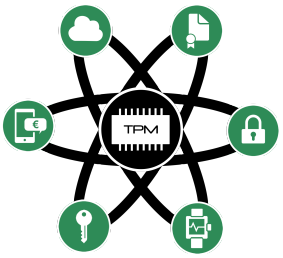

FutureTPM

D3.4

Second Report on the Security of the 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 / D3.4 / v1.1
Workpackage:	WP 3
Due Date:	July, 2020
Actual Submission Date:	30 September, 2020
Responsible Organisation:	SUR
Editor:	Georgios Fotiadis , José Moreira
	Kaitai Liang
Dissemination Level:	PU
Revision:	v1.1
Abstract:	In this report, we deliver the main contributions towards modelling of TPM abstractions, with the predefined ideal functionalities, and showcase how this can be integrated in the security modelling for a specific application domain (Secure Device Management).
Keywords:	TPM, modelling, ideal functionalities

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 order to provide the FutureTPM project's envisioned services with the appropriate levels of *security*, *privacy* and *assurance*, we need to define trust models that are able to capture the complex relationships between all involved entities and components. This model must not only capture the FutureTPM and the applications (and use cases) that rely on it, but also the TPM-based environment in which they operate, as described in D1.1 [8].

Towards this direction, this deliverable reports on the current progress on modeling the TPM functionalities as *abstractions*: **ideal functionalities in order to be able to use them in models for complex protocols such as the ones used in the envisioned use cases including remote attestation, policy-based key management, etc.** This concept of "idealized functionalities" mean that we only expose to the (TPM) client applications, those crypto operations (of the TPM) that are relevant to the application at hand, and we abstract from the models all the TPM crypto functionalities that are for self-consumption (e.g., any crypto related to secure-storage, key hierarchies, etc. that is internally used by the TPM). That is, in addition of assuming that "cryptography is perfect", as in the symbolic model, we also assume that "TPM cryptograpy operations are perfect". **Thus, our goal is to be able to identify the right trust model that will enable us to express fine-grained functionalities of the TPM beyond the leveraged crypto.**

The focus is on further exploring the modelling of one of the FutureTPM use case and more specifically the Device Management Scenario [8]. We proceed with a particular instance of this use case, in which the main objective is to establish a secure TLS communication between a Router and a Network Management System (NMS) in a network infrastructure. We first identify the TPM commands that are needed in this scenario. These are related to the creation of TPM keys which are sealed to specific PCR values by exploiting the Enhanced Authorization (EA) mechanism, particularly PCR policies and in addition, these commands include certification, credential management, encryption and signature operations, which are essential for remote attestation.

For remote attestation, based on the Device Management use case, we opted for following the generic version in the literature which is the IBM attestation protocol. Then we provide the ideal functionalities for those commands based on all identified security and trust requirements requirements. We present the security modelling for the creation of Attestation Key (AK) and TLS key as well as the IBM attestation protocol, which is used in order to certify the AK and finally, we summarize some preliminary security features that should be considered in this reference scenario.

Contents

List of Figures

Chapter 1

Introduction

The main objective of Work Package (WP) 3, towards the security modelling of the Trusted Platform Module (TPM), is to model the TPM commands in such a way so that we formally capture the security of the core TPM's functionalities, by excluding the cryptography that is used internally (e.g. hashing or asymmetric encryption) and that can be considered secure. We refer to this model of TPM commands as "ideal functionalities". This concept of "idealized functionalities" mean that we only expose to the (TPM) client applications, those crypto operations (of the TPM) that are relevant to the application at hand, and we abstract from the models all the TPM crypto functionalities that are for self-consumption (e.g., any crypto related to secure-storage, key hierarchies, etc. that is internally used by the TPM).
Instead of working with the whole TPM command set, we identify a specific subset of those commands that are of core importance; they are widely used in most reference scenarios in the literature, including various application domains as the ones envisioned through the FutureTPM use cases. Such commands are related to **object creation and loading in the TPM, policy-based authorization and session management, remote attestation and those using TPM objects for cryptographic purposes**, such as for signing or for encryption. The modelling of this subset of TPM commands will serve as a good baseline and can be enhanced in order to formally capture the security of more complex scenarios and protocols, as well as additional functionalities, in the context of the use cases of the project and beyond. **Our goal is to be able to identify the right trust model that will enable us to express fine-grained functionalities of the TPM beyond the leveraged crypto.**

