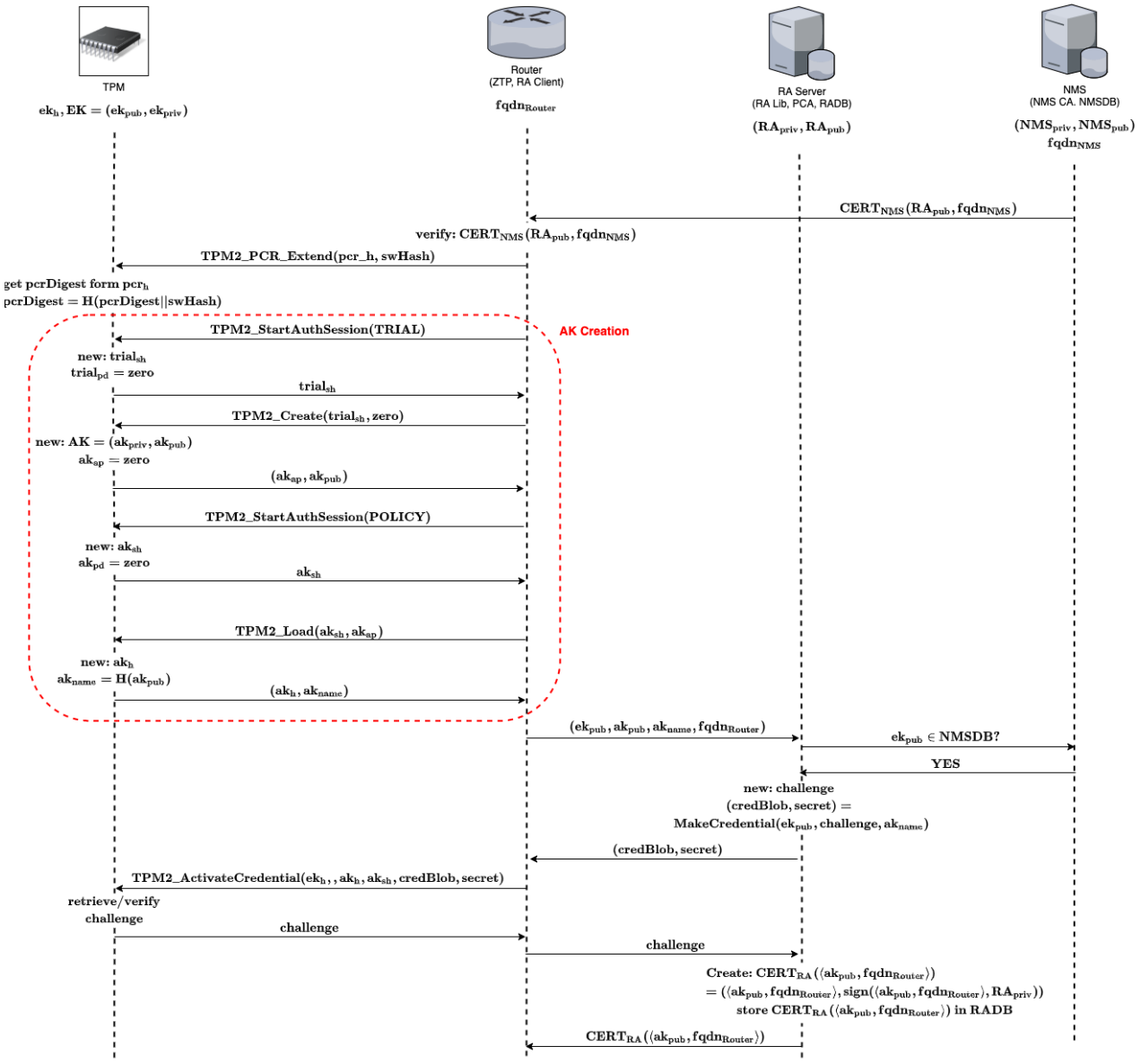

Figure 2.1: Creation of AK by the Router and certification by the RA Server

The first action of the Router is to extend a specific PCR with its current system state. More concretely, the RA Client executes the command

$$\texttt{TPM2\_PCRExtend}(pcr_h, swstate),$$

where $swstate$ is a digest value that corresponds to the current software state of the Router. Then the TPM will update the PCR digest value that is referenced in the handle pcr_h as $pcrDigest = H(pcrDigest \parallel swstate)$, for some hash function $H()$. For the creation and certification of the AK, the following steps are executed between the Router, the TPM, the RA Server and the NMS:

1. *Obtain RA Server certificate:* The Router obtains the certificate $cert_{NMS}(RA_{pub}, fqdn_{NMS})$ of the RA Server, from the NMS.

2. *Extract $fqdn_{NMS}$ from certificate and begin the enrolment:* The Router extracts the value $fqdn_{NMS}$ from the certificate.

3. *Create AK key:* The Router creates the AK for signing, using the TPM. Recall from Deliverable D3.4 [11] that this AK will not be linked to any policy. This implies that the authorization policy of the key will be zero. The AK creation is presented in the red frame in Figure 2.1. In particular the Router executes the command `TPM2_StartAuthSession(TRIAL)` in order to initiate a trial session. The TPM creates a fresh session handle $trial_{sh}$ and initializes the policy digest $trial_{pd}$ of this session to zero. Then the TPM sends the handle $trial_{sh}$ back to the Router. The Router executes the command `TPM2_Create`$(trial_{sh}, zero)$ in order to create the AK pair. The TPM sets the authorization policy ak_{ap} of the AK to zero, it creates the key pair $AK = (ak_{priv}, ak_{pub} = pk(ak_{priv}))$ and returns (ak_{ap}, ak_{pub}) to the Router.

   In order for the Router to load the key to the TPM, it creates a policy session, by executing the command `TPM2_StartAuthSession(POLICY)`. The TPM creates a fresh session handle ak_{sh}, it initializes the policy digest ak_{pd} of this session to zero and sends to the Router the session handle ak_{sh}. The Router then executes the command `TPM2_Load`(ak_{sh}, ak_{ap}) in order to load the key in the TPM. The TPM creates the handle ak_h for the AK, as well as its name $ak_{name} = H(ak_{pub})$ and returns both (ak_h, ak_{name}) to the Router. More details on the abstract description of the TPM commands that are used in this step, as well as a detailed description of the creation of TPM keys, are given in Deliverable D3.3 [10, Section 2.2].

4. *Get AK cert:* The Router asks the RA Server to issue a certificate for the AK it generated. For this purpose, it sends to the RA Server the public parts ek_{pub} and ak_{pub} of the EK and AK, the name ak_{name} of the AK and the $fqdn_{Router}$. This initiates the enrolment process.

5. *Check if Router EK credential is in NMSDB:* The RA Server asks the NMS if the EK credential of the Router requesting an AK certificate has been added to the NMSDB (the Router is already enrolled); this prevents any Router from getting an AK certificate. The NMS checks the EK credential against the entries in its NMSDB and responds accordingly.

6. *Generate credential blob and verify challenge response by Router:* The RA Server asks the Router to prove that it possesses the EK. This is accomplished through the IBM remote attestation protocol. In particular, the RA Server creates a fresh value $challenge$ and executes the TPM command

   $$\texttt{TPM2\_MakeCredential}(ek_{pub}, challenge, ak_{name}),$$

   which outputs the pair $(credBlob, secret)$ [1]. The value $credBlob$ is composed of the encryption of the $challenge$ with a symmetric key generated using a seed value and an HMAC value for this encryption with a key also generated by the seed value. The value $secret$ is essentially the encryption of the seed value using the public part of the endorsement key ek_{pub} (see Deliverable D3.4 [11, Figure 2.2], for more details on the abstract description

[1] We note here that the RA Server does not have access to the Router's TPM. The execution of the command `TPM2_MakeCredential` can be either done using a virtual TPM, or it can be done by simply executing the same instructions and operations of this command.

of the TPM command `TPM2_MakeCredential`). The pair $(credBlob, secret)$ is sent to the Router. The proof is derived from the Router, by executing the command

$$\texttt{TPM2\_ActivateCredential}(ek_h, ak_h, ak_{sh}, credBlob, secret),$$

which retrieves the $challenge$ value after performing the necessary validation checks and sends it to the RA Server (see Deliverable D3.4 [11, Figure 2.3], for the abstract description of the TPM command `TPM2_ActivateCredential`).

7. *Generate AK certificate and send it to Privacy CA:* The RA Server generates a certificate for the AK generated by the Router and asks Privacy CA in the RA Server to sign the certificate.

8. *Sign AK certificate:* The Privacy CA in the RA Server signs the AK and the RA Server sends the AK certificate to the Router. This is represented by $cert_{RA}(\langle ak_{pub}, fqdn_{Router}\rangle)$.

9. *Store AK certificate:* RA Server stores the signed AK certificate in the RADB.

After the successful completion of the above steps, both the Router and the RA Server have the certificate $cert_{RA}(\langle ak_{pub}, fqdn_{Router}\rangle)$ for the AK that is created by the Router using the TPM. The certified AK will be then used in the next step, in particular in order for the RA Server to certify the TLS key that will be created by the Router.

The commands that are needed in the AK certification phase are, initially `TPM2_PCRExtend` and then the commands `TPM2_StartAuthSession`, `TPM2_Create`, `TPM2_Load`, for creating and loading the AK, and `TPM2_MakeCredential`, `TPM2_ActivateCredential` for creating and activating a credential. The abstract description of these commands is presented in Deliverables D3.3 [10], for the first three and [11] for the credential-related commands. Their modelling as ideal functionalities, as well as the modelling of the certification of the AK were the main subjects of Deliverable D3.4 [11]. A recap of this model will be presented in Section 3.2

2.2 Certification of the TLS key

The second step towards the modelling of the device management use case is the creation of a TLS key by the Router and its certification, using the previously generated AK key. Again, this step requires the interaction between the Router and the TPM, the RA Server and the NMS. The exchanged messages between these parties are described in Figure 2.2. For additional information and a more detailed description we again refer the reader to Deliverable D6.3 [9].

We give a detailed description of the TLS certification procedure, since the modelling of this step is one of the innovations considered also in this deliverable. Recall from D3.4 [11] that unlike the AK, the TLS key will be linked to a PCR policy. That is, the PCR value that was extended in the AK certification process, will be used in order to update the policy digest of the policy session associated to the TLS key. Precisely, the following steps are executed for the TLS certification, which follow the description in Deliverable D6.3 [9].

1. *Generate TLS key and CSR with SKAE:* The Router creates a TLS signing key, using the TPM and this TLS key is linked with a PCR policy. The process of creating a TPM key that is bound to a PCR policy is presented in detail in Deliverable D3.3 [10, Chapter 2, Figure 2.11]. In is also described in the red frame in Figure 2.2. In brief, the Router executes the command `TPM2_StartAuthSession(TRIAL)` in order to create a trial session with handle

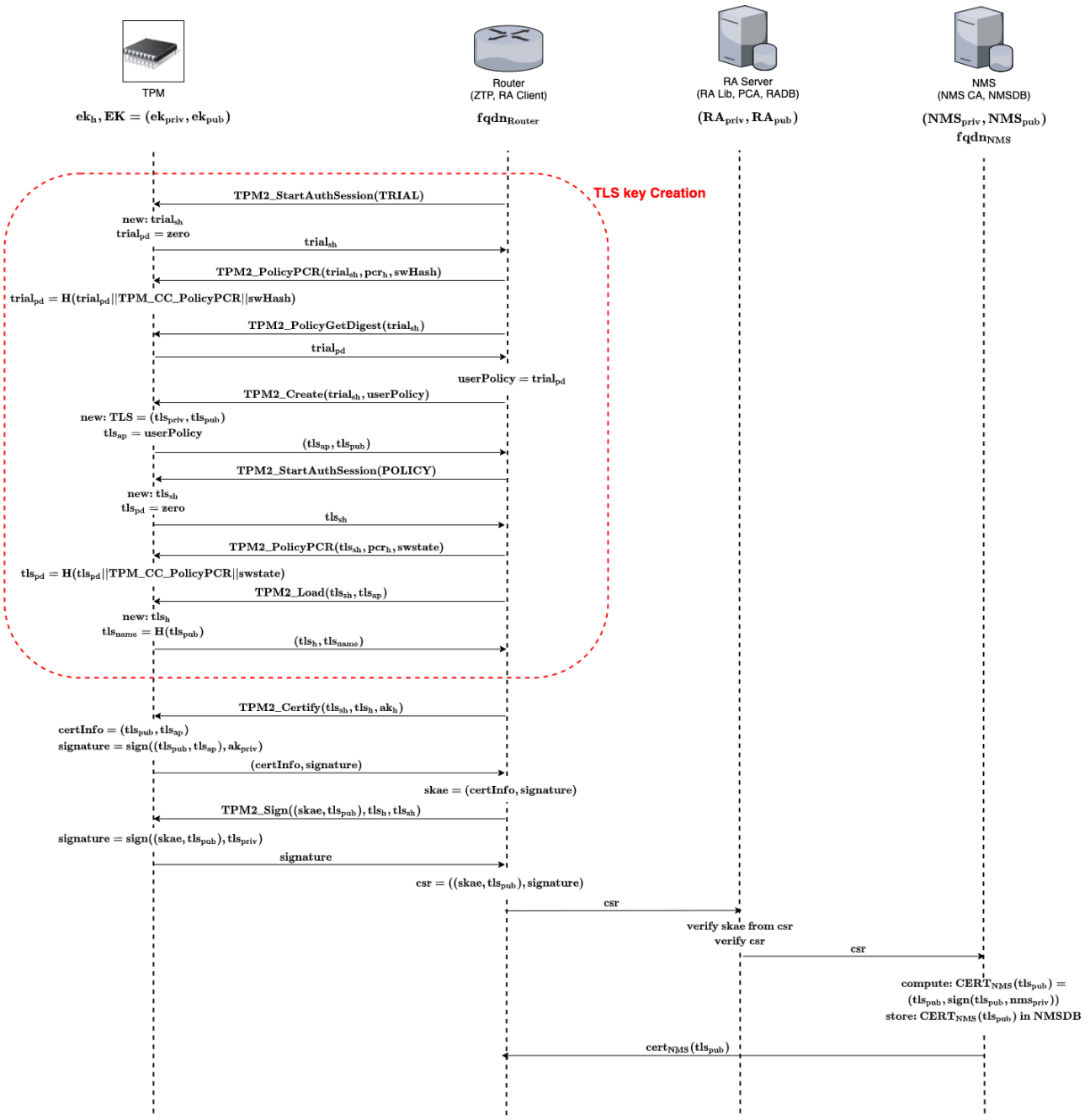

Figure 2.2: Creation of TLS key by the Router and certification by the RA Server & NMS

$trial_{sh}$ and initializes its policy digest $trial_{pd}$ to zero. The TPM returns the handle of the session to the Router, which executes the command `TPM2_PolicyPCR`$(trial_{sh}, pcr_h, v)$, with input the handle of the session, the handle of the PCR and the digest value $v = (pcrDigest \parallel swstate)$, which is the expected PCR value that was updated using the `TPM2_PCRExtend` command. The TPM updates the policy digest of the session according to the relation

$$trial_{pd} = H(trial_{pd} \parallel \text{'TPM\_CC\_PolicyPCR'} \parallel v),$$

where $H()$ is a hash function. The Router executes `TPM2_PolicyGetDigest`$(trial_{sh})$ in order to obtain the policy digest $trial_{pd}$ and uses this value to create the TLS key $TLS = (tls_{priv}, tls_{pub})$, via the command `TPM2_Create`$(trial_{sh}, trial_{pd})$. The TPM sets the authorization policy $tls_{ap} = trial_{pd}$ and returns the authorization policy tls_{ap} and the public

part tls_{pub} of the TLS key to the Router. Now the Router creates a policy session with `TPM2_StartAuthSession(POLICY)` and the TPM sets $tls_{pd} = zero$ and returns the session handle tls_{sh} to the Router. The Router executes `TPM2_PolicyPCR`(tls_{sh}, pcr_h, v) in order to update tls_{pd} and then `TPM2_Load`(tls_{sh}, tls_{ap}) and obtains the handle and name of the TLS key: (tls_h, tls_{name}). The process of creating the TLS key is presented in contained in the red box in Figure 2.2.