As a starting point, we have chosen to focus on the modelling of the Device Management use case, which is described in detail in Deliverables D4.1, D6.1 and D6.3 [10, 9, 11]. The network device management demonstrator intends to show how system integrity challenges can be solved, at scale, in the scenario of a distributed telecommunications infrastructure composed of many network devices that are centrally managed. In the demonstrator, network routers equipped with a QR-TPM are required to prove their hardware identity and software integrity to a Network Management System (NMS). The process is integrated with the usual management operations that the NMS is performing across the entire lifecycle of the router, from deployment stage through regular operation until their decommissioning, by leveraging the concept of Remote Attestation. Based on the outcome of this process, the NMS can decide whether any given router can be trusted for routing user traffic or, if it cannot be trusted, whether it should be avoided, e.g. by adjusting the routing policy on its neighbouring routers.
Due to the nature of this application domain, it serves as an excellent baseline for our modelling due the fact that it considers all core TPM functionalities that are a common reference in most TPM-based applications in the literature. These include the object creation operations, policy

session management (especially Platform Configuration Register (PCR) policy), the EA mechanism for session-based authorization and also remote attestation, which can be accomplished either by the TPM Direct Anonymous Attestation protocol, which is a built-in TPM functionality, or by more generic protocols, such as the IBM remote attestation protocol [14].

Towards this abstract mdelling, we are going to use the abstract description of TPM commands [12] that are used in order to create a TPM key and load it into the host trusted platform. These commands have been modelled, in the context of D3.3 [12], as ideal functionalities, in other words a model that idealizes all TPM functionalities except those that serve to provide cryptography to a consumer application and further captures their intended semantics rather than their actual implementation. The key creation process is based on the Enhanced Authorization (EA) mechanism which is a new feature in the TPM2.0. In particular, following the description of the Device Management use case, there are keys that need to be sealed to specific PCR values and hence they are linked to a PCR policy. This is a common practice in other TPM scenarios as well, in addition to the use cases of this project.

1.1 Methodology

From a high-level perspective, the Device Management use case consists of three main entities: a (set of) Routers equipped with a TPM, a Remote Attestation (RA) Server and the Network Management System (NMS), which is responsible for managing the Routers through Transport Layer Security (TLS) connections. In order to establish this TLS channel, each new Router needs to create a TLS key via the TPM and this TLS key has to be certified by the RA Server. For this certification, the Router needs to first create an Attestation Key (AK), which is also certified by the RA Server. This later process is achieved through the IBM remote attestation protocol. In this deliverable, we continue the work that was started in D3.3 and we focus on the modelling of the creation of both the AK and TLS keys, as well as the certification of the AK. The process of certifying the TLS key is ongoing and will be integrated in our model in the next and final deliverable (D3.5).

We start our modelling approach by first presenting an abstract description of additional TPM commands that are needed in order to create the AK and TLS keys and for their certification. In particular, these are `TPM2_Certify`, `TPM2_MakeCredential`, `TPM2_ActivateCredential`, as well as `TPM2_Sign`/`TPM2_VerifySignature` and `TPM2_PK_Encrypt`/`TPM2_PK_Decrypt`. The aim of this abstract description is to realize how these commands operate in real-life, based on the TPM specification manuals [22, 23]. These TPM commands are necessary in this use case, in which they reflect (i) credential operations, (ii) secure data communication via encryption, and (iii) data integrity verification via signature. We also present a detailed description of the IBM remote attestation protocol [14], which is used in the certification of the AK. In the description, we make use of the commands `TPM2_MakeCredential` and `TPM2_ActivateCredential` which are basically used to provide a “challenge-response” mode of remote attestation.

The next step is to model these additional TPM commands, in terms of ideal functionalities, as it was done in Deliverable D3.3. For the modelling of these ideal functionalities we use the Stateful Applied Pi Calculus (SAPiC) [15] language, which was also used in Deliverable D3.3. When integrating the ideal functionalities of D3.3 in our model, we realized that certain updates need to be done. Thus, we also present a revised version of these ideal functionalities, in particular for the commands `TPM2_Create` and `TPM2_StartAuthSession` in Chapter 3, where we also justify the reasons for these revisions. Finally, we present an overview of the modelling tools we are going

to use, introduce the technical roadmap and challenges in the use case modelling and model the AK and TLS key creation and IBM attestation protocol. We also present a preliminary set of security properties that should be tested in the use case.

1.2 Structure of the Report

Chapter 2 provides the abstract description of TPM commands which are needed for the deployment of the Device Management use case and the description of the IBM attestation protocol. In Chapter 3, we present the modelling of the new TPM commands as ideal TPM functionalities and the revised version of specific ideal functionalities from D3.3. Finally, Chapter 4 presents the security modelling for creating the AK and TLS keys as well as the certification of the AK with the IBM remote attestation protocol and further mention some security features.

Chapter 2

TPM Commands and Related Protocols

In Deliverable D3.3 [12] we presented an abstract description of the TPM commands that are used in the process of creating and loading a TPM key. In particular, we focused on a specific instance of EA, where the TPM key to be created is linked to a certain Platform Configuration Register (PCR) value, via a PCR policy session. This was done due to the remote attestation and configuration integrity verification process of the device management use case, in which the TPM keys that are created are linked with a PCR policy. In this chapter we will present an abstract description of additional TPM commands that are needed in order to integrate the model of D3.3 to the device management use case, as this is described in D6.3 [11].

In brief, the commands that we describe here are used by the RA Client, running in the Router, which communicates with a TPM that is embedded in the Router. This interaction between the RA Client and the TPM is used in order to the RA Client to create an Attestation Key (AK) and a TLS key as well as in the certification of both by the RA Server. In particular, the certification of the AK by the RA Server is accomplished by the IBM remote attestation protocol, which we describe in detail in Section 2.2. The exact scenario that uses the commands we describe here, as well as the entities that are involved, is presented in Chapter 4. This corresponds to the user story HWDU.NO.1 of the device management use case [11, page 38]. We note here that in Deliverable D3.3 we have used the notation `tpm.CommandName` for describing specific TPM commands. From now on we will use the notation `TPM2_CommandName` in order to be in line with the TPM2.0 specification manual [22, 23, 24].

2.1 Abstract Description of TPM Commands

The main TPM commands for creating and loading a key into the TPM are `TPM2_PCRExtend`, `TPM2_StartAuthSession`, `TPM2_PolicyPCR`, `TPM2_Create` and `TPM2_Load`. These are described in D3.3 [12, Section 2.2]. In order to model the device management use case, we will also need to describe the following additional TPM commands:

1. `TPM2_Certify`: provides a specific type of attestation. A TPM attests to an asymmetric key pair in order to vouch that a key pair is protected by a genuine but unidentified TPM and has particular properties. This attestation has the form of a signature signed by the TPM over information that describes the key pair. The signature is created using an AK protected by the TPM where the AK also needs to be attested by the “Attestation CA” [22].

2. `TPM2_MakeCredential`: uses a TPM public key in order to protect a given secret. Depending on the usage of the command this secret can be used directly to encrypt an AK certificate, or it can be used as a challenge before issuing the AK certificate.

3. TPM2_ActivateCredential: verifies that the secret in TPM2_MakeCredential is indeed protected by the right TPM public key and it is bound to the right AK and returns the secret.
4. TPM2_PK_Encrypt/TPM2_PK_Decrypt: refer to public key encryption/decryption. In particular they correspond to TPM2_RSA_Encrypt and TPM2_RSA_Decrypt, but we are replacing the "RSA" by "PK", so that the commands are not dependent on the underlying algorithm.
5. TPM2_Sign/TPM2_VerifySignature: refer to the process of signing a message and verifying the signature.

As in D3.3, our abstract description of the above TPM commands is based on the TPM Specification Parts 1 and 3 [22, 24].

Certify an object: TPM2_Certify **[24, page 170]**

It is used as a specific type of attestation, where a TPM attests to an asymmetric key pair, via a signature that is created with an AK that is certified by an external entity called the "Attestation CA". Hence, this command proves that an object (key) with a specific name is loaded in the TPM. This proof implies that the public area of the object (public key) is self consistent and it is associated with a valid sensitive area (secret key) [24]. The command takes the following input:

- objectHandle: The handle of the object to be certified. Session-based authorization (policy or HMAC) is needed to access the content of the objectHandle.
- signHandle: The handle of the signing key (AK) that will be used to sign the attestation structure. Session-based authorization (policy or HMAC) is needed to access the content of the signHandle.
- qualifyingData: Additional data provided by the user that will be included in the attestation structure. This is usually a nonce to indicate the freshness of the attestation [22].
- inScheme: The asymmetric signature scheme that is used to sign the object.

In our case, the authorization for accessing both handles is performed via PCR policy sessions. This means that the TPM will lookup for the authPolicy of each key and will compare it with the policy digest value of the corresponding policy session. If the two comparisons are successful, the TPM will allow access to the contents of the two keys. The TPM will lookup for the Name and the qualified name qualName of the two keys in the corresponding creationData field (see Section 2.2.1 in D3.3 [12]) and will create the attestation structure attestStruct, based also on some additional information in qualifyingData, if any, provided by the user. Hence, the attestation structure has the form:

$$\texttt{attestStruct} \leftarrow (\texttt{Name1}, \texttt{qualName1}, \texttt{Name2}, \texttt{qualName2}, \texttt{qualifyingData}),$$

where Name1, qualName1 refer to the object's name and qualified name and Name2, qualName2 refer to the name and qualified name of the AK that is used for signing. Then the TPM will sign the attestation structure and it will output the following parameters:

- certifyInfo: This is the attestation structure that was signed by the TPM.
- signature: The signature on certifyInfo, using the signing key that is referenced in signHandle (AK).