Further, the Router needs to create the Subject Key Attestation Evidence (SKAE). The SKAE consists of the public part of the TLS key plus its authorization policy and the signature of both using the private part of the AK (see [14] for more information). In other words

$$skae = (\langle tls_{pub}, tls_{ap}\rangle, sign(\langle tls_{pub}, tls_{ap}\rangle, ak_{priv})).$$

In order to create the SKAE, the Router executes the command

$$\texttt{TPM2\_Certify}(tls_{sh}, tls_h, ak_h).$$

The abstract description of the command `TPM2_Certify` is presented in Deliverable D3.4 [11, Figure 2.1]. The output of this command is the pair $(certInfo, signature)$, where $certInfo$ is the value to be certified, in our case $\langle tls_{pub}, tls_{ap}\rangle$ and

$$signature = sign(\langle tls_{pub}, tls_{ap}\rangle, ak_{priv}).$$

Note that the signature on $\langle tls_{pub}, tls_{ap}\rangle$ is created using the private part of the AK. The pair $(certInfo, signature)$ is returned to the Router which creates the SKAE value $skae = (certInfo, signature)$.

Now the Router creates a Certification Signing Request (CSR) for the TLS key. The CSR is composed of the message $(skae, fqdn_{NMS}, tls_{pub})$ and the signature on that message using the private part of the TLS key. For the creation of the signature, the Router executes the command

$$\texttt{TPM2\_Sign}((skae, fqdn_{NMS}, tls_{pub}), tls_{sh}, tls_h),$$

with input the message to be signed $(skae, fqdn_{NMS}, tls_{pub})$, the handle of the session tls_{sh} and the handle of the TLS key tls_h. The command `TPM2_Sign` is described in Deliverable D3.4 [11, Figure 2.4]. We note here that in order for the signing command to be executed, an authorization check is required for using the TLS key. In other words, the TPM checks whether the authorization policy of the TLS key tls_{ap} is equal to the policy digest of the session tls_{pd}. After the correct execution of the `TPM2_Sign` command, the Router creates the CSR as:

$$csr = [(skae, fqdn_{NMS}, tls_{pub}), sign((skae, fqdn_{NMS}, tls_{pub}), tls_{priv})].$$

2. *Get TLS key cert and begin the enrollment:* The Router asks the RA Server to issue a certificate for the Router TLS key. That is, the Router sends to the RA Server the csr that it created.

3. *Check if AK cert is in DB:* The RA Server, which acts as a verifier, checks if there is a certificate for the AK the Router used for signing the TLS key. Recall from the previous section that the RA Server stores the AK certificate $CERT_{RA}(\langle ak_{pub}, fqdn_{Router}\rangle)$ in its database RADB.

4. *Verify SKAE from CSR:* The RA Server, verifies that the message $(skae, fqdn_{NMS}, tls_{pub})$ is indeed signed by the private part of the TLS key. It also verifies the correct construction of the $skae$ value, in other words, it checks whether the public part of the TLS key is signed by the AK that is associated to this Router.

5. *Get signed TLS key cert from CSR:* The RA Server forwards the csr to the NMS, so that the NMS CA can sign it.

6. *Verify CSR has correct FQDN from EK cred:* The NMS verifies that the value $fqdn_{NMS}$ is correct.

7. *Sign TLS key cert:* The NMS signs the TLS key certificate. That is, it creates the certificate $CERT_{NMS}(tls_{pub})$; basically it signs the public part of the TLS key with its private singing key NMS_{priv}.

8. *Store TLS key cert in NMSDB:* The NMS stores the TLS key certificate of the Router in the NMSDB; the TLS key certificate is delivered to Router.

9. *Enrolment complete, received AK and TLS key cert:* The Router successfully receives the TLS key certificate.

After the successful completion of the above steps, both the Router and the NMS have the TLS certificate $CERT_{NMS}(tls_{pub})$.

2.3 Establishment of TLS Connection

Once both the AK and TLS certificates are created, the NMS is able to establish a secure communication channel with the Router. This phase requires the use of the TPM command `TPM2_Quote` in order to create and sign a quote via the TPM. We present here the abstract description of this command.

```
1  TPM2_Quote(signH, sesH, qData, inScheme, pcrSelect)
2  auth = authorization(signH, sesH)
3  if (auth = TRUE)
4      get ssk from signH
5      for i1,...,in in pcrSelect
6          create attestStruct = [PCR[i1]||...||PCR[in]]
7      quote = H(attestStruct||qData)
8      signature = sign(quote, ssk)
9      output(quote, signature)
10 else output(FAILURE)
```

Figure 2.3: The command `TPM2_Quote`

The command `TPM2_Quote` is used to quote selected PCR values. In other words, the TPM will create a digest value which is computed as the hash of the concatenation of specific PCR values, chosen by the caller and the resulting digest will be signed by the TPM. An abstract description of `TPM2_Quote` is given in Figure 2.3. It requires as input the handle `signH` of a signing key that will be used to sign the quote. In order to access the contents of the handle, enhanced authorization is required and so the handle of a session `sesH` is also present in the input. In addition, some external information `qData` and the signature scheme `inScheme` are also given as

input, along with a set of indices `pcrSelect`, which points to the PCRs that will be included in the quote. The TPM will compute the `quote` value as $\texttt{quote} = \texttt{H}(\texttt{PCR}[\texttt{i}_1] \parallel \ldots \texttt{PCR}[\texttt{i}_n] \parallel \texttt{qData})$, for $i_1, \ldots, i_n \in \texttt{pcrSelect}$. The command will output quote and the signature on that quote, using the secret signing key `ssk` referenced in `signH`, namely `signature = sign(quote, ssk)`. The command is described in Figure 2.3.

The establishment of the secure communication channel between the Router and the NMS involves the following steps:

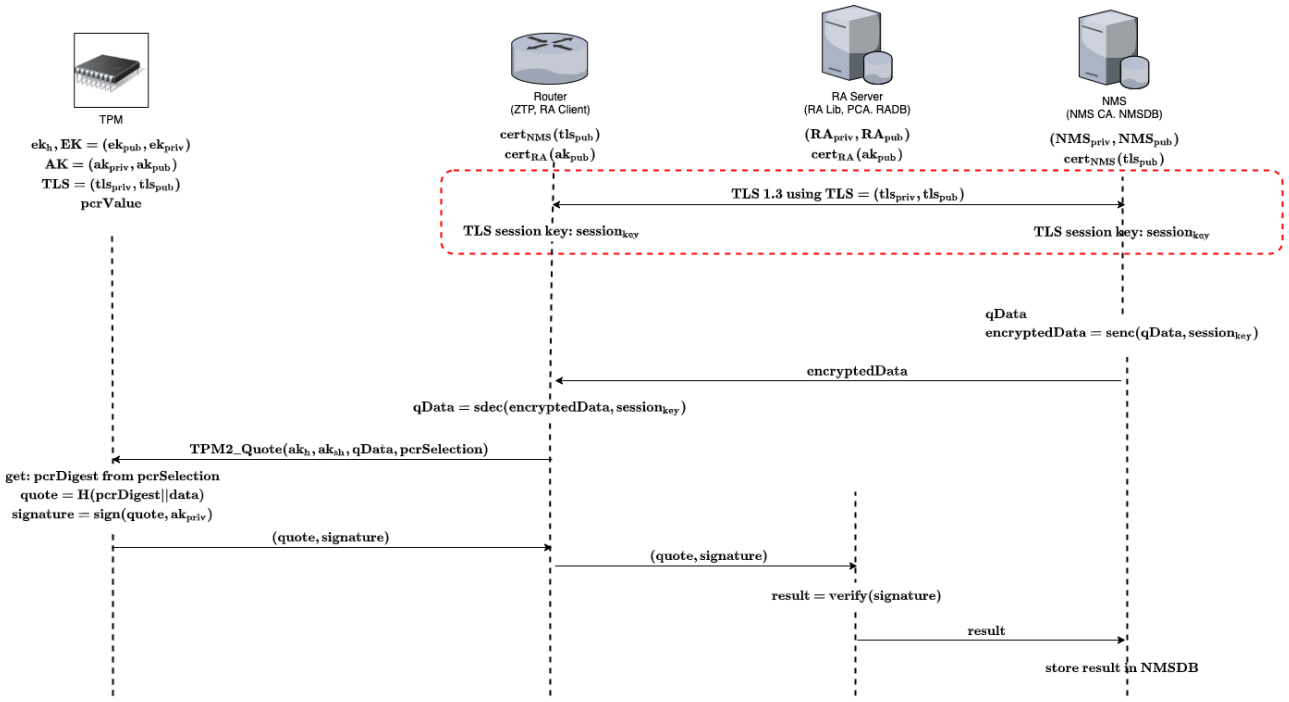

Figure 2.4: Establishment of TIS communication channel between the Router and the NMS

1. *Establish TLS connection:* The NMS establishes a TLS connection with managed routers. This is accomplished via the TLS1.3 protocol and using the TLS key $TLS = (tls_{priv}, tls_{pub})$ that was previously created by the Router. Modelling the interactions between the Router and the NMS in the context of the TLS1.3 protocol is outside the scope of this report and therefore we exclude the precise description of the TLS1.3 part. The result of this step is the establishment of a common session key $session_{key}$ between the Router and the NMS. This is a symmetric encryption key that will be used by these two parties for secure communication.

2. *Verify router TLS key cert:* The NMS verifies the TLS certificate by querying the NMSDB.

3. *Collect measurements and generate TPM quote:* The NMS encrypts some additional qualifying data $qData$ using the symmetric encryption key $session_{key}$ and sends the ciphertext c' to the Router. The Router executes the command

$$\texttt{TPM2\_Quote}(ak_h, ak_{sh}, qData, pcrSelection),$$

in order to create a signature on the quote $quote = H(pcrDigest \parallel qData)$, using the private part of the AK, where $pcrDigest$ is the digest contained in the PCR referenced in $pcrSelection$ and $H()$ is a hash function. The TPM returns the pair $(quote, signature)$ to the Router.

4. *Send measurements and TPM quote:* The Router forwards the pair $(quote, signature)$ to the RA Server .

5. *Check if AK cert is in DB:* The RA Server checks whether the quote it received has been signed by a TPM AK for which a certificate $CERT_{RA}(\langle ak_{pub}, fqdn_{Router}\rangle)$ was released by the RA Server.

6. *Verify measurements and TPM quote:* The RA Server (verifier) verifies the measurements and TPM quote sent by the Router. Essentially this implies the verification of the signature by the RA Server, using the public part ak_{pub} of the AK.

7. *Send verification result:* RA Server sends the result of router integrity verification to the NMS so that it can be seen by the Network Administrator.

8. *Store verification result:* The result of the Router integrity verification is stored in NMSDB.

Chapter 3

Security Modelling of Device Management Use Case

We advance to the main part of this report which is the security modelling of the device management scenario. More specifically, we describe our models for the three phases that were analyzed in Chapter 2: **certification of the AK**, **certification of the TLS key** and **establishment of TLS communication**. We start by reviewing our model for the certification of the AK key that was presented in Deliverable D3.4 [11]. We then proceed with the detailed description of the compiled models used for the creation of a TLS signing key by the Router and its certification by the RA Server and the NMS. Finally, we present the third part of the model, the establishment of a TLS communication channel between the Router and the NMS and the generation of attestation reports.

Modelling Notation. In our TPM-related models, we denote $\mathcal{F}_{\texttt{TPM2\_CommandName}}$ as the ideal functionality we are targeting, or equivalently our model for the TPM command `TPM2_CommandName`.

3.1 Overview of Modelling Tools, Approach and Challenges

Before we proceed with the description of our models, we summarize the adopted approach, the tools used, and the challenges encountered. This section is mainly a summary of observations and discussions from previous deliverables for completeness purposes.

3.1.1 Modelling Approach

Recall from Deliverable D3.4 [11] that our adversarial model is the usual Dolev-Yao model [12], in which the adversary is allowed to monitor and modify all exchanged messages between the processes. In order to keep our model as simple as possible, we consider three main processes: the TPM, the Router and the RA Server with the NMS acting as a single process, as shown in Figure 3.1. The components that are part of each entity, which are described in the previous chapter, are aggregated into these three processes as follows:

- Process Router = [Router, ZTP Agent, RA Client]
- Process RA and NMS Server = [RA Server, RA Lib, Privacy CA, RADB] and [NMS, CA, NMSDB]

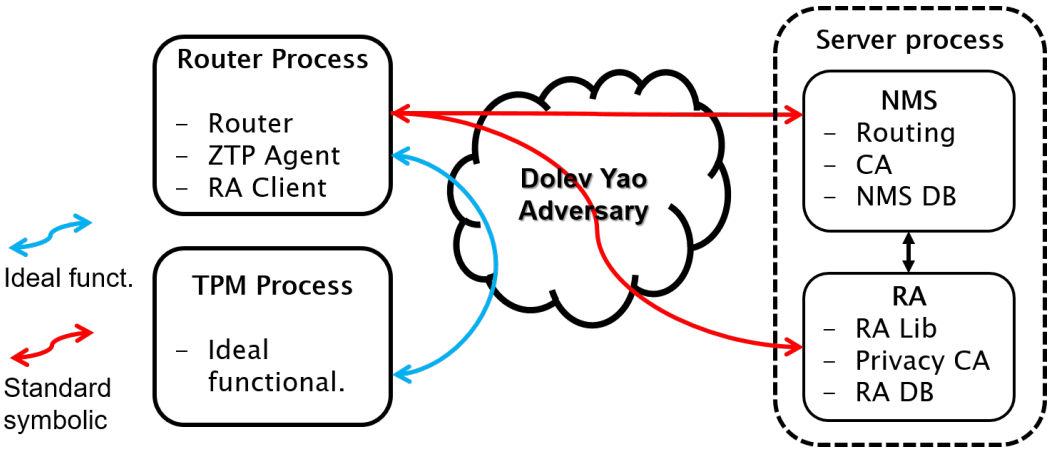

Figure 3.1: Adversarial model overview

- Process: TPM = [Set of ideal functionalities]

Our main goal, when defining the model for the device management use case, is to capture the communication between these three processes in a way that replicates the real-world interactions and modes of operation to the maximum possible extent. The reasoning behind aggregating the RA and NMS servers into a single process is based on the assumption that there is a trusted infrastructure supporting the interactions between these two processes. Hence it is not a viable target for an adversary. This includes not only the communication between these two servers, but also the access to their local databases or services.

As shown in Figure 3.1, the adversary is allowed to monitor and interfere in the communication between the Router and the servers according to the rules of the Dolev-Yao model. However, the model for the communication between the Router and the TPM needs to take into account the newly introduced concept of "ideal functionalities" (i.e., our models of the TPM commands), and it is considered independently. As discussed in Deliverable D3.4 [11], the ideal case would be to test for the envisioned security properties under the presence of the strongest adversary type that we can consider—hence, the adoption of the Dolev Yao adversary model. However, this is an overly powerful, unrealistic adversarial model in the real world: the adversary can intercept, drop, replay, etc. any message between any of the processes. In particular, the adversary can read, drop or send arbitrary commands to the TPM at any time point, and regard the attached TPM as a *TPM Oracle*. In order to create a model that allows a more realistic approach, we treat the channel between the Router and the TPM in a special way. For an initial approach, it could be considered that it is a completely private channel, i.e., the adversary does not have access to it. For relaxed trust assumptions (e.g., when the router has malware installed) we can allow the adversary certain degree of control over that channel (e.g., by making it a public channel with restrictions). However, this initial idea does not work well in practice, since modelling this channel as a synchronous, private channel in the applied pi-calculus lead to a number of issues that will be discussed below in Section 3.1.3.