An abstract description of the command TPM2_Certify is given in Figure 2.1.

```
TPM2_Certify(objectHandle, signHandle, qualifiedData, inScheme)
 1: objectAuthCheck ⟵ Authorization()        //authorization check for objectHandle
 2: if objectAuthCheck == TRUE then
 3:   signAuthCheck ⟵ Authorization()          //authorization check for signHandle
 4:   if signAuthCheck == TRUE then
 5:     retrieve Name1 and qualName1 from objectHandle
 6:     retrieve Name2 and qualName2 from signHandle
 7:     retrieve SIGN algorithm from inScheme
 8:     retrieve secret signing key sk from signHandle
 9:     attestStruct ⟵ (Name1, qualName1, Name2, qualName2, qualifyingData)
10:     certifyInfo ⟵ attestStruct
11:     signature ⟵ SIGN(sk, certifyInfo)
12:     output(certifyInfo, signature)
13:   else output(FAIL)
14: else output(FAIL)
```

Figure 2.1: `TPM2_Certify()` command for certifying an object

Create a credential: `TPM2_MakeCredential` **[24, page 72]**

This command protects a “plaintext" using a public key, and binds this “plaintext” to an object `Name`. This “plaintext" can be a message, a key, a challenge, a piece of secret information, etc. It takes the following input:

- `handle`: The handle of the loaded public area `pk` of a Storage key that will be used to protect the “plaintext” information and to encrypt the seed which will be generated in the creation of the `credentialBlob`.
- `credential`: This refers to the “plaintext” that will be protected by the Storage key referenced in `handle`. In other words it is the secret information the user wishes to protect.
- `objectName`: The name of the object to which the `credential` is bound. This is contained in the field `creationData` in the key structure.

Storage keys are encryption keys and can be either asymmetric or symmetric. In the case of the `TPM2_MakeCredential` command, the Storage key is asymmetric and its public part referenced in `handle` is loaded in the TPM[1]. Therefore, no authorization is required in order to use the public part of the underlying Storage key. The TPM creates a value which is named `credentialBlob`. This is a TPM custom mode of (symmetric) Authenticated Encryption (AE) of `credential` that is bound to `objectName` under a symmetric encryption key k_1 and MAC key k_2. In particular, the value `credentialBlob` is generated in the following way:

$$\texttt{credentialBlob} \leftarrow [\texttt{Enc}(\texttt{k}_1, \texttt{credential}), \texttt{HMAC}(\texttt{k}_2, (\texttt{Enc}(\texttt{k}_1, \texttt{credential}) || \texttt{objectName}))],$$

where “||” denotes concatenation[2]. The TPM also computes the value `secret`. This corresponds to the encryption of the value `seed` that was used to derive the keys k_1, k_2, with the public part of

[1] It is explicitly mentioned in [24] that the loaded public area of the key referenced in `handle` must correspond to a Storage Key, otherwise the credential cannot be properly sealed.

[2] This is similar to the Encrypt-then-MAC (EtM) AE mode, in which the `credentialBlob` would be created as:

$$\texttt{credentialBlob} \leftarrow [\texttt{Enc}(\texttt{k}_1, \texttt{credential}), \texttt{HMAC}(\texttt{k}_2, \texttt{Enc}(\texttt{k}_1, \texttt{credential})].$$

The difference is that the TPM custom AE concatenates the `objectName` after the ciphertext $\texttt{Enc}(\texttt{k}_1, \texttt{credential})$.

```
TPM2_MakeCredential(handle, credential, objectName)
 1: retrieve pk from handle
 2: create seed
 3: create symmetric enc key:  k1 ⟵ KDF1(seed||objectName)
 4: create MAC key:  k2 ⟵ KDF2(seed)
 5: c ← Enc(k1, credential), hmac ⟵ HMAC(k2, (c||objectName))
 6: credentialBlob ⟵ (c, hmac), secret = Enc(pk, seed)
 7: output(credentialBlob, secret)
```

Figure 2.2: `TPM2_MakeCredential()` command for protecting an object

the storage key. In other words $\texttt{secret} = \texttt{Enc}(\texttt{pk}, \texttt{seed})$. The output of the `TPM2_MakeCredential` command is the `credentialBlob` and the ciphertext `secret`. The command is summarized in Figure 2.2.

Activate a credential: `TPM2_ActivateCredential` **[24, page 68]**

This command is used in order to activate a credential that is previously generated using the command `TPM2_MakeCredential`. This means that the command `TPM2_MakeCredential` cre-

```
TPM2_ActivateCredential(activateHandle, keyHandle, credentialBlob, secret)
 1: keyAuthCheck ⟵ Authorization()              //authorization check for keyHandle
 2: if keyAuthCheck == TRUE then
 3:   objectAuthCheck ⟵ Authorization()    //authorization check for activateHandle
 4:   if objectAuthCheck == TRUE then
 5:     validationCheck ⟵ ValidateInput()     //check properties for activateHandle
 6:     if validationCheck == SUCCESS then
 7:       get sk from keyHandle
 8:       get objectName from activateHandle
 9:       seed ⟵ Dec(sk, secret)
10:       get c, hmac from credentialBlob
11:       recreate MAC key:  k2 ⟵ KDF2(seed)
12:       checkHMAC ⟵ ValidateHMAC(k2, (c||objectName), hmac)
13:       if checkHMAC == SUCCESS then
14:         recreate symmetric decryption key:  k1 ⟵ KDF1(seed||objectName)
15:         credential ⟵ Dec(k1, c)
16:         output(credential)
17:       output(FAIL)
18:     output(FAIL)
19:   output(FAIL)
```