3.1.2 Modelling Tools

As we have already mentioned in the introduction and also in Deliverable D3.4 [11], in order to better support the modelling of the device management use case and the verification and analysis of its security properties, we use the Tamarin prover and its front-end SAPiC [3, 15, 16, 19]. In

$\langle M, N \rangle ::= x, y, z \in \mathcal{V}$	variables
$\mid p \in PN$	public names
$\mid n \in FN$	fresh names
$\mid f(M_1, \ldots, M_n)$ s.t. $f \in \Sigma$ of arity n	function application
$\langle P, Q \rangle ::=$	processes
$\mid 0$	terminal (null) process
$\mid P \mid Q$	parallel execution of processes P and Q
$\mid !P$	replication of process P
$\mid \nu n; P$	binds n to a new fresh value in process P
$\mid \text{out}(M, N); P$	outputs message N to channel M
$\mid \text{in}(M, N); P$	inputs message N to channel M
$\mid \text{if } Pred \text{ then } P$ [else Q]	P if predicate $Pred$ holds; else Q
$\mid \text{event } F; P \quad F \in \mathcal{F}$	executes event (action fact) F
$\mid P + Q$	non-deterministic choice
$\mid \text{insert } M, N; P$	inserts N at memory cell M
$\mid \text{delete } M; P$	deletes content of memory cell M
$\mid \text{lookup } M \text{ as } x \text{ in } P$ [else Q]	if M exists, bind it to x in P; else Q
$\mid \text{lock } M; P$	gain exclusive access to cell M
$\mid \text{unlock } M; P$	waive exclusive access to cell M
$\mid [L] -[A]\rightarrow [R]; P \quad (L, R, A \in \mathcal{F}^*)$	provides access to Tamarin MSRs

Table 3.1: SAPiC syntax

Notation: $n \in FN, x \in \mathcal{V}, M, N \in \mathcal{T}, F \in \mathcal{F}$. As opposed to the applied pi-calculus [1], SAPiC's input construct $\text{in}(M, N); P$ performs pattern matching instead of variable binding.

particular, we developed our models using the SAPiC front-end, which allows us to define protocols in a calculus (similar to applied pi-calculus) rather than directly into Tamarin multiset rewrite rules (MSRs). Therefore, the front-end converts the processes specification into (labeled) MSRs to be analysed by Tamarin. The verification of the security properties is achieved by modelling these properties using Tamarin lemmas. *Tamarin is a state-of-the-art tool for symbolic verification and automated analysis of security properties in protocols, under the Dolev-Yao model [12], with respect to an unbounded number of sessions*. We refer to the Tamarin manual [3] for more information, as well as to Chapter 4 for the analysis of our Tamarin lemmas.

The reasoning behind using Tamarin and SAPiC for our modelling process is justified in Deliverable D3.4 [11, Section 4.1]. In summary, we opted out from considering other automated tools like ProVerif [4], or its extension StatVerif [2], as they are less suitable for modelling protocols with non-monotonic global state, e.g., protocols that "forget" information. However, it is rather challenging to model protocols with arbitrarily mutable global state, as it is required in the environment considered here. Another reason for choosing Tamarin and SAPiC is that there is already several existing works, such as the works by Shao et al. [21, 20], which have been successful in proving cryptographic properties for TPM functionalities.

For completeness, we briefly describe the main SAPiC syntax, depicted in Table 3.1. We also refer the reader to the Tamarin prover manual [3], to Deliverable D3.4 [11, Section 4.1], and to the original work by Kremer and Künnemann [16]. The calculus comprises an order-sorted term algebra with countably infinite sets of publicly known names PN, freshly generated names FN and variables $\mathcal{V}$. It also comprises a signature Σ, i.e., a set of function symbols, each with an arity. The messages are elements of a set of terms $\mathcal{T}$ over PN, FN and $\mathcal{V}$, built by applying the

function symbols in Σ.

The set of facts is defined as

$$\mathcal{F} = \{F(M_1, \ldots, M_k) \quad \text{s.t.} \quad M_1, \ldots, M_k \in \mathcal{T}, \quad F \in \Sigma \quad \text{of arity} \quad n\}.$$

The special fact $K(M)$ states that the term M is known to the adversary. For a set of roles, the Tamarin MSRs define how the system, i.e., protocol, can make a transition to a new state. An MSR is a triple of the form $[L] -[A]\rightarrow [R]$, where L and R are the premise and conclusion of the rule, respectively and A is a set of action facts, modeled by SAPiC events. For a process P the trace $\mathrm{Tr}(P) = [F_1, \ldots, F_k]$ is an ordered sequence of action facts generated by firing the rules in order.

Tamarin allows to express security properties as temporal, guarded first-order formulas, modelled as trace properties. The construct $F@i$ states that the fact F occurs (equivalently is true) at timepoint i. A property can be specified as a *lemma* or a *restriction*, depending if the property being verified or enforced [19]. We will discuss more on this in Chapter 4.

Everything that SAPiC does can be expressed in MSRs in Tamarin. However, compared to direct MSR encoding, modelling using SAPiC helps to develop a concise model that guarantees that the user cannot make mistakes in modeling state, concurrency, locks, progress, reliable channels, or isolated execution environments. For some of these, the encoding is likely more clever than ad-hoc modelling a user would come up with using MSRs. In this context, SAPiC has a better chance for termination.

3.1.3 Modelling Challenges

We discuss some of the main challenges that were encountered when working with the aforementioned tools. We note that, except for observational equivalence (which is out of the scope of our analyses), Tamarin is sound and complete, but it may not terminate, since the protocol verification problem is known to be undecidable. In that case, **user intervention is required in the form of auxiliary lemmas**.

Modelling the TPM-Router channel. Even though the model for this channel is simple conceptually, this has posed a significant drawback when trying to model it. As discussed above, the initial idea is that this channel should behave as a private channel if the router is not compromised, and it should allow some control to the adversary when the trust model is relaxed, e.g., making the channel public if we assume a full compromise. Of course, if the adversary has full control over the TPM and the communication channel with the host Router, then it is impossible to prove any meaningful security property, and hence we ignore this case.

In Deliverable D3.4 [11, Section 4.2] we discussed several alternatives to model the communication channel between the Router and the TPM. We further extend this discussion in order to reflect the conclusions obtained from the modelling process. Recall that our goal is to model a channel that behaves similarly to a private channel when there is no compromise between the router and the TPM. The final approach, therefore, has considered a combination of several strategies provided by SAPiC, since a single one has proven to be unsuccessful for our purposes. Therefore, our final model uses a combination of:

1. Usage of Tamarin restrictions to limit the capabilities of the adversary: This consists in using a template for sending/receiving messages to/from the TPM with events placed at the appropriate locations as follows:

```
//Template for sending a TPM command (Router process)
let pat_tpm_send_command = <TPM_CommandCode, param_1, ..., param_k> in
event TPM_SendCommand(pat_tpm_send_command);
out(pat_tpm_send_command);
P

//Template for receiving a TPM command (TPM process)
let pat_tpm_send_receive = <TPM_CommandCode, param_1, ..., param_k> in
in(pat_tpm_receive_command);
event TPM_SendCommand(pat_tpm_receive_command);
Q //TPM processess command here.
```

Moreover, we have to implement the following restriction on the execution traces, which will limit the adversary capabilities:

```
restriction RestrictionTpmCommand:
   "All c #i.TPM_ReceiveCommand(c)@i
     ==> ( (Ex #j.TPM_SendCommand(c)@j & (j < i))
          & not(Ex #k.TPM_ReceiveCommand(c)@k & not(#k = #i)))"
```

That is, we ensure that in order for a TPM to receive (and process) a message, an (injective) TPM call must have been executed in the Router process. Therefore, the above restriction will forbid the adversary from calling the TPM arbitrarily, unless the Router has first made a well-formed TPM call. This approach, however, gives the adversary the capability to eavesdrop and rearrange the messages sent to the TPM, which might not represent a realistic scenario.

2. Usage of the public channel: Even though we could use a similar strategy for the output of the TPM, whenever it is possible, we output the response of the TPM to the public channel. This means that, indeed, we are considering a more pessimistic scenario, since the adversary has full control on this output. However, since the TPM is a trusted component and the adversary does not have access to internal secrets such as the EK or private parts of the objects, this approach has worked well in many occasions. That is, in most cases, we can assume that the output of the TPM is available to the adversary to prove the security properties we are interested in.

3. Direct usage of multiset rewrite rules. The SAPiC calculus has an advanced (and often discouraged) feature which allows direct access to the multiset rewrite system of Tamarin. See Table 3.1 above. In order to emulate an asynchronous message transfer between the Router and the TPM, we use the following templates to make calls at the Router process and receive calls at the TPM process:

```
//Template for sending a TPM command (Router process)
let pat_tpm_send_command = <TPM_CommandCode, param_1, ..., param_k> in
[ ]--[ ]->[TpmCommandNameIn(pat_tpm_send_command)];
P

//Template for receiving a TPM command (TPM process)
let pat_tpm_send_receive = <TPM_CommandCode, param_1, ..., param_k> in
[TpmCommandNameIn(pat_tpm_send_command)]--[ ]->[ ];
Q //TPM processess command here.
```

> Note that the state fact `TpmCommandNameIn(pat_tpm_send_command)` is produced by the Router, consumed by the TPM process, and it is not availble to the adversary at any time point. A similar approach can be devised for the TPM outputs, i.e., the TPM produces a state fact that is consumed by the Router process.

As commented above, our model required a combination of these approaches to specific TPM calls in order to be able to prove the desired security properties and reach termination, and examples of those alternatives can be found in the code listings in Section 3.3 below. Clearly, if a security property holds in a channel specified with the strategies discussed above, then it will also hold for a private channel.

Partial deconstructions and non-termination We have also discussed in Deliverable D3.4 [11] that when modelling in Tamarin, it is sometimes the case that the tool encounters partial deconstructions left (also known as open chains), complicating the proof of a security property by either taking a very long time, or not terminating. One of the reasons that causes this malfunction is that when modelling certain protocols with complex interactions between the processes or persistent state, such as the device management use case, even if we don't explicitly model the secure communications between the parties involved, the protocols use cryptography to protect secrets that are exchanged. However, if the adversary forwards an encrypted, unknown secret to a process, then it can be the case that the process might decrypt or, in general, execute an operation unavailable to the adversary, and then output the result of this operation. If Tamarin identifies that a process outputs a processed message, then it cannot exclude the possibility that this message represents an "useful" knowledge for the adversary. Therefore, Tamarin concludes that such honest processes can be regarded as an oracle, e.g., a decryption oracle, and it might try to use it unsuccessfully to derive a key that it needs to decrypt another term.

Partial deconstructions occur in the pre-processing step of the verification of the protocol, when Tamarin tries to identify all the possible sources for all the state facts used in the protocol (see [18] for more details). As we will see in the next chapter, this happens in the AK certification process of the device management use case, particularly when using our models for the TPM commands `TPM2_MakeCredential` and `TPM2_ActivateCredential`. We tackle this problems by using a *sources lemma*, in order to guide the proof and help Tamarin terminate. More concretely, as discussed in Deliverable D3.4 [11], partial deconstructions occur in this case when the tool is unable to identify the origin of the fresh value for the $\sim$`challenge` in the PCA protocol for certifying the AK creation.

We remark that partial deconstructions and non-termination are two related but not directly dependent issues, i.e., one does not directly imply the other. Removing partial deconstructions often helps to achieve termination, but this is not always the case. Non-termination heavily depends on the efficiency of the heuristics, and it can sometimes change drastically even with minor modifications of the model. Therefore, we had to find the appropriate approaches in modelling the different components in order to achieve termination in the verification of the required security properties.

3.2 Recap of the AK Certification Model

In this section we present a brief overview of the work that was done in Deliverable D3.4 [11, Chapter 4] for the security modelling of the AK certification process in the device management use case, based on the abstraction described in Figure 2.1 above. This is the final version of

```
1  let main_process =
2    ( TPM || Router || RANMSServer )
3
4  //TPM objects
5  new ~ek_sk;
6  let ek_pk = pk(~ek_sk) in
7  out(ek_pk);
8
9  event SecretKey('EK', ~ek_sk);
10
11 new ~ek_h;
12 out(~ek_h);
13
14 //Router objects
15 new ~RouterID;
16 out(~RouterID);
17
18 //RA objects
19 new ~ServerID;
20 out(~ServerID);
21 new ~ssk_ra;
22 let spk_ra = pk(~ssk_ra) in
23 out(spk_ra);
24
25 event SecretKey('RAK', ~ssk_ra);
26
27 //NMS objects
28 new ~ssk_nms;
29 let spk_nms = pk(~ssk_nms) in
30 out(spk_nms);
31
32 event SecretKey('NMS', ~ssk_nms);
33
34 main_process
```

Figure 3.2: The main process for AK certification in SAPiC

the model for the AK certification in that it also contains the necessary events that will be used in Section 4.1 in order to capture the security properties that are relevant to this process.

Initialization and main process. In the device management use case, the EK is the parent key of the AK in an endorsement hierarchy. Recall from Deliverable D3.4 [11] modelling this key hierarchy is out of the scope of our models, and hence we assume that the EK pair, denoted $(\sim\texttt{ek\_sk}, \texttt{ek\_pk})$ and the EK handle $\sim\texttt{ek\_h}$ are generated as global variables and available to the corresponding parties. In addition, we also assume that the RA/NMS Server process has two pairs of signing keys $(\sim\texttt{ssk\_ra}, \texttt{spk\_ra})$ corresponding to the RA Server and $(\sim\texttt{ssk\_nms}, \texttt{spk\_nms})$ corresponding to the NMS. Given any fresh private key $\sim\texttt{k\_priv}$, we can generate its associated public key via the function application $\texttt{k\_pub} = \texttt{pk}(\sim\texttt{k\_priv})$.

The main process is depicted in Figure 3.2, and it is triggered at line 34. It essentially creates the private keys and the associated public keys available to all the processes and the adversary, a number of identifiers (RouterID and ServerID) used for convenience when defining the security properties, and it runs the TPM, Router and RA/NMS Server processes in parallel.

3.2.1 The TPM Process

Now, we describe our model for the TPM process. According to Figure 2.1, for the AK certification process, the Router requires four TPM commands: `TPM2_StartAuthSession`, `TPM2_Create`, `TPM2_Load` and `TPM2_ActivateCredential`. Actually, as we have mentioned above, there is a fifth TPM command that is executed by the Server, namely `TPM2_MakeCredential`. The ideal functionalities for these five TPM commands are presented in Deliverable D3.3 [10] and D3.4 [11]. Recall from Section 2.1 that the command `TPM2_MakeCredential` makes no use of secret information (i.e., it is only provided in the TPM specification for convenience) and is executed by the RA Server either using a software TPM, or by implementing the operations described in the TPM

```
let TPM =
  insert ⟨'authPolicy', ~ek_h⟩, nil;
  insert ⟨'privatePart', ~ek_h⟩, ~ek_sk;
  insert ⟨'publicPart', ~ek_h⟩, ek_pk;
  insert 'PCR', nil;

  !(
    //TPM2_Create
    (
      let pat_tpm_command =
        ⟨'TPM2_Create', authPolicy⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);
      new ~k_h;
      new ~k_sk;
      lock 'device';
      let k_pk = pk(~k_sk) in
      insert ⟨'authPolicy', ~k_h⟩, authPolicy;
      insert ⟨'privatePart', ~k_h⟩, ~k_sk;
      event SecretKey('AK', ~k_sk);
      insert ⟨'publicPart', ~k_h⟩, k_pk;
      out(⟨~k_h, k_pk⟩);
      unlock 'device'
    ) ||
    //TPM2_StartAuthSession
    (
      let pat_tpm_command =
        ⟨'TPM2_StartAuthSession'⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);
      new ~s_h;
      lock 'device';
      insert ⟨'policyDigest', ~s_h⟩, nil;
      out(~s_h);
      unlock 'device'
    ) ||
    //TPM2_ActivateCredential
    (
      let pat_tpm_command =
        ⟨'TPM2_ActivateCredential', a_h, a_sh,
        k_h, credBlob⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);

      lock 'device';

      lookup ⟨'policyDigest', a_sh⟩ as a_sh_pd in
      lookup ⟨'authPolicy', a_h⟩ as a_ap in

      if a_ap = a_sh_pd then
        lookup ⟨'publicPart', a_h⟩ as a_pk in
        lookup ⟨'privatePart', k_h⟩ as k_sk in

        if verifyCredential(a_pk, k_sk, credBlob) = true then
          let challenge =
            activateCredential(a_pk, k_sk, credBlob) in
          event Receive(challenge);
          [ ]--[ ]-> [Tpm2ActivateOut(challenge)];
          unlock 'device'
        else
          unlock 'device'
      else
        unlock 'device'
    )
  )
```

Figure 3.3: The TPM process for AK certification in SAPiC

specification [23, Section 24] (see also Deliverable D3.4 [11, Figure 2.2]), without involving a TPM. For this reason, we exclude this function from our model.

Recall also from Deliverable D3.4 [11], that we do not need to model the command `TPM2_Load`, since we can assume that the handle of the AK is returned by the `TPM2_Create` command. Therefore, our model for the TPM process in the AK certification consists of three ideal functionalities, namely: $\mathcal{F}_{\texttt{TPM2\_StartAuthSession}}$, $\mathcal{F}_{\texttt{TPM2\_Create}}$ and $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}$. The SAPiC code for the TPM process is presented in Figure 3.3. At the beginning of the TPM process, we initialize the authorization policy of the EK to zero, as well as its private and public parts. In addition, we also initialize the PCR value to zero. The initialization of the TPM process is followed by the three ideal functionalities, which are under replication and in parallel composition, in order to denote the fact that multiple calls are allowed to each method.

```
1  let Router =
2   (
3     new ~fqdn_router;
4     out(~fqdn_router);
5
6     new ~tid;
7     out(~tid);
8
9     let pat_cert_nms =
10      ⟨⟨spk_ra, fqdn_nms⟩, signature_cert_nms⟩ in
11    in(pat_cert_nms);
12    if verify(signature_cert_nms, ...) = true then
13    event HasKey('NMS', ~RouterID, spk_nms);
14
15    let pat_tpm_command1 =
16          ⟨'TPM2_Create', nil⟩ in
17    event TPM_SendCommand(...);
18    out(pat_tpm_command1);
19    in(⟨ak_h, ak_spk⟩);
20
21    out(⟨'RA_enrollrequest', ~fqdn_router,
22      ek_pk, ak_spk⟩);
23    in(⟨'RA_enrollrequest_resp', credentialBlob⟩);
24
25    let pat_tpm_command2 =
26          ⟨'TPM2_StartAuthSession'⟩ in
27    event TPM_SendCommand(...);
28    out(pat_tpm_command2);
29    in(ak_sh);
30    event GenerateAK(~tid, ak_spk);
31    event HasKey('EK', ~RouterID, ek_pk);
32
33    let pat_tpm_command3 =
34      ⟨'TPM2_ActivateCredential', ak_h, ak_sh,
35      ~ek_h, credBlob⟩ in
36    event TPM_SendCommand(pat_tpm_command3);
37    out(pat_tpm_command3);
38    [Tpm2ActivateOut(challenge)]--[]-⟩ [];
39
40    event RouterRunning(~RouterID, ~ServerID, challenge);
41
42    out(⟨'RA_enrollcert', challenge⟩);
43
44    let cert_ak =
45      ⟨⟨ak_spk, ~fqdn_router⟩, signature_cert_ak⟩ in
46    in(⟨'RA_enrollcert_resp', cert_ak⟩);
47
48    if verify(signature_cert_ak, ...) = true then
49      event RouterCommit(~RouterID, ~ServerID, cert_ak);
50      event ReceiveCertAK(~RouterID, cert_ak);
51      event HasKey('RA', ~RouterID, spk_ra);
52      event HasKey('AK', ~RouterID, ak_spk);
53      event RouterFinish()
54  )
```

Figure 3.4: The Router process for AK certification in SAPiC

3.2.2 The Router Process

Figure 3.4 describes the Router process model in SAPiC. This process is not under replication, as we are proving the security for a single Router, and we assume that each Router executes the enrollment phase only once. Each Router is associated to a unique (fresh) Fully Qualified Domain Name (FQDN) represented by the value ~`fqdn_router` at the beginning of the Router process, and a Router identifier, represented by the value ~`RouterID`, which is defined in the main process. The reason for defining the Router identifier is merely for convenience in expressing some of the security properties in a standard way, according to [17].

In lines 9–12, the Router receives the RA Server public key certificate and verifies it using the corresponding public key `spk_ra`. In lines 15–17, the Router calls the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Create}}$ in order to create the AK, and receives the AK handle `ak_h` and its public part `ak_spk` from the TPM. Then, the Router outputs the message:

$$\langle \text{`RA\_enrollrequest'}, \sim\texttt{fqdn\_router}, \texttt{ek\_pk}, \texttt{ak\_spk} \rangle,$$

in order to initiate the enrollment process through the PCA protocol. The response to this request is received from the RA Server, which is described by the message

$$\langle \text{`RA\_enrollrequest\_resp'}, \texttt{credentialBlob} \rangle.$$

The `credentialBlob` value is essentially the output of `TPM2_MakeCredential`, executed at the RA/NMS Server side. The Router calls the ideal functionality $\mathcal{F}_{\texttt{TPM2\_StartAuthSession}}$ in order to

```
1  let RANMSServer =
2   !(
3     new ~fqdn_nms;
4     out(~fqdn_nms);
5     let cert_ra = ⟨⟨spk_ra, ~fqdn_nms⟩, sign(⟨spk_ra, ~fqdn_nms⟩, ~ssk_nms)⟩ in
6     out(cert_ra);
7
8     in(⟨'RA_enrollrequest', fqdn_router, ek_pk, ak_spk⟩);
9
10    new ~challenge;
11    event Source(~challenge);
12    let credentialBlob = makeCredential(ek_pk, ~challenge, ak_spk) in
13    out(⟨'RA_enrollrequest_resp', credentialBlob⟩);
14
15    in(⟨'RA_enrollcert', ~challenge⟩);
16
17    let cert_ak = ⟨⟨ak_spk, fqdn_router⟩, sign(⟨ak_spk, fqdn_router⟩, ~ssk_ra)⟩ in
18
19    event GenerateCertAK(~ServerID, cert_ak);
20    event ServerRunning(~ServerID, ~RouterID, cert_ak);
21    event ServerCommit(~ServerID, ~RouterID, ~challenge);
22    out(⟨'RA_enrollcert_resp', cert_ak⟩);
23
24    event HasKey('AK', ~ServerID, ak_spk);
25    event HasKey('NMS', ~ServerID, spk_nms);
26    event HasKey('RA', ~ServerID, spk_ra);
27    event HasKey('EK', ~ServerID, ek_pk);
28    event ServerFinish()
29  )
```

Figure 3.5: The RA/NMS Server process for AK certification in SAPiC

create a session handle `ak_sh` for the AK (lines 33–37) and then calls the ideal functionality $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}$, with input the the AK session handle `ak_sh`, the EK handle `~ek_h`, the AK handle `ak_h` and the `credentialBlob` that was received from the Server. According to the description of $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}$ in Figure 3.3, the TPM will output the value `challenge` that was initially generated by the RA/NMS Server. The Router outputs this value in the message $<$ `'RA_enrollcert'`, `challenge` $>$, providing evidence of the authenticity of the Router to the RA/NMS Server, which will create the AK certificate. Finally, the Router receives the AK certificate `cert_ak` and verifies its correctness using the RA Server public key `spk_ra` (lines 44–48).

3.2.3 The RA/NMS Server Process

Our model for the aggregated RA/NMS Server in SAPiC is presented in Figure 3.5. We also declare a public `~ServerID` in the main process for the same convenience reason as we did in the Router process above. The RA/NMS Server receives the following request from the Router

$$\langle \texttt{'RA\_enrollrequest'}, \texttt{fqdn\_router}, \texttt{ek\_pk}, \texttt{ak\_spk} \rangle,$$

which initiates the enrollment process.

We note that in this statement, `ek_pk` is pattern-matched with the value that the Server expects, i.e., the registered EK. This models the fact that the RA/NMS Server will only proceed in the next

steps of the protocol, if ek_pk matches with the value in its trusted, local database. Next, the RA/NMS Server creates a fresh challenge and executes the function symbol

$$\texttt{makeCredential}(\texttt{ek\_pk}, \sim\texttt{challenge}, \texttt{ak\_spk}),$$

which emulates the execution of the TPM2_MakeCredential call of a local TPM. The output is the credentialBlob, which is cryptographically bound to the AK and EK, and it is sent to the Router in line 13 through the message:

$$\langle \text{`RA\_enrollrequest\_resp'}, \texttt{credentialBlob} \rangle.$$

The Router will receive the credentialBlob value and will proceed by executing the ideal functionality $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}$, in order to obtain and verify the challenge value.

In the abstract description of the TPM command TPM2_MakeCredential [11, Section 2.1], we see that this command involves a symmetric encryption operation (for encrypting the challenge) and an HMAC computation (for ensuring the challenge authenticity). In turn, the command TPM2_ActivateCredential involves the decryption and the HMAC verification. As we discussed in the Introduction, the ideal functionalities for both commands should exclude these cryptographic operations. We can do this by defining three function symbols makeCredential/3, activateCredential/3 and verifyCredential/3, satisfying the equational theories:

$$\begin{aligned} \texttt{activateCredential}(\texttt{n}, \texttt{k}, \texttt{makeCredential}(\texttt{pk}(\texttt{k}), \texttt{m}, \texttt{n})) &= \texttt{m} \\ \texttt{verifyCredential}(\texttt{n}, \texttt{k}, \texttt{makeCredential}(\texttt{pk}(\texttt{k}), \texttt{m}, \texttt{n})) &= \texttt{true} \end{aligned}$$

The RA/NMS Server receives the challenge from the Router and it pattern matches with the previously created value at line 15. If the expected value is received, the RA/NMS Server proceeds by creating the AK certificate cert_ak. This is done in line 17:

$$\texttt{cert\_ak} = \langle \langle \texttt{ak\_spk}, \texttt{fqdn\_router} \rangle, \texttt{sign}(\langle \texttt{ak\_spk}, \texttt{fqdn\_router} \rangle, \sim\texttt{ssk\_ra}) \rangle,$$

where the first part, namely $\langle \texttt{ak\_spk}, \texttt{fqdn\_router} \rangle$ is provided for clarity, and the second part is the signature on that message. This is output to the public channel, and forwarded (by the adversary) to the Router via the message $\langle \text{`RA\_enrollcert\_resp'}, \texttt{cert\_ak} \rangle$ (line 22).

3.3 Modelling of TLS Certification

We now proceed with the second part of the device management modelling, which is the creation of a TLS signing key pair by the Router using the TPM, and its certification by the RA/NMS Server. As in the AK certification process, we are interested in the interactions between the three processes: the Router, the TPM and the RA/NMS Server. The main process description is fairly similar to the case of the AK certification process, except for the fact that we also regard the AK as available to the honest parties, and we omit it from the discussion for brevity.

3.3.1 The TPM Process

Recall from Section 2.2 that the TPM commands that are needed for the creation of the TLS key are TPM2_StartAuthSession, TPM2_Create, and TPM2_Load, where the modelling of the latter command is not required in our ideal-functionality framework. The ideal functionalities for the first

```
let TPM =
  insert ⟨'authPolicy',  ~ak_h⟩, nil;
  insert ⟨'privatePart', ~ak_h⟩, ~ak_sk;
  insert ⟨'publicPart',  ~ak_h⟩, ak_pk;
  insert 'PCR', nil;

  !(
    //TPM2_PCR_Extend
    (
      let pat_tpm_command =
        ⟨'TPM2_PCR_Extend', value⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);

      lock 'device';
      lookup 'PCR' as pcr in
      insert 'PCR', ⟨value, pcr⟩;
      unlock 'device'
    ) ||
    //TPM2_StartAuthSession
    (
      let pat_tpm_command =
        ⟨'TPM2_StartAuthSession'⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);
      new ~s_h;
      lock 'device';
      insert ⟨'policyDigest', ~s_h⟩, nil;
      event CreateHandle(~s_h);
      out(~s_h);
      unlock 'device';
      0
    ) ||
    //TPM2_PolicyPCR
    (
      let pat_tpm_command =
            ⟨'TPM2_PolicyPCR', s_h⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);

      lock 'device';
      lookup 'PCR' as pcrL in
      lookup ⟨'policyDigest', s_h⟩ as pL in
      event PolicyPCR(s_h, pL);
      insert ⟨'policyDigest', s_h⟩,
        ⟨pcrL, 'TPM_CC_PolicyPCR', pL⟩;
      unlock 'device'
    ) ||
    //TPM2_Create
    (
      let pat_tpm_command =
        ⟨'TPM2_Create', authPolicy⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);
      new ~k_h;
      new ~k_sk;
      event SecretKey('TLS', ~k_sk);
      lock 'device';
      let k_pk = pk(~k_sk) in
      insert ⟨'authPolicy',  ~k_h⟩, authPolicy;
      insert ⟨'privatePart', ~k_h⟩, ~k_sk;
      insert ⟨'publicPart',  ~k_h⟩, k_pk;
      [ ]--[ ]-> [Tpm2CreateOut(⟨~k_h, k_pk⟩)];
      unlock 'device'
    ) ||
    //TPM2_Certify
    (
      let pat_tpm_command =
        ⟨'TPM2_Certify', obj_sh, obj_h, a_h⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);
      lock 'device';

      lookup ⟨'policyDigest', obj_sh⟩ as obj_pd in
      lookup ⟨'authPolicy', obj_h⟩ as obj_ap in

      if obj_pd = obj_ap then
            lookup ⟨'publicPart', obj_h⟩ as obj_pk in
            lookup ⟨'privatePart', a_h⟩ as a_sk in
            out(sign(⟨obj_pk, obj_ap⟩, a_sk));
            unlock 'device'
      else
            unlock 'device'
    ) ||
    //TPM2_Sign
    (
      let pat_tpm_command = ⟨'TPM2_Sign', k_sh, k_h, m⟩ in
      in(pat_tpm_command);
      event TPM_ReceiveCommand(...);
      lock 'device';

      lookup ⟨'policyDigest', k_sh⟩ as k_pd in
      lookup ⟨'authPolicy', k_h⟩ as k_ap in

      if k_pd = k_ap then
        lookup ⟨'privatePart', k_h⟩ as k_sk in
        [ ]--[ ]-> [Tpm2SignOut(sign(m, k_sk))];
        unlock 'device'
      else
        unlock 'device'
    )
  )
```

Figure 3.6: The TPM process for TLS certification in SAPiC

two commands have already been presented in the previous section. Since the TLS key is linked to a PCR policy, we also need to model the commands TPM2_PCRExtend and TPM2_PolicyPCR. The ideal functionalities $\mathcal{F}_{\texttt{TPM2\_PCRExtend}}$ and $\mathcal{F}_{\texttt{TPM2\_PolicyPCR}}$ were presented in detail in Deliverable D3.3 [10] and they are also described in the model for the TPM in this phase, in Figure 3.6.