Figure 2.3: `TPM2_ActivateCredential()` command for activating a credential

ates an association between a "plaintext" with an object `Name`, but in order for this association to be enabled, the command `TPM2_ActivateCredential` needs to be executed. The command takes the following input:

- `activateHandle`: The handle of the object that is associated with the `credential` value in the `credentialBlob`.

- keyHandle: The handle of the key that was used to encrypt the seed.
- credentialBlob: This is the first output of the command TPM2_MakeCredential.
- secret: The second output of the command TPM2_MakeCredential, basically the encryption of the seed with the public part of the key referenced in keyHandle.

The use of both keys referenced in activateHandle and keyHandle require authorization (policy or HMAC). For the key that is associated with the credential, extra checks are performed in order to ensure that the key is asymmetric and that it is a restricted decryption key. The private part sk of the key referenced in keyHandle is needed to retrieve the seed that was used in the creation of the symmetric encryption key k_1 and the MAC key k_2 in the TPM2_MakeCredential command. The Name of the key in activateHandle is required in order to reconstruct the symmetric encryption key k_1, as well as for the verification of the HMAC value that was used to create the credentialBlob (this is done with ValidateHMAC$(k_2, (c||\text{objectName}), \text{hmac})$). The output of the command is the initial credential. The abstraction of the activateHandle command is summarized in Figure 2.3.

Sign a message: TPM2_Sign [24, page 209]

The command TPM2_Sign is used in order to create a signature over a message that is provided by the user, using either a symmetric or an asymmetric signing key. For signatures that are generated by symmetric algorithms (HMAC, SMAC), the signing key is unrestricted. In the case

```
TPM2_Sign(keyHandle, digest, inScheme, validation)
 1: keyAuthCheck ⟵ Authorization()              //authorization check for keyHandle
 2: if keyAuthCheck == TRUE then
 3:   keyValidation ⟵ ValidateKey()              //validation check for signing key
 4:   if keyValidation == SUCCESS then
 5:     validationCheck ⟵ ValidateInput()                        //validate digest
 6:     if validationCheck == SUCCESS then
 7:       retrieve ssk from keyHandle
 8:       retrieve SIGN algorithm from inScheme
 9:       signature ⟵ SIGN(ssk, digest)
10:       output(signature)
11:     else output(FAIL)
12:   else output(FAIL)
13: else output(FAIL)
```

Figure 2.4: TPM2_Sign() command for signing a digest

of asymmetric signing algorithms, the key is restricted. That is, a particular asymmetric signing key can be used in order to sign a digest value that is produced by the TPM, while in addition, a validation needs to be provided, showing that the digest value is indeed created by the TPM. The command takes the following input:

- keyHandle: The handle of the signing key that will be used for signing.
- digest: The digest value that will be signed. This is produced by the TPM.

- `inScheme`: A reference to signing algorithm that will be used. The signing key needs to be compatible with the chosen algorithm.
- `validation`: If the signing key is restricted, this value is a proof that the `digest` is produced by the TPM.

The abstract description of this command is presented in Figure 2.4. The use of the signing key requires authorization, using sessions. In our case, this corresponds to PCR policy session-based authorization. `ValidateKey()` refers to the the process of checking whether the signing key is compatible with the chosen algorithm that is referenced in `inScheme`. On the other hand, `ValidateInput()` is the process of checking the validity of the `digest` value, i.e. whether the digest is produced by the TPM. The output of the command is the signature, generated by the TPM, or `FAIL`, if either the key authorization process, or at least one of the validity checks fails.

```
TPM2_VerifySignature(keyHandle, digest, signature)
1: retrieve psk from keyHandle
2: validation ← VerifySIGN(psk, digest, signature)
3: output(validation)
```

Figure 2.5: `TPM2_VerifySignature()` for verifying a signature generated by `TPM2_Sign`

The verification of a signature that is generated by the TPM command `TPM2_Sign` is done with the command `TPM2_VerifySignature`. This is summarized in Figure 2.5. Note that the use of the public signing key `psk` requires no authorization, since the command uses keys that are already loaded in the TPM [24]. In addition, `VerifySIGN()` refers to the verification of the signature and it depends on the algorithm that was used for creating the signature.