The main operation of the command TPM2_PCRExtend is to extend a value provided by the caller to specific PCRs. As described in the TPM specification, the command extends value by updating the pcrDigest value of a PCR as:

$$\texttt{pcrDigest} \leftarrow \texttt{H}(\texttt{pcrDigest}||\texttt{value}).$$

In other words, this means that each value of the PCR corresponds to a chain of hash values. Since the intention of the ideal functionalities is to replace the cryptography used internally by the TPM with non-cryptographic approaches, in our ideal functionality $\mathcal{F}_{\texttt{TPM2\_PCRExtend}}$, we represent this chain of hash values by simply appending each value to a PCR list pcr. More concretely, we write:

$$\texttt{pcr} \leftarrow \langle \texttt{value}, \texttt{pcr} \rangle,$$

to denote that value has been appended (equivalently, extended) to the PCR list pcr. This can be seen in line 17 in Figure 3.6.

The same approach is also applied in the case of the ideal functionality $\mathcal{F}_{\texttt{TPM2\_PolicyPCR}}$. That is, the policy digest of a session is considered as a list pL and in order to update this policy digest with a PCR value, we simply append pcrL and the constant 'TPM_CC_PolicyPCR' to the policy digest list pL. The constant value 'TPM_CC_PolicyPCR' is added as an indication, showing which policy type was used for updating the policy digest of the session. This is described as

$$\texttt{pL} \leftarrow \langle \texttt{pcrL}, \text{`}\texttt{TPM\_CC\_PolicyPCR}\text{'}, \texttt{pL} \rangle,$$

and is implemented at lines 42–46 in Figure 3.6.

According to the description of the TLS certification process in Section 2.2, once the TLS key has been created, it needs to be certified using TPM2_Certify command, in order to create the SKAE signature that will be included in the CSR. The idealization of this command was presented in Deliverable D3.4 [11, Figure 3.1] and it is also described in Figure 3.6. Furthermore, we have also modelled the command TPM2_Sign, which is used by the Router in order to create the signature of the CSR. This is also described in Deliverable D3.4 [11, Figure 3.4] and in Figure 3.6. Note that both commands generate a signature, therefore we use the Tamarin builtin theory for signing sign/2 and for verifying the output verify/3, related by the well-known equation $\texttt{verify}(\texttt{sign}(\texttt{m}, \texttt{sk}), \texttt{m}, \texttt{pk}(\texttt{sk})) = \texttt{true}$.

3.3.2 The Router Process

The model for the Router in the TLS certification phase is described in Figure 3.7. We model the fact that the PCR is first extended with an publicly known software state. The next step is to create the TLS key by calling the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Create}}$, with input the authorization policy for the key tls_authPolicy, namely:

$$\texttt{tls\_authPolicy} = \langle \underbrace{\langle \texttt{swstate}, \texttt{nil} \rangle}_{\texttt{expected PCR value}}, \text{`}\texttt{TPM\_CC\_PolicyPCR}\text{'}, \texttt{nil} \rangle,$$

where $\langle \texttt{swstate}, \texttt{nil} \rangle$ corresponds to the current PCR list and swstate represents the current (trusted) state of the Router. The TPM returns the handle tls_h of the TLS key and its public

```
1  let Router =
2   (
3     new ~tid;
4
5     let pat_tpm_command0 =
6       ⟨'TPM2_PCR_Extend', swstate⟩ in
7     event TPM_SendCommand(...);
8     out(pat_tpm_command0);
9
10    let tls_authPolicy =
11      ⟨⟨swstate, nil⟩, 'TPM_CC_PolicyPCR', nil⟩ in
12    let pat_tpm_command1 =
13          ⟨'TPM2_Create', tls_authPolicy⟩ in
14    event TPM_SendCommand(...);
15    out(pat_tpm_command1);
16
17    [Tpm2CreateOut(⟨tls_h, tls_spk⟩)]--[]-⟩ [];
18
19    event HasKey('TLS', ~RouterID, tls_spk);
20
21    let pat_tpm_command2 =
22      ⟨'TPM2_StartAuthSession'⟩ in
23    event TPM_SendCommand(...);
24    out(pat_tpm_command2);
25    in(tls_sh);
26
27    event GenerateTLS(~tid, tls_spk);
28
29    let pat_tpm_command3 =
30      ⟨'TPM2_PolicyPCR', tls_sh⟩ in
31    event TPM_SendCommand(...);
32    out(pat_tpm_command3);
33    let pat_tpm_command4 =
34      ⟨'TPM2_Certify', tls_sh, tls_h, ~ak_h⟩ in
35    event TPM_SendCommand(...);
36    out(pat_tpm_command4);
37    in(skae);
38
39    event RouterUseTLS(tls_spk);
40
41    event HasKey('AK', ~RouterID, ak_pk);
42
43    if verify(skae, ... ) = true then
44      let pat_tpm_command5 =
45            ⟨'TPM2_Sign', tls_sh, tls_h,
46            ⟨tls_spk, skae⟩⟩ in
47      event TPM_SendCommand(...);
48      out(pat_tpm_command5);
49      [Tpm2SignOut(csr)]--[]-⟩ [];
50
51      event GenerateValue('CSR', ~RouterID, csr);
52      event RouterRunning(~RouterID, ~ServerID, ...);
53      out(⟨⟨tls_spk, skae⟩, csr⟩);
54
55      event HasKey('NMS', ~RouterID, spk_nms);
56
57      in(⟨tls_spk, signature_cert_tls⟩);
58      if verify(signature_cert_tls, ... ) = true then
59        event RouterCommit(~RouterID, ~ServerID, ...);
60        event RouterFinish()
61  )
```

Figure 3.7: The Router process for TLS certification in SAPiC

part `tls_spk` to the Router. Then, the Router calls the ideal functionality $\mathcal{F}_{\texttt{TPM2\_StartAuthSession}}$ in order to create a new policy session with handle `tls_sh`, and then calls the ideal functionality $\mathcal{F}_{\texttt{TPM2\_PolicyPCR}}$ in order to update the policy digest in that policy session. Note that according to the description of $\mathcal{F}_{\texttt{TPM2\_PolicyPCR}}$, this policy digest should be updated to

$$\langle \texttt{pcrL}, \texttt{`TPM\_CC\_PolicyPCR'}, \texttt{nil} \rangle,$$

hence it should coincide with the `authPolicy` of the created TLS key.

Next, the Router calls the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Certify}}$, in order to create the SKAE for the TLS key at line 34. According to Figure 3.6, this SKAE is the signature on the public part of the TLS key and its authorization policy with the private part of the AK, that is,

$$\texttt{skae} = \texttt{sign}(\langle \texttt{tls\_spk}, \texttt{tls\_authPolicy} \rangle, \texttt{ak\_ssk}).$$

For convenience, we assume that the Router process verifies this value at line 43.

The Router creates the CSR, calling the command `TPM2_Sign`, which in our model is the signature on the message $\langle \texttt{tls\_spk}, \texttt{skae} \rangle$, using the private part of the TLS key:

$$\texttt{signature\_csr} = \texttt{sign}(\langle \texttt{tls\_spk}, \texttt{skae} \rangle, \texttt{tls\_ssk}),$$

```
1  let RANMSServer =
2   !(
3     let tls_authPolicy = ⟨⟨swstate, nil⟩, 'TPM_CC_PolicyPCR', nil⟩ in
4
5     in(⟨⟨tls_spk, skae⟩, signature_csr⟩);
6
7     if verify(skae, ⟨tls_spk, tls_authPolicy⟩, ak_pk) = true then
8     if verify(signature_csr, ⟨tls_spk, skae⟩, tls_spk) = true then
9       event ReceiveValue('CSR', ~ServerID, csr);
10      event HasKey('TLS', ~ServerID, tls_spk);
11      event HasKey('AK', ~ServerID, ak_pk);
12      event HasKey('NMS', ~ServerID, spk_nms);
13
14      event ServerCommit(~ServerID, ~RouterID, ⟨tls_spk, swstate, ak_pk⟩);
15
16      let pat_cert_tls = ⟨tls_spk, sign(tls_spk, ~ssk_nms)⟩ in
17      event ServerRunning(~ServerID, ~RouterID, pat_cert_tls);
18      out(pat_cert_tls);
19      event ServerFinish()
20  )
```

Figure 3.8: The RA/NMS Server process for TLS certification in SAPiC

which is sent to the RA/NMS Server for verification. The last action of the Router in the TLS certification process is to receive the TLS certificate `<tls_spk, signature_cert_tls>` by the RA/NMS Server, and verify its correctness:

$$\texttt{verify}(\texttt{signature\_cert\_tls}, \texttt{tls\_spk}, \texttt{spk\_nms}) = \texttt{true}.$$

3.3.3 The RA/NMS Server Process

The modelling for the RA/NMS Server process in the case of the TLS certification phase is presented in Figure 3.8. The main actions of the RA/NMS Server in the TLS certification process are to verify the SKAE and CSR received from the Router, and the creation of the TLS certificate. Since we are modelling the NMS and the RA Servers as a single process, we ignore the communication between these two entities and any interaction with local databases. Consequently, we consider the creation of the TLS certificate in the same process.

The RA/NMS Server receives the message

$$\langle\langle \texttt{tls\_spk}, \texttt{skae}\rangle, \texttt{signature\_csr}\rangle$$

from the Router and verifies the `signature_csr` and `skae` values, using the public part of the TLS and AK keys, respectively (lines 7–8):

$$\begin{aligned}&\texttt{if}\ \ \texttt{verify}(\texttt{skae}, \langle \texttt{tls\_spk}, \texttt{tls\_authPolicy}\rangle, \texttt{ak\_spk}) = \texttt{true}\ \ \texttt{then}\\ &\texttt{if}\ \ \texttt{verify}(\texttt{signature\_csr}, \langle \texttt{tls\_spk}, \texttt{skae}\rangle, \texttt{tls\_spk}) = \texttt{true}\ \ \texttt{then}\end{aligned}$$

Finally, the RA/NMS Server creates the TLS certificate by signing the public part of the TLS key `tls_spk` with its secret signing key $\sim$`ssk_nms`, and outputs the TLS certificate:

$$\texttt{out}(\langle \texttt{tls\_spk}, \texttt{sign}(\texttt{tls\_spk}, \sim\texttt{ssk\_nms})\rangle).$$

3.4 Modelling of TLS Communication and Attestation

The last phase in the device management use case that we model is the establishment of the secure communication channel between the Router and the NMS in order to provide attestation reports. Recall from our description in Section 2.3 that, in this phase, both the TLS key and the AK are used for different purposes. The TLS key is used between the Router and the NMS, in order to establish a session key. This session key is a symmetric encryption key that allows the Router and the NMS to exchange messages in an encrypted manner. Modelling this step in the TLS communication is out of the scope in our objectives, where we mainly focus on the usage of the AK. In particular, we model the ability of the Router to sign quotes using the command `TPM2_Quote` and transmitting these quotes to the Server process. The NMS verifies these quotes, using the public part of the AK, proving that the Router software is in correct, expected state. For the encrypted TLS communication between the Router and the NMS, we assume that a secret session key is established for symmetric encryption and it is available to the Router and the NMS. We emulate this TLS session key establishment through the usage of a shared session key via persistent state in SAPiC.

For our modelling purposes, we model two types of Routers, a “corrupted” and an “honest” one, so that in each execution of the protocol, Tamarin will non-deterministically choose one of the two versions. The “corrupted” Router is assumed to have access to the TPM and the session key, and hence it can sign quotes and transmit them to the NMS, but we assume that it is not in a correct integrity state. The purpose in this case is to prove that the NMS will not accept quotes from a Router whose integrity values do not match the expected ones.

3.4.1 The TPM Process

As we already described in Section 2.3 above, there one additional TPM command that needs to be modelled, the `TPM2_Quote`. The ideal functionality $\mathcal{F}_{\texttt{TPM2\_Quote}}$ is presented in Figure 3.9. As described in Figure 2.3, the main action of the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Quote}}$ is to create a signature on certain externally provided data and selected PCRs, using a signing key. This key in our model is the AK that is previously generated by the Router and is certified by the Server. However, since we consider this phase independent from the AK certification, we assume that the AK is a global key and its public part is available to the corresponding parties.

```
let TPM =
  insert ⟨'authPolicy',  ~ak_h⟩, nil;
  insert ⟨'privatePart', ~ak_h⟩, ~ak_sk;
  insert ⟨'publicPart',  ~ak_h⟩, ak_pk;

  insert ⟨'policyDigest',  ~ak_sh⟩, nil;

  insert ⟨'authPolicy',  ~tls_h⟩, ⟨⟨swstate, nil⟩, 'TPM_CC_PolicyPCR', nil⟩;
  insert ⟨'privatePart', ~tls_h⟩, ~tls_sk;
  insert ⟨'publicPart',  ~tls_h⟩, tls_pk;

  //TPM2_Quote
  !(
    let pat_tpm_command = ⟨'TPM2_Quote', k_h, k_sh, data⟩ in
    [Tpm2QuoteIn(tid, pat_tpm_command)]--[]-⟩ [];
    event TPM_ReceiveCommand(pat_tpm_command);

    lock 'device';
    lookup 'PCR' as pcr in
    lookup ⟨'authPolicy', k_h⟩ as aP in
    lookup ⟨'policyDigest', k_sh⟩ as pD in

    if pD = aP then
      lookup ⟨'privatePart', k_h⟩ as k_privatePart in
      [ ]--[ ]-> [Tpm2QuoteOut(tid, sign(⟨data, pcr⟩, k_privatePart))];
      unlock 'device'
    else
      unlock 'device'
  )
```

Figure 3.9: The TPM process for TLS connection in SAPiC

```
let Router =
(
 (
  insert 'PCR', ⟨'x', swstate, nil⟩;
  insert ⟨'policyDigest', ~tls_sh⟩,
   ⟨⟨'x', swstate, nil⟩, 'TPM_CC_PolicyPCR', nil⟩;

  event Corrupted();

  !(
   new ~tid;

   event HasKey('AK', ~RouterID, ak_pk);

   lookup 'session_key' as sess_key in
   in(encrypted_qData);
   let qData =
    sdec(encrypted_qData, sess_key) in
   event Receive(qData);

   let pat_tpm_comm1 =
    ⟨'TPM2_Quote', ~ak_h, ~ak_sh, qData⟩ in
   event TPM_SendCommand(...);
   [ ]--[ ]-> [Tpm2QuoteIn(~tid, pat_tpm_comm1)];
   [Tpm2QuoteOut(~tid, quote)]--[]-⟩ [];

   event GenerateValue('QUOTE', ~RouterID, quote);

   let encrypted_quote = senc(quote, sess_key) in
   event RouterRunning(~RouterID, ~ServerID, ...);
   out(encrypted_quote);
   event RouterFinish1()
  )
 ) +
 (
  insert 'PCR', ⟨swstate, nil⟩;
  insert ⟨'policyDigest', ~tls_sh⟩,
   ⟨⟨swstate, nil⟩, 'TPM_CC_PolicyPCR', nil⟩;

  event Trusted();

  !(
   new ~tid;

   event HasKey('AK', ~RouterID, ak_pk);

   lookup 'session_key' as sess_key in
   in(encrypted_qData);

   let qData =
    sdec(encrypted_qData, sess_key) in
   event Receive(qData);

   let pat_tpm_comm1 =
    ⟨'TPM2_Quote', ~ak_h, ~ak_sh, qData⟩ in
   event TPM_SendCommand(...);
   [ ]--[ ]-> [Tpm2QuoteIn(~tid, pat_tpm_comm1)];
   [Tpm2QuoteOut(~tid, quote)]--[]-⟩ [];

   event GenerateValue('QUOTE', ~RouterID, quote);

   let encrypted_quote = senc(quote, sess_key) in
   event RouterRunning(~RouterID, ~ServerID, ...);
   out(encrypted_quote);
   event RouterFinish2()
  )
 )
)
```

Figure 3.10: The Router process for TLS connection in SAPiC

3.4.2 The Router Process

As discussed above, we consider two integrity states of the Routers, depending on whether the Router is in correct state ("trusted") or not ("corrupted"). SAPiC has the non-deterministic choice consctuct "+" that allows the tool to branch in one of these two cases. Our model for the Router in this phase of the use case is presented in Figure 3.10.