Encrypt/Decrypt a message: `TPM2_PK_Encrypt/TPM2_PK_Decrypt` **[24, pages 100, 104]**

We assume that the command `TPM2_PK_Encrypt` refers to the asymmetric encryption process, using a quantum-resistant cryptographic algorithm. The abstract description we will present here will be independent from the chosen algorithm. It takes the following input:

- `keyHandle`: The handle of the asymmetric encryption key.
- `message`: The message to be encrypted.
- `inScheme`: A reference to encryption and padding algorithms that will be used.
- `label`: An optional label associated to the `message`. This is used in the descryption, where the association with the message might need to be verified.

The command process is described in Figure 2.6. Note that as in the case of `TPM2_Sign`, the validity of the public key needs to be verified, but no authorization is required. The command outputs the encrypted message `outData`. The decryption process is performed via the `TPM2_PK_Decrypt` command, which takes as input the handle `keyHandle` of the public key pair, whose public part was used for encryption, the ciphertext, the `inScheme` parameter which refers to the padding algorithm that was used during encryption and the field `label`, which as we pointed out earlier is an optional field in case the message needs to be verified. Since the private part `sk` of the key referenced in `keyHandle` is used, authorization for accessing this private key is required. Recall all key authorizations in our model will be based on PCR policies. The command outputs either the original plaintext `message`, of `FAIL`, depending on whether the authorization was successful.

```
TPM2_PK_Encrypt(keyHandle, message, inScheme, label)
1: keyValidation ⟵ ValidateKey()                    //validation check for signing key
2: if keyValidation == SUCCESS then
3:   retrieve pk from keyHandle
4:   retrieve ENC algorithm from inScheme
5:   outData ⟵ ENC(pk, message)
6:   output(outData)
7: else output(FAIL)
```

Figure 2.6: TPM2_PK_Encrypt() command for asymmetric encryption of a message

```
TPM2_PK_Decrypt(keyHandle, ciphertext, inScheme, label)
1: authCheck ⟵ Authorization()                      //authorization for secret key
2: if authCheck == TRUE then
3:   retrieve sk from keyHandle
4:   retrieve DEC algorithm from inScheme
5:   message ⟵ DEC(sk, ciphertext)
6:   output(message)
7: else output(FAIL)
```

Figure 2.7: TPM2_PK_Decrypt() command for asymmetric decryption

2.2 The IBM Remote Attestation Protocol

In this section we give a brief and simplified description of the IBM remote attestation protocol [14] that is used in the device management use case, in particular in HWDU.NO.1 [11, §3.1.3.1], for the certification of the AK and it can be viewed as a "challenge-response" protocol between a client and a server. In the device management use case, the IBM remote attestation protocol is executed between the RA Client, which is equipped with a TPM and the RA Server. This interaction that captures the certification of the AK by the RA Server is summarized in Figure 2.8. The RA Client sends to the RA Server its Fully Qualified Domain Name (FQDN), the public part of the AK $\mathtt{ak}_{\mathtt{psk}}$ and the certificate of the EK $\mathtt{cert}_{\mathtt{M}}(\mathtt{ek}_{\mathtt{pk}})$, which is an X509 certificate containing the signature of the $\mathtt{ek}_{\mathtt{pk}}$ with the TPM Root Endorsement Key (EK) (this is denoted as M). This initiates the *enrollment process*. Upon receipt, the RA Server first performs certain validation checks on the received values. In particular, it checks whether this particular FQDN is already enroled, whether the $\mathtt{cert}_{\mathtt{M}}(\mathtt{pk}_{\mathtt{ek}})$ is valid and whether the AK satisfies the required properties e.g. the AK must be a signing restricted key and compatible with a specific signing algorithm. Then the RA Server proceeds with the creation of the challenge, in order to verify that the EK and AK are indeed created by the specific TPM.

This challenge is the result of the command TPM2_MakeCredential, which as described in Figure 2.2. Note here that the RA Server does not have to be equipped with a TPM in order to execute this command. It can either emulate the TPM2_MakeCredential command by following the exact same steps, as it is described in Figure 2.8 (steps 4–10), or for convenience, it can use a local TPM in order to execute the command. In any case, it requires the public part of the EK, which is extracted by the $\mathtt{cert}_{\mathtt{M}}(\mathtt{ek}_{\mathtt{pk}})$ certificate (step 2 in Figure 2.8) and the name akName of the AK (step 3 in Figure 2.8). According to [23, page 99], the name of an object that is created in a TPM corresponds to the hash of its public part. Steps 4–10 correspond to the steps of the