We assume that the "trusted" Router has been extended the PCR with the expected software hash measurements, and the PCR should be in the state `<swstate, nil>`. The "corrupted" Router, on the other hand, has been extended with an additional measurement that reflects unexpected software, and thus, the PCR is in the incorrect state `<'x', swstate, nil>`. This is the only difference in the model of the two versions of the Router. All other actions follow in the same way in both versions of the process.

In particular, the Router first establishes a shared TLS session key, which is emulated in lines 12 and 42. Then, it receives the `encrypted_qData` from the Server and decrypts it in order to obtain the qualifying data `qData`, using the common session key `session_key`. It calls the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Quote}}$, with input the handle `~ak_h` of the AK, the handle `~ak_sh` of the

```
let RANMSServer =
 !(
   new ~tid;
   new ~session_key;
   new ~qData;

   event GenerateKey(~tid, ~session_key);
   event GenerateQData(~tid, ~qData);
   event Source(~qData);
   event SecretKey('SessionKey', ~session_key);

   insert 'session_key', ~session_key;

   event HasKey('AK', ~ServerID, ak_pk);

   let encrypted_qData = senc(~qData, ~session_key) in
   event NMSUseKey(~session_key);
   out(encrypted_qData);

   in(encrypted_quote);
   let quote = sdec(encrypted_quote, ~session_key) in

   if verify(sdec(encrypted_quote, ~session_key), ⟨~qData, ⟨swstate, nil⟩⟩, ak_pk) = true then
     event ReceiveValue('QUOTE', ~ServerID, quote);
     event ServerCommit(~ServerID, ~RouterID, ⟨~qData, ~session_key⟩);
     event ServerFinish()
 )
```

Figure 3.11: The RA/NMS Server process for TLS connection in SAPiC

corresponding policy session that is associated to the AK, and the received qualifying data `qData`. We note here that in the input to the TPM, we have also considered an identifier `~tid`, in order to distinguish the different executions of the protocol, as this will be required to prove adequately the security properties. The TPM outputs the `quote` value which is the signature on the message `<qData, pcr>` using the private part of the AK. This quote is then encrypted by the Router with the `session_key` and the resulting ciphertext `encrypted_quote` is sent to the Server.

3.4.3 The RA/NMS Server Process

The RA/NMS Server initiates the establishment of the secure communication channel. It first emulates the creation of a TLS session key, and then it creates a new value `~qData` that corresponds to the qualifying data and encrypts it with the session key. The ciphertext `encrypted_qData` is then sent to the Router.

After receiving the `encrypted_quote` from the Router, the Server decrypts it using the common session key `session_key` and obtains the `quote` value. Recall that this is a signature on the message `<qData, pcr>` signed with the AK, and hence the Server can verify its validity using the public part `ak_pk` of the AK.

Chapter 4

Formalization and Verification of Security Properties

In this chapter, we start by providing an intuitive description of the security properties that the remote attestation scheme is designed to provide, in the context of **integrity, confidentiality, and secure measurement**. Then we proceed with the description of these security properties in the form of lemmas and their formal verification using the Tamarin prover. In particular, we will consider a set of six main security properties:

Sanity-check lemmas: Such lemmas are necessary in order to ensure the correctness of the model. More precisely, sanity-check lemmas are often used to show that the model executes (reaches) all possible branches, and therefore, that the components reach the end of the protocol. The remaining properties depend on satisfying this reachability property, otherwise they might be trivially satisfied.

Availability of keys at honest processes: Such lemmas show that all honest parties have initial access to the trusted key material required, so that they can build the chain of trust.

Key freshness and secrecy: These lemmas ensure that the keys created during the protocol execution are fresh and not available to the adversary. In Tamarin, we use the special action fact $K(m)$ to denote that the term m is known by the adversary.

Authentication: We consider the agreement property from Lowe's hierarchy [17] in the parameters exchanged in the protocol. Whenever it is possible (e.g., for a session key), we also require injective and mutual agreement. As depicted in Figure 4.1, this property is usually encoded through the usage of the events "Running" and "Commit". The placement of these events has some flexibility, but not all placements are correct. The "Running" event must be placed *before* the party A that is being authenticated sends its last message, and the "Commit" event must be placed *after* the party B, to whom the authentication is being made, receives the last message from A.

Transfer of information as generated: Such lemmas ensure that cryptographic material, such as certificates or the TPM quotes, are received at the destination process as generated by the process of origin.

No reuse of key: We make sure that a specific key is used only once (mainly used for session keys).

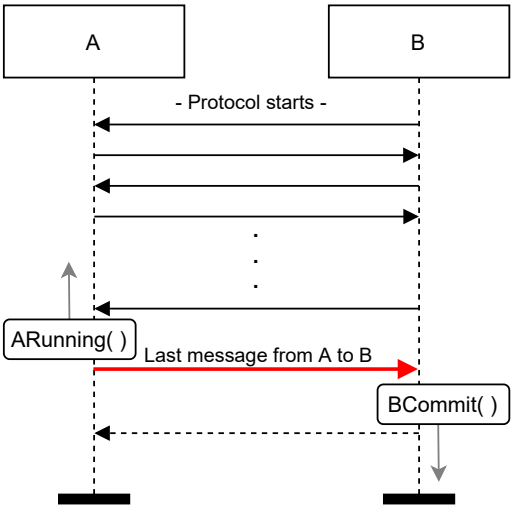

Figure 4.1: Placement of Running and Commit events for the authentication (agreement) property

In what follows, we discuss these properties for the AK certification part in Section 4.1, for the TLS certification in Section 4.2, and for the establishment of the secure communication channel and attestation reporting in Section 4.3.

4.1 Security Properties for AK Certification

Sources lemma. Before delving into the security properties for this part, we discuss a required sources lemma in order to remove partial deconstructions in this model. Recall from Chapter 3 that in the PCA-based IBM protocol the server generates a random value `challenge` and sends it to the Router in protected form, namely inside the `credentialBlob`, which is the output of the ideal functionality $\mathcal{F}_{\texttt{TPM2\_MakeCredential}}$. The Router receives the `credentialBlob`, and retrieves the `challenge` by using the ideal functionality $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}$. As we discussed in Deliverable D3.4 [11, Section 4.2], this caused partial deconstructions (open chains) left, which in turn made the proof far more complicated, and even impossible to terminate. We tackled this problem using a sources lemma.

More concretely, we introduce two events `Source(challenge)` and `Receive(challenge)`. The first is placed in the RA/NMS Server process, since this is the entity that initially generates the `challenge`, while the second is placed in the TPM process, which is the entity that retrieves the `challenge` from the `credentialBlob`. Then we define the following lemma:

```
lemma SourcesLemma [sources]:
  "All m #t1. Receive(m)@t1 ==>
    ((Ex #t2. KU(m)@t2 & (t2 < t1)) | (Ex #t3. Source(m)@t3))"
```

This lemma states that whenever an honest process (Router) is able to retrieve a secret `m`, then this `m` was either known beforehand by the adversary, or it was freshly created by another honest process (RA/NMS Server). Since honest processes in our model do not leak keys, the lemma helps Tamarin to realize that the output of the protocol can not be used by the adversary to gain any additional information.

Sanity check. We introduce two events `RouterFinish()` and `ServerFinish()`. The first is placed at the end of the Router process (Figure 3.4), right after the Router verifies the certificate

cert_ak received from the RA/NMS Server. The second is placed at the end of the RA/NMS Server process, as shown in Figure 3.5, right after the Server outputs the AK certificate. We define the sanity check lemmas for these events as

```
lemma RouterFinishes:
exists-trace
  "Ex #t1. RouterFinish()@t1"

lemma ServerFinishes:
exists-trace
  "Ex #t1. ServerFinish()@t1"
```

which ensure that the model has reached the events RouterFinish() and ServerFinish(), at least once (exists-trace). Sanity lemmas are important as they ensure that the model executes correctly. They must be proved before any other security property.

Availability of keys at honest processes. We need to prove that all honest processes, in this case the Router and RA/NMS Server, have access to the keys required in order to successfully and securely complete the AK certification process. For this purpose we define the event:

```
HasKey(label, process_id, k)
```

where label refers to the key identifier (e.g., AK key), process_id refers a unique identifier of the process, and k to the actual key value. In the AK certification process we have four key pairs that are created and for which we wish to verify that their public parts are available to the Router and the Server. These are the EK ek_pk with label 'EK', the RA signing key spk_ra with label 'RAK', the NMS signing key spk_nms with label 'NMS', and the AK ak_spk with label 'AK'.

Each of those events appears twice in the model, one time in each process. Then we define the lemma:

```
lemma AvailabilityKey:
  "All label id1 id2 k1 k2 #t1 #t2. not(id1 = id2) &
  HasKey(label, id1, k1)@t1 & HasKey(label, id2, k2)@t2 ==> (k1 = k2)"
```

Note that the above is an "all-traces" lemma, meaning that we require the proof to hold for all possible executions of the protocol, i.e., for all AK certification instances. The above lemma states that whenever two events HasKey with the same label are launched at different processes, then it must be the case that the associated key is the same in both processes.

Key freshness. We prove this property only for the public part of the AK, which is the only key in our model that is created by the TPM. We define the event

```
GenerateAK(thread_id, k),
```

at the Router side, where thread_id is the thread identifier. Then we model the following lemma:

```
lemma FreshnessAK:
  "All tid1 tid2 k #t1 #t2. GenerateAK(tid1, k)@t1 &
  GenerateAK(tid2, k)@t2 ==> (tid1 = tid2)"
```

This lemma states that whenever an AK with public part k is created in two different thread executions with identifiers tid1 and tid2 respectively, then these two thread executions are actually the same, i.e., tid1 = tid2. Note that we want this property to hold for all thread executions, hence this is also an "all-traces" lemma.

Secrecy. We prove this property for the private parts of all keys that are used in the AK certification process. These are the RA Server's key `ssk_ra` with label `'RA'`, the NMS key `ssk_nms` with label `'NMS'`, the EK `ek_sk` with label `'EK'` and the AK `ak_sk` with label `'AK'`. We define the event:

```
SecretKey(label, k_priv)
```

and we model the following lemma:

```
lemma SecretKey:
  "All label k #t1. SecretKey(label, k)@t1 ==> (not (Ex #t2. K(k)@t2))"
```

where the special action fact `K(k)` denotes knowledge of the key `k` by the adversary. The above lemma states that once the key `k` is referred by the event then the adversary has no knowledge of it at any time point.

Correct transfer. We prove this property for the AK certificate `cert_ak`. In particular, we consider the following two events:

```
GenerateCertAK(~ServerID, cert_ak)

ReceiveCertAK(~RouterID, cert_ak)
```

The first will be placed on the RA/NMS Server side, after the creation of the AK certificate, while the second on the Router side, after the verification of the AK certificate. We require that the following lemma hold:

```
lemma CorrectTransfer:
  "All id1 m #t1. ReceiveCertAK(id1, m)@t1 ==>
  (Ex id2 #t2. GenerateCertAK(id2, m)@t2 & (t2 < t1))"
```

The lemma states that whenever a Router with identifier `RouterID` receives a value `m` at a timepoint `t1`, then there exists a timepoint `t2`, in which a Server with identifier `ServerID` has sent this value `m`.

Authentication. For authentication, as we discussed in Deliverable D3.4 [11, Section 4.5], we consider the strongest possible version of the authentication property from Lowe's hierarchy [17]. In this particular case, and since the Router executes the AK certification once at the beginning, we model mutual, injective agreement. Some templates for authentication lemmas are provided in the Tamarin manual [3], and we adopt them according to our requirements. We declare the following events:

```
ServerRunning(~ServerID, ~RouterID, cert_ak)
RouterCommit(~RouterID, ~ServerID, cert_ak)

RouterRunning(~RouterID, ~ServerID, challenge)
ServerCommit(~ServerID, ~RouterID, ~challenge)
```

That is, for each of the two processes, we introduce a "Commit" and a "Running" event. These events must be appropriately placed: the "Running" event must be placed before the referenced party sends its last message, and the corresponding "Commit" must be placed at the other party after receiving that message. Thus, we define the following lemma:

Lemma:	SourcesLemma	RouterFinishes	ServerFinishes	AvailabilityKey
Type:	sources	exists-trace	exists-trace	all-traces
Verified:	yes	yes	yes	yes
Steps:	70	37	29	1378

Lemma:	FreshnessAK	CorrectTransfer	Authentication	SecretKey
Type:	all-traces	all-traces	all-traces	all-traces
Verified:	yes	yes	yes	yes
Steps:	4	383	403	12

Table 4.1: Results for AK certification

```
lemma Authentication:
  "(All X Y param #t1. RouterCommit(X, Y, param)@t1 ==>
    ( ( Ex #t2. ServerRunning(Y, X, param)@t2 & (t2 < t1) )
    & not( Ex X2 Y2 #t3. RouterCommit(X2, Y2, param)@t3 & not(#t3 = #t1) )
    )
  ) &
  (All X Y param #t1. ServerCommit(X, Y, param)@t1 ==>
    ( ( Ex #t2. RouterRunning(Y, X, param)@t2 & (t2 < t1) )
    & not(Ex X2 Y2 #t3. ServerCommit(X2, Y2, param)@t3 & not(#t3 = #t1))
    )
  )"
```

The above lemma states that for each `RouterCommit` event executed by the Router `X`, then the Server `Y`, executed the corresponding `ServerRunning` event earlier and for each run of the protocol there is a unique `RouterCommit` event executed by the Router. The converse occurs for the events `ServerCommit` and `RouterRunning`.

Automated analysis results. The model for the AK certification, as well as all the above lemmas are contained in the file `AK_model.sapic`. We can run the verification of the lemmas through the command:

```
tamarin-prover --prove AK_model.sapic
```

and it takes Tamarin around 1h (VM 3 cores, 4GB RAM on Intel(R) Core(TM) i5-4570 @ 3.20GHz) to successfully verify all the above lemmas. Table 4.1 summarizes the results of this section.

4.2 Security Properties for TLS Certification

Sanity check. Similarly as in the case of the AK certification process, we define the events `RouterFinish()` and `ServerFinish()` (see Figures 3.8 and 3.7), which signal the end of the two processes. The last action of the RA/NMS Server is to output the TLS certificate it has signed, with the private key of the NMS. On the other hand, the last action on the Router side is to verify the TLS certificate it received. The sanity check lemmas are identical as the ones we have for the AK certification, and we omit them here. The rest of the lemmas are verified using replication for the TPM and RA/NMS Server processes.

Availability of keys at honest processes. We prove that all honest processes, the Router and RA/NMS Server, have access to the keys that are used in the TLS certification process. There are three keys that are required here, the AK, the TLS key and the NMS singing key. Thus, we define the event

```
HasKey(label, process_id, k),
```

as we did above in the case of the AK certification, and instantiate it with the labels ‘NMS’, ‘AK’ or ‘TLS’ accordingly. The template of the AvailabilityKey, therefore mimics the case discussed above for the AK certification:

```
lemma AvailabilityKey:
  "All label id1 id2 k1 k2 #t1 #t2. not(id1 = id2) &
  HasKey(label, id1, k1)@t1 & HasKey(label, id2, k2)@t2 ==> (k1 = k2)"
```

Key freshness. We check this property for the TLS key, since this is created by the Router using the TPM. More concretely, we define the event

```
GenerateTLS(thread_id, k)
```

where thread_id refers to the Router thread identifier. We model a similar lemma as we did for the AK:

```
lemma FreshnessTLS:
  "All tid1 tid2 k #t1 #t2. GenerateTLS(tid1, k)@t1 &
  GenerateTLS(tid2, k)@t2 ==> (tid1 = tid2)"
```

In other words, there do not exist two different threads, for which the same TLS key is created.

Secrecy. Again, we verify that the secrecy property is satisfied for the private parts of all keys that are used in the TLS certification phase. These are the TLS key tls_ssk with label ‘TLS’, the NMS private signing key ssk_nms with label ‘NMS’, and the private part of the AK, ak_sk with label ‘AK’. Following the modelling of the secrecy property in the AK phase, we define the event SecretKey(label, k_priv) and the lemma SecretKey in a similar way.

Correct transfer. We model this property for the CSR that is generated by the Router and is transferred to the RA/NMS Server. We create two events:

```
GenerateValue('CSR', ~RouterID, csr)

ReceiveValue('CSR', ~ServerID, csr)
```

and define the lemma:

```
lemma CorrectTransfer:
  "All label id1 m #t1. ReceiveValue(label, id1, m)@t1 ==>
  (Ex id2 #t2. GenerateValue(label, id2, m)@t2 & (t2 < t1))"
```

The lemma proves that whenever a Server with identifier ServerID receives a CSR at timepoint t1, then there exists a timepoint t2, in which a Router with identifier RouterID has sent this unmodified CSR.

Lemma:	RouterFinishes	ServerFinishes	AvailabilityKey	FreshnessTLS
Type:	exists-trace	exists-trace	all-traces	all-traces
Verified:	yes	yes	yes	yes
Steps:	103	92	537	4

Lemma:	CorrectTransfer	Authentication	SecretKey
Type:	all-traces	all-traces	all-traces
Verified:	yes	yes	yes
Steps:	406	2987	10

Table 4.2: Results for TLS certification

Authentication. In the authentication direction from Router to Server, we prove a non-injective agreement for the parameters `<tls_spk, swstate, ak_pk>`. These parameters correspond to the public parts of the TLS and AK keys and the expected software state `swstate` of the Router, which is extended to the PCR. In the direction from Server to Router, we use the TLS certificate `<tls_spk, signature_cert_tls>` as the agreement parameter. Therefore, we require the events

```
ServerRunning(~ServerID, ~RouterID, <tls_spk, signature_cert_tls>)
RouterCommit(~RouterID, ~ServerID, <tls_spk, signature_cert_tls>)

RouterRunning(~RouterID, ~ServerID, <tls_spk, swstate, ak_pk>)
ServerCommit(~ServerID, ~RouterID, <tls_spk, swstate, ak_pk>)
```

placed appropriately at the Router and at the RA/NMS Server processs. Then we model the lemma

```
lemma Authentication:
  "(All X Y param #t1. RouterCommit(X, Y, param)@t1 ==>
    ( Ex #t2. ServerRunning(Y, X, param)@t2 & (t2 < t1) )
  ) &
  (All X Y param #t1. ServerCommit(X, Y, param)@t1 ==>
    ( Ex #t2. RouterRunning(Y, X, param)@t2 & (t2 < t1) )
  )"
```

This lemma states that for each `ServerCommit` event executed by the Server `X`, then Router `Y`, executed the corresponding `RouterRunning` event earlier. In other words, the Server will commit to a set of parameters, in particular to the values `<tls_spk, swstate, ak_pk>`, if there is a Router that was previously using these parameters. Similarly, it also states the agreement in the other direction for the parameters `<tls_spk, signature_cert_tls>`.

Automated analysis results. The model for the TLS certification process and the corresponding lemmas discussed in this section are presented in the file `TLS_model.sapic`. We can run the command

```
tamarin-prover --prove TLS_model.sapic
```

to verify the lemmas presented, and it will take Tamarin around 1h (VM 3 cores, 4GB RAM on Intel(R) Core(TM) i5-4570 @ 3.20GHz), producing the results summarized in Table 4.2. We remark that no sources lemma were required in this case.

4.3 Security Properties for TLS Communication & Attestation

Sanity check. We define the event `ServerFinish()` for the server, however, the Router requires two events, namely `RouterFinish1()` and `RouterFinish2()`, since it may branch in two different states (a trusted and a corrupted state). We place these events at the end of each branch, and define the lemmas

```
lemma RouterFinishes1:
exists-trace
  "Ex #t1. RouterFinish1()@t1"

lemma RouterFinishes2:
exists-trace
  "Ex #t1. RouterFinish2()@t1"

lemma ServerFinishes:
exists-trace
  "Ex #t1. ServerFinish()@t1"
```

which ensure reachability of all the possible branches.

Availability of keys at honest processes. In this case, we check the availability of the AK at the Router and NMS on the RA Server side using the event

```
HasKey(label, process_id, k).
```

We omit the body of the `AvailabilityKey` lemma for brevity, as it follows the same approach as for the AK and TLS phases.

Key freshness. We check this property for the `session_key` and the qualifying data `qData` as well. We define the events

```
GenerateKey(thread_id, k)
GenerateQData(thread_id, data)
```

where `thread_id` refers to the Server thread identifier, and then we declare the lemmas:

```
lemma FreshnessQData:
  "All tid1 tid2 k #t1 #t2. GenerateQData(tid1, k)@t1 &
  GenerateQData(tid2, k)@t2 ==> (tid1 = tid2)"

lemma FreshnessKey:
  "All tid1 tid2 k #t1 #t2. GenerateKey(tid1, k)@t1 &
  GenerateKey(tid2, k)@t2 ==> (tid1 = tid2)"
```

These lemmas verify that there are no two different threads, for which the same session key and qualifying data are created.

Secrecy. We verify that secrecy holds for the secret part of the AK `ak_sk`, with label `'AK'`, and for the session key `session_key`, with label `'SessionKey'`. Therefore we define the event `SecretKey(key_label, k)` and the lemma `SecretKey` exactly as in the case of AK and TLS certification.

Correct transfer. We model this property for the quote that is generated by the Router and verified by the RA/NMS Server. This requires the usual events:

```
GenerateValue('QUOTE', ~RouterID, quote)

ReceiveValue('QUOTE', ~ServerID, quote)
```

on the Router side and RA/NMS Server side respectively. The lemma:

```
lemma CorrectTransfer:
  "All label id1 m #t1. ReceiveValue(label, id1, m)@t1 ==>
  (Ex id2 #t2. GenerateValue(label, id2, m)@t2 & (t2 < t1))"
```

verifies that whenever a `quote` is received by the NMS, then this `quote` has been created by the Router and it has been unmodified.

Authentication. We verify the injective agreement property for the qualifying data and the session key, `<qData, session_key>`. We consider the usual template with "Commit" and "Running" events, namely:

```
RouterRunning(~RouterID, ~ServerID, <qData, session_key>)

ServerCommit(~ServerID, ~RouterID, <~qData, ~session_key>)
```

and the lemma:

```
lemma Authentication:
  "All X Y param #t1. ServerCommit(X, Y, param)@t1 ==>
    ( ( Ex #t2. RouterRunning(Y, X, param)@t2 & (t2 < t1) )
    & not(Ex X2 Y2 #t3. ServerCommit(X2, Y2, param)@t3 & not(#t3 = #t1))
    )"
```

This lemma states that for each `ServerCommit` event executed by the Server `X`, then the Router `Y` executed the corresponding `RouterRunning` event earlier. In other words, the Server will commit (injectively) to the set of parameters `<~qData, ~session_key>`, if there is a Router that was previously using these parameters and for each run of the protocol there is a unique "Commit" event.

No reuse of session key. We ensure that the session key is not reused for this phase through the event `NMSUseKey(session_key)` on the NMS side, and the lemma:

```
lemma NoReuse:
  "All k #t1 #t2. NMSUseKey(k)@t1 & NMSUseKey(k)@t2 ==> (#t1 = #t2)"
```

This lemma states that whenever two events `NMSUseKey` are executed by the same NMS and for the same session key, then these two events are actually the same event. In other word, there is no execution trace where the session key is reused to encrypt the qualifying data.

Corrupted router. In this part of the device management model, we consider an additional security property. Recall from Section 3.4, that for the Router process we have modelled two different states. A "corrupted" state in which the PCR value is extended to an invalid software state and the "trusted" state, in which the PCR has the correct software state. We intend to prove that if the Router is in the "corrupted" state, the NMS will not accept the quote from this Router. For this purpose, we define the event `Corrupted()` on the branch of the Router process that models the "corrupted" Router and define the lemma:

Lemma:	RouterFinishes1,2	ServerFinishes	AvailabilityKey	FreshnessKey	FreshnessQData
Type:	exists-trace	exists-trace	all-traces	all-traces	all-traces
Verified:	yes, yes	yes	yes	yes	yes
Steps:	18, 18	18	24	4	4

Lemma:	SecretKey	CorrectTransfer	Authentication	NoReuse	Corrupted
Type:	all-traces	all-traces	all-traces	all-traces	all-traces
Verified:	yes	yes	yes	yes	yes
Steps:	6	9	2237	8	1699

Table 4.3: Results for TLS communication and attestation

```
"All #t1. Corrupted()@t1 ==>
  not(Ex X Y param #t2. ServerCommit(X, Y, param)@t2 & (#t1 < #t2))"
```

where the event `ServerCommit` is the same as defined in the `Authentication` lemma. This lemma states that for all executions of the protocol where the Router is in a “corrupted” state, there does not exist a case in which the NMS commits to a set of parameters, which is exactly what we want to prove.

Automated analysis results. The model for the establishment of a secure TLS communication and attestation and the corresponding lemmas discussed in this section are presented in the file `Quote_model.sapic`. We can run the command

```
tamarin-prover --prove Quote_model.sapic
```

to verify the lemmas presented, and it will take Tamarin around 0.5h (VM 3 cores, 4GB RAM on Intel(R) Core(TM) i5-4570 @ 3.20GHz), producing the results summarized in Table 4.3. We remark that, again, no sources lemma were required in this case.

Chapter 5

Extending our Security Models Towards Enhanced System Reliability

After having formalized and verified the notion of secure remote attestation, we now proceed in showcasing how our models and verification methodology can be extended to also achieve ***security, privacy, integrity and system reliability*** in the more generic context of "*Systems-of-Systems*"; with the other two envisioned use cases in the domains of Fintech and Assistive Healthcare providing the natural extensions to be considered.

In particular, we argue that the idealized functionalities, considered as part of the *trusted platform command abstraction* model, are a common reference point in most TPM-based services. Therefore, since the intuition behind such an abstraction is the definition of a generic model that can serve as a specification of primitives for TPM operations, we believe that the decomposition of additional features and functionalities—needed by other application domains and environments—would require only minor *refinements* and the modelling of a small set of extra TPM commands. *This is mainly due to the fact that most of nowaday's TPM-based applications are using the already (FutureTPM) considered algorithms and protocols in an attempt to create trust aware service graph chains: namely, the DAA protocol, EA mechanism, and PCR management*. As an example, Smart and Ritter [22] have presented a remote electronic protocol that uses trusted computing and particularly a TPM for ensuring the trustworthiness of remote voters. In the description of this protocol, the commands that are needed belong in the set of core TPM commands that we have identified and modelled. Therefore, a reasonable future step would be to extend our models to other application domains as well.

The overall goal is to set the scene of a **refinement-based methodology** for proving security, privacy, and integrity guarantees of trusted platform modules and machine-checked proofs of refinements for models in the *Secure Mobile Wallet and Payments*, and *Activity Tracking* use cases. **This can act as evidence on the generality and applicability of the produced models to be considered as an extensible verification methodology for enabling rigorous reasoning about the security properties of Future TPMs**.[1]

Towards this direction, an important part in our modelling approach is to identify the parties that are involved in each scenario, as well as the TPM commands to be executed. Then, we will need to define the ideal functionality for each TPM command, as a refinement of our trusted platform command abstraction model—following the same process as we did in the context of the device management ecosystem. In what follows, we give a tentative description on how this

[1] All our models and proof scripts will be made open-source upon publication of the submitted papers. These models are designed to be modular and amenable to extension by the community

modelling approach could be applied in the other two use cases of the project, namely: Secure Mobile Wallet & Payments and Activity Tracking.

5.1 Secure Mobile Wallet and Payments Use Case

In the Secure Mobile Wallet use case, the core functionality that needs to be modelled is the *sealing* process of the underlying financial and user authentication tokens. This is translated to a series of interactions between a mobile device, running the FreePOS application installed, and a TPM that is attached to the device acting as the Root-Of-Trust. The motivation behind executing this type of sealing functionality is to protect the creation and usage of a TPM key to be binded to another secret factor (e.g., password) that needs to be provided by the user, as proof-of-possession, before allowing the further secure execution of a financial transaction (more information can be found in Deliverable D6.5).

The sealing process works as follows. The FreePOS application creates a random password, via the TPM, using the command `TPM2_GetRandom`. Then it creates an asymmetric encryption TPM key, following the process that we have already described in this report for the AK creation, but also in Deliverable D3.3 [10, Chapter 2]. More specifically, it executes the command `TPM2_StartAuthSession` in order to create a fresh trial session and then executes the command `TPM2_PolicyPCR` in order to update the policy digest of the trial session with the digest value of a PCR corresponding to the system configuration state. The command `TPM2_PolicyGetDigest` is executed in order to get the policy digest value under which the TPM key is created and protected using the command `TPM2_Create`. The authorization policy of the key is set as the policy digest of the trial session and, in addition, the authorization value of the key is set as the password that was previously generated. As we already described, such a TPM key can be loaded into the TPM using he process of EA authorization. That is, the FreePOS application creates a policy session using `TPM2_StartAuthSession` and calls the `TPM2_PolicyPCR` in order to update the policy digest of the policy session using a specific PCR value. The TPM key can be loaded using the command `TPM2_Load`, only if the policy digest of the policy session matches the authorization value of the key.

The difference in this scenario compared to what has been modelled in the context of the Device Management use case is that the sensitive area of the generated key contains valuable information (i.e., password[2]) which represents the authorization value of the key. This password is now sealed into a specified set of PCRs which, in turn, requires the successful execution of the `TPM2_Unseal` command in order to verify the correct state of the device before allowing the usage of the TPM key for further signing any financial transactions. This command requires the same authorization process as in the case of the `TPM2_Load`. It checks whether the policy digest of the policy session matches the authorization policy of the key and if this is true, it outputs the sensitive area of the key, or equivalently the password.