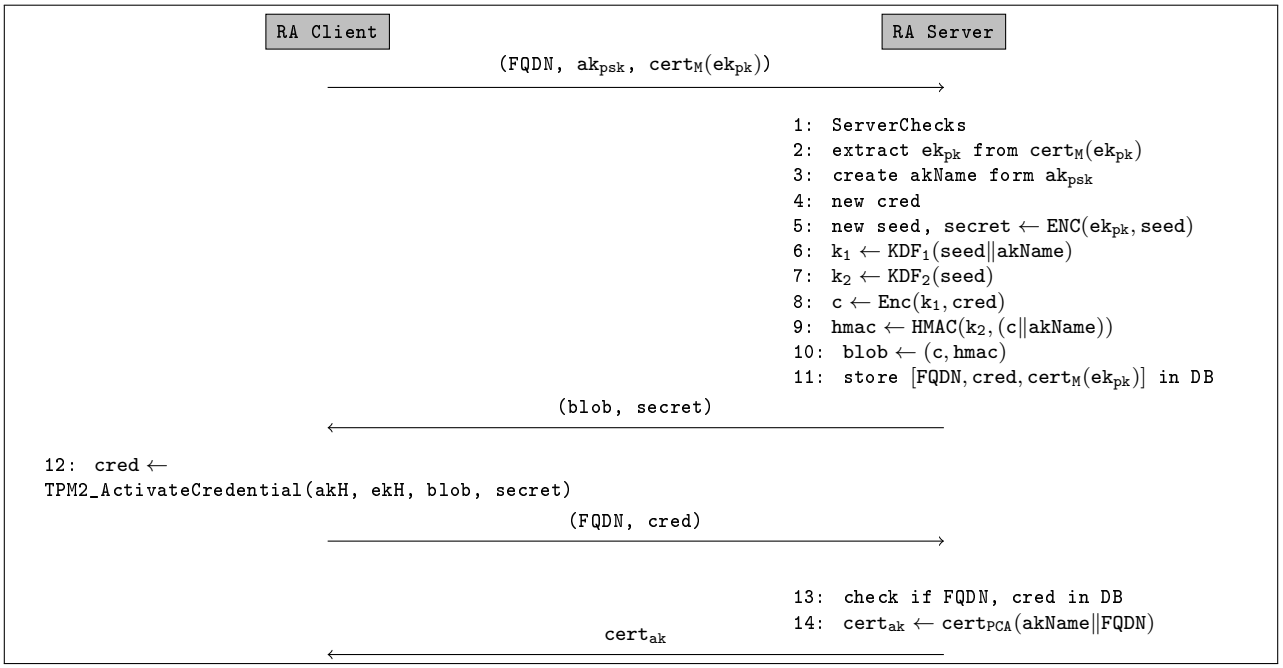

Figure 2.8: IBM remote attestation protocol

`TPM2_MakeCredential` command, where `cred` is the protected value generated by the RA Server and `(blob, secret)` is the RA Server's challenge which is sent to the RA Client. Further, the RA Server stores the triple $[\texttt{FQDN}, \texttt{cred}, \texttt{cert}_M(\texttt{ek}_{pk})]$ in its Database `DB` (step 11 in Figure 2.8).

The RA Client upon receiving the challenge `(blob, secret)` it executes, via the TPM, the command `TPM2_ActivateCredential`, with input the handle `akH` of the AK, the handle `ekH` of the EK and the pair `(blob, secret)`, in order to verify the validity of the received parameters (see Figure 2.3 for the checks that are performed). The command will return the credential `cred` that was initially generated by the RA Server (step 12 in Figure 2.8). Then the RA Client sends back to the RA Server the pair `(FQDN, cred)`. The RA Server checks that the received values are matching the corresponding entry in the DB (step 13 in Figure 2.8) and asks the Privacy Certification Authority (PCA) to create a certificate for this AK, namely $\texttt{cert}_{ak} \leftarrow \texttt{cert}_{PCA}(\texttt{akName}||\texttt{FQDN})$ (step 14 in Figure 2.8). This is returned to the RA Client in clear and hence the RA Client now has the certificate $\texttt{cert}_{ak}$ of the AK. A successful completion of the protocol implies that the RA Client is enrolled at the RA Server.

Chapter 3

Modelling TPM Commands

In Deliverable D3.3 [12, Section 2.3] we presented the modelling of specific TPM commands that are related to the process of creating a TPM key and loading it into the TPM, where a TPM key is associated to a PCR policy. In this chapter we model the additional TPM commands that are described in Section 2.1, which are necessary in order to model the process of creating an AK and a TLS key, as well as their certification.

Following Deliverable D3.3 [12, Section 2.3], we use the term "ideal functionality" which refers to our modelling approach for each TPM command. The key feature of our concept of the ideal functionality, is that it idealizes all TPM functionalities except those that serve to provide cryptography to a consumer application (e.g., hashing or asymmetric encryption), capturing their intended semantics rather than their actual implementation. As mentioned in D3.3, idealizing cryptography that is used internally allows one to study the security of applications and systems that rely on the TPM by: i. checking that the TPM is used in a way that allows this idealization; and ii. analyzing only cryptography relevant to the application itself.