Therefore, it is obvious that the modelling of this scenario requires the use of most idealized functionalities already present in our trusted platform commands abstraction model (Deliverables D3.3 [10] and D3.4 [11]). We would only need to formalize the additional `TPM2_Unseal` command (as the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Unseal}}$); however, this would be rather straightforward since it would be very similar to the already existing $\mathcal{F}_{\texttt{TPM2\_Load}}$ functionality. The only difference would correspond to the returned value which in the case of $\mathcal{F}_{\texttt{TPM2\_Unseal}}$ should be the authorization

[2] In the actual demonstrator described in D6.5, this password is represented by the key handle extracted from he Yubiko hardware component

value of the key, hence, a randomly generated value referring to the password that was previously generated by the FreePOS app. This brief use case description, justifies our earlier claims on the generality and applicability of the produced verification methodology while requiring minimum refinements. *Consequently the model we have presented in Chapter 3 covers the modelling of the Secure Mobile Wallet wallet use case as well, meaning that it can be easily adjusted with only minor modifications.*

5.2 Activity Tracking Use Case

The main TPM functionality that needs to be modelled in the Activity Tracking use case, as described in Deliverable D3.2 [6], is the DAA protocol. In summary, DAA can be thought as a set of cryptographic primitives which are used in order to create anonymous digital signatures for enhanced user-controlled privacy. The DAA scheme is one of the most important TPM functionalities for supporting platform authentication while enabling the provision of privacy-preserving and accountable services. DAA is based on group signatures that allow remote attestation of a device associated to a Trusted Component (TC) while offering strong anonymity guarantees. Standardised by the Trusted Computing Group (TCG), DAA retains user anonymity, provides user-controlled unlinkability, and identifies signatures created by compromised devices.

In particular, DAA supports anonymous signing of a message, certification of TPM keys and remote attestation of the platform's state using records of the TPM's PCRs. Chen (2010) et. al [5] have identified the following properties of DAA schemes, which have been formally analyzed by Wesemeyer et al. (2020) [25].

1. *User-controlled anonymity*: Signer's identity cannot be revealed from a signature and signatures cannot be linked without the signer's consent.
2. *User-controlled traceability*: Signatures can be produced only by a TC and they can be linked only with the signer's consent.
3. *Non-frameability*: An adversary cannot produce a signature associated with an honest TC.
4. *Correctness*: Valid signatures cannot be forged and are verifiable, and linkable, where needed.

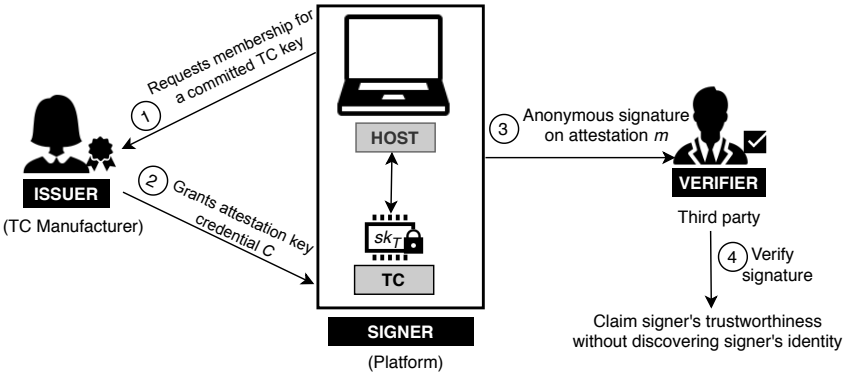

Figure 5.1: An overview of the entities involved in a DAA protocol

A typical DAA scheme consists of the following entities, as shown in Fig.5.1:

- **Signer**: A signer is a combination of a Host and a Trusted Component (TC). The host is a computing platform (e.g., a laptop). The TC is a tamper-resistant hardware device embedded on the host which provides integrity measurement of the host's software state, e.g., a Trusted Platform Module (TPM). A TC is uniquely identified by an endorsement key, sk_T, which is defined during the manufacturing stage and it is known only by the TC.
- **Issuer**: This is a trusted-third party that is responsible for verifying the correctness of the signer i.e., verifying that the key information derived by the TC is validated by the TC's manufacturer. Upon a successful verification, the Issuer grants the attestation key credential C to the signer. The credential C enables the signer to use a zero-knowledge proof protocol to anonymously sign the attestation message m obtained by TC.
- **Verifier**: Any other system entity or third-party that can verify a platforms' credentials in a privacy-preserving manner using DAA algorithms; without the need of knowing the platform's identity. Upon receiving an anonymous signature by the signer, it validates signer's integrity and authenticity, without discovering the signer's identity.

A DAA scheme enables a signer to prove the possession of the issued credential C to a verifier by providing a signature, which allows the verifier to authenticate the signer without revealing the credential C and signer's identity. In a nutshell, DAA is essentially a two-step process where, firstly, the registration of a TC executes once and during this phase the TC chooses a secret key ($SETUP$). This secret key is stored in secure storage so that the host cannot have access to it. Next the TC talks to the issuer so that it can provide the necessary guarantees of its validity ($JOIN$). The issuer then places a signature on the public key, producing the Attestation Identity Credential (AIC) C. The second step is to use this C for anonymous attestations on the platform ($SIGN$), using Zero-Knowledge Proofs. These proofs convince a verifier that a message is signed by some key that was certified by the issuer, without knowledge of the TC's *DAA key* or C ($VERIFY$). Of course, the verifier has to trust that the issuer only issues C to valid TCs.

In the Activity Tracking use case, there are four main entities participating in the protocol: a Personal Web Application (S5PersonalTracker or S5DataAnalysis) which is the Host device (e.g., users' mobile device) containing a TPM, the TPM itself acting as the Root-Of-Trust, the Issuer, and the Web Server & Analytics Engine (S5Tracker Analytics Engine). The S5Tracker Analytics Engine, acting as the Verifier, accepts connections from remote machines loaded with the S5PersonalTracker or S5DataAnalysis binaries (i.e. which either belong to individuals or to analysts), without being able to distinguish and identify to whom they belong to, while the authenticity of the Host machine is examined by the TPM. The instantiation of the DAA protocol in such an environment is illustrated in Figure 5.2.

The security modelling is, therefore, equivalent to the targeted modelling of the DAA protocol and particularly the three sub-phase, namely the *JOIN*, *SIGN* and *VERIFY* protocols. The TPM commands that are used for these sub-protocols are listed in Deliverable D3.2 [6]. These are:

`TPM2_StartAuthSession`	`TPM2_PolicyPCR`
`TPM2_Create/TPM2_Load`	`TPM2_Sign/TPM2_VerifySignature`
`TPM2_Certify`	`TPM2_MakeCredential/TPM2_ActivateCredential`

which have already been modelled and the corresponding ideal functionalities are presented in Deliverables D3.3 [10] and D3.4 [11]. Two additional TPM commands are mentioned in Deliverable D4.1 [8, Section 4.2], the `TPM2_Hash` and `TPM2_Commit` commands. The first can be excluded from the modelling, since it is used for creating a hash value. The second is used in

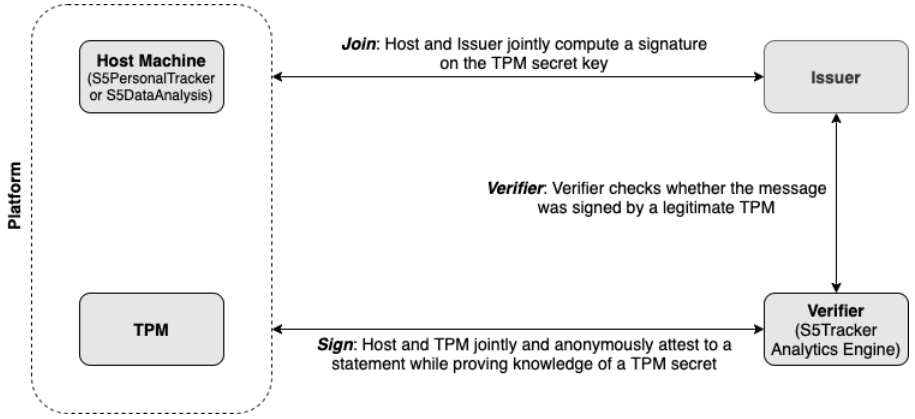

Figure 5.2: The DAA protocol in the case of activity tracking [8]

order to create certain values that will be used by the `TPM2_Sign` command and in particular for creating an anonymous signature. By the description of the `TPM2_Commit` command, included in the TPM specification manual [24, p.198], we can see that this also performs cryptographic operations based on the signing algorithm that is used; therefore, can be replaced with an equivalent equational theory operation.

Chapter 6

Conclusion

Recent proposals for trusted hardware platforms, such as TPMs and/or Intel SGX, offer compelling security features but lack formal guarantees. In this deliverable, we presented our complete work on the **security modelling of the TPM**—as a decentralized Root-Of-Trust—coupled with the **formalization and verification of its security properties**. More specifically, by leveraging the "*trusted platform command abstractions*", extracted in the previous versions of this deliverable, we compiled a: (*i*) newly introduced **verification methodology, based on a "bottom-up" approach,** in which the focus is on modelling the core TPM functionalities towards building chains of trust (instead of considering the TPM as a whole), and (*ii*) **formalization of idealized TPM functionalities** along with their security properties and a **realistic adversarial model**. *This break-down of TPM ideal functionalities and services allows for a more effective verification process towards building a global picture of the entire TPM platform security modelling as a Root-Of-Trust.* These models are designed to be modular and amenable to extension by the community.

The latter represents a formal model of a TPM command that captures the actions of the trusted platform module, when the command is executed, in such a way that excludes the cryptographic operations carried out internally and replaces them with non-cryptographic approaches. We essentially developed a **trusted abstract platform model consisting of a specific set of formally-specified primitives sufficient to implement the core TPM functionalities beyond the core crypto operations**. Such an abstraction modelling can enable the reasoning about and comparing different TPM services under various adversarial models and for different security guarantees, excluding any possible implications from the leveraged cryptographic primitives. For trusted platform module implementers, such a representation can be considered as a golden model for the expected system behaviour. From the perspective of formally verifying trusted hardware components, this model can provide a means of reasoning about security and privacy (of offered services) without being bogged down by the intricacies of various crypto primitives considered in the different platforms.

We also **formalized the notion of secure remote attestation towards trust aware service graph chains** (in the context of the envisioned device management use case) and presented Tamarin security proofs showing that our models satisfy the three key security properties that entail secure remote attestation and execution: **integrity, confidentiality, and secure measurement**. Furthermore, in order to model this service, we also considered additional TPM processes such as the creation of TPM keys, the Enhanced Authorization (EA) mechanism, the management of the Platform Configuration Registers (PCRs), and the creation and management of policy sessions. In this context, we have successfully identified and modelled all relevant TPM commands, used in these functionalities (as extracted in Deliverables D3.3 [10] and D3.4 [11]), and have also

extended our trusted abstract platform model with such idealized functionalities.

In order to simplify the modelling of the secure remote attestation and execution, in the context of the device management use case, and to avoid any unexpected behaviour from our Tamarin prover (e.g., non-termination), we have decomposed the overall service into three main phases: (*i*) the certification of the AK, (*ii*) the certification of the TLS key, and (*iii*) the establishment of a secure communication channel between the Router and the NMS. We proceeded with the security modelling of all these three parts and continued with the verification of the achieved properties, thus, enabling the evaluation of the overall soundness: ***A device, with a TPM attached, that satisfies secure measurement, integrity, and confidentiality properly also satisfies secure remote attestation.***

In this context, we have modelled the relevant security properties in the form of lemmas. Such security properties (aiming for integrity, confidentiality and secure measurements) include the secrecy and freshness of keys, correct transfer of information between the involved parties, availability of keys at honest parties, authentication properties, as well as sanity check lemmas that show the correct execution of the model.

Finally, we have also put forth number of challenges that were encountered during this modelling and verification process and the actions taken in order to overcome them. We have also argued that such a verification methodology, based on the use of trusted abstract platform models and idealized functionalities, is more than just a set of proofs of correctness of specific services (e.g., secure remote attestation) but can also enable the security modelling of the TPM as a whole merging various functionalities offered by the different abstraction layers. The trusted abstract platform model can serve as a specification of primitives of TPM operation, and is designed to be extensible towards additional features (as presented in Chapter 5) and additional guarantees against sophisticated attackers. **Overall, we believe that the produced models provide the baseline for an extensible verification methodology that enables rigorous reasoning about the security properties of *Future TPMs***.

Chapter 7

List of Abbreviations

Abbreviation	Translation
AE	Authenticated Encryption
AK	Attestation Key
CA	Certification Authority
CSR	Certificate Signing Request
DAA	Direct Anonymous Attestation
EA	Enhanced Authorization
EK	Endorsement Key
FQDN	Fully Qualified Domain Name
MSR	Multiset Rewriting Rule
NMS	Network Management System
PCA	Privacy Certification Authority
PCR	Platform Configuration Register
RA	Remote Attestation
SAPiC	Stateful Applied Pi Calculus
SKAE	Subject Key Attestation Evidence
TLS	Transport Layer Security
TPM	Trusted Platform Module
TTP	Trusted Third Party
WP	Work Package
ZTP	Zero Touch Provisioning

References

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[2] Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark D Ryan. Statverif: Verification of stateful processes. *Journal of Computer Security*, 22(5):743–821, 2014.

[3] David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. Tamarin prover (v. 1.4.1), January 2019. https://tamarin-prover.github.io/.

[4] Bruno BLANchet, V Cheval, X Allamigeon, and B Smyth. Proverif: Cryptographic protocol verifier in the formal model. *URL http://prosecco. gforge. inria. fr/personal/b-bLANche/proverif*, 2010.

[5] Liqun Chen. A daa scheme requiring less tpm resources. In Feng Bao, Moti Yung, Dongdai Lin, and Jiwu Jing, editors, *Information Security and Cryptology*, pages 350–365, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] The FutureTPM Consortium. First report on the security of the TPM. Deliverable D3.2, FutureTPM, June 2019.

[7] The FutureTPM Consortium. Technical integration points and testing plan. Deliverable D6.1, FutureTPM, July 2019.

[8] The FutureTPM Consortium. Threat modelling & risk assessment methodology. Deliverable D4.1, FutureTPM, February 2019.

[9] The FutureTPM Consortium. Demonstrators implementation report – first release. Deliverable D6.3, FutureTPM, April 2020.

[10] The FutureTPM Consortium. Second report on security models for the TPM. Deliverable D3.3, FutureTPM, February 2020.

[11] The FutureTPM Consortium. Second report on the security of the TPM. Deliverable D3.4, FutureTPM, September 2020.

[12] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[13] Ken Goldman. Attestation Protocols. Technical report, IBM, December 2017. https://www.ibm.com/developerworks/library/l-trusted-boot-openPOWER-trs/index.html.

[14] TCG Infrastructure Working Group et al. TCG Infrastructure Workgroup Subject Key Attestation Evidence Extension. *Specification Version*, 1, 2005.

[15] Steve Kremer and Robert Kunnemann. Sapic - a stateful applied pi calculus. http://sapic.gforge.inria.fr/.

[16] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.

[17] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*, pages 31–43. IEEE, 1997.

[18] Simon Meier. *Advancing automated security protocol verification*. PhD thesis, ETH Zurich, 2013.

[19] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on CAV*, pages 696–701. Springer, 2013.

[20] Jianxiong Shao, Yu Qin, and Dengguo Feng. Formal analysis of HMAC authorisation in the TPM2.0 specification. *IET Information Security*, 12(2):133–140, March 2018.

[21] Jianxiong Shao, Yu Qin, Dengguo Feng, and Weijin Wang. Formal analysis of enhanced authorization in the TPM 2.0. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 273–284. ACM, 2015.

[22] Matt Smart and Eike Ritter. True trustworthy elections: remote electronic voting using trusted computing. In *International Conference on Autonomic and Trusted Computing*, pages 187–202. Springer, 2011.

[23] Trusted Computing Group (TCG). TPM 2.0 library specification - part 1: Architecture. Available at: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf.

[24] Trusted Computing Group (TCG). TPM 2.0 library specification - part 3: Commands - code. Available at: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_code_pub.pdf.

[25] Stephan Wesemeyer, Christopher J. P. Newton, Helen Treharne, Liqun Chen, Ralf Sasse, and Jorden Whitefield. Formal analysis and implementation of a TPM 2.0-based direct anonymous attestation scheme. In Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese, editors, *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 784–798. ACM, 2020.