An example of idealizing cryptography is also presented in [12, Section 2.1.1], concerning a scenario where EA authorization is used by an honest user in order to create a TPM-protected (non-primary) object sealed to a PCR state (that is, protected by a policy constructed by measuring the state of some subset of the PCRs). Recall that in reality, the TPM key needs to be bound to the policy digest value of a PCR policy session. That is, the authorization policy of the key must be set as the digest value of the policy digest of a PCR policy, which is constructed iteratively through hash chaining the parameters of successive policy commands. In our ideal functionality, we model this by keeping all measurements in a policy digest list $\mathtt{pD}$, where in each new policy evaluation we simply append the new measurement in this list.

For the rest of this report, the ideal functionality of a command $\mathtt{TPM2\_CommandName}$ will be denoted as $\mathcal{F}_{\mathtt{TPM2\_CommandName}}$. In addition and following Deliverable D3.3 [12, Section 2.3], our modelling of the ideal functionality $\mathcal{F}_{\mathtt{TPM2\_CommandName}}$ will be presented in SAPiC [15]. For the syntax in SAPiC, see Table 4.2, or [15].

3.1 Additional Ideal TPM Functionalities

Ideal functionality for certifying an object: $\mathcal{F}_{\mathtt{TPM2\_Certify}}$

The ideal functionality for certifying a key is described in Figure 3.1. It takes as input the handle $\mathtt{obj_h}$ of the object that we wish to certify, the corresponding session handle $\mathtt{sH}$ that was used in the creation of that object and the handle of the AK $\mathtt{ak_h}$. As the description of the command

TPM2_Certify suggests (see Figure 2.1), both keys need proper authorization. However, as we

$\mathcal{F}_{\texttt{TPM2\_Certify}} :=$

```
1: in(<'Certify', obj_sh, obj_h, ak_h>);        //input: key handles and object's session
   handle
2: lock obj_sh; lock obj_h; lock ak_h;
3: lookup <policyDigest, obj_sh> as pD in      //get policy digest from policy session
4: lookup <authPolicy, obj_h> as aP in              //get authPolicy value from object
5: if aP == pD then
6:   lookup <publicPart, obj_h> as obj_pk in                //get object's public part
7:   lookup <publicPart, ak_h> as ak_spk in                      //get AK public part
8:   lookup <privatePart, ak_h> as ak_ssk in                   //get AK private part
9:   let certifyInfo = <obj_pk, ak_spk> in
10:  let signature = revealSign(certifyInfo, ak_ssk)
11:  out(certifyInfo, signature);
12:  unlock obj_sh; unlock obj_h; unlock ak_h
13: else unlock obj_sh; unlock obj_h; unlock ak_h
```

Figure 3.1: Ideal functionality $\mathcal{F}_{\texttt{TPM2\_Certify}}()$

discussed in the previous section, in our modelling we assume that the AK is not bound to any policy and hence authorization for the AK is not required.

The ideal functionality checks whether the authorization policy aP of the object matches the policy digest value pD of the corresponding session and if this is true, it retrieves the public parts of the object and the AK, as well as the private part $\texttt{ak}_{\texttt{ssk}}$ of the AK. This will be used to sign the attestation structure (or certifyInfo) and the ideal functionality will return both the attestation structure certifyInfo and the signature on that structure under the private part of the AK. Note that the attestation structure is constructed with the public parts of the two keys, instead of their names as it is described in the TPM2_Certify command. In addition, the signature on the attestation structure is modelled via the Tamarin built-in revealSign/2 function [4]. The signature can be then verified using the equational theory:

$$\texttt{revealVerify}(\texttt{revealSign}(\texttt{certifyInfo}, \texttt{ak}_{\texttt{ssk}}), \texttt{certifyInfo}, \texttt{ak}_{\texttt{spk}}) = \texttt{true}.$$

Tamarin also contains the function symbol getMessage/1, which given the signature that is created as $\texttt{revealSign}(\texttt{certifyInfo}, \texttt{ak}_{\texttt{ssk}})$ it outputs the message that is signed, via:

$$\texttt{getMessage}(\texttt{revealSign}(\texttt{certifyInfo}, \texttt{ak}_{\texttt{ssk}})) = \texttt{certifyInfo}.$$

Ideal functionality for creating a credential: $\mathcal{F}_{\texttt{TPM2\_MakeCredential}}$

We model the TPM command TPM2_MakeCredential, for protecting a credential cred. In order to do so, we need to note that the storage key that will be used the EK with handle $\texttt{ek}_{\texttt{h}}$, represented as the key pair $(\texttt{ek}_{\texttt{sk}}, \texttt{ek}_{\texttt{pk}} = \texttt{pk}(\texttt{ek}_{\texttt{pk}}))$. On the other hand, the object to which the credential cred is bound is the AK with handle $\texttt{ak}_{\texttt{h}}$, represented as the pair $(\texttt{ak}_{\texttt{ssk}}, \texttt{ak}_{\texttt{spk}} = \texttt{pk}(\texttt{ak}_{\texttt{spk}}))$.

The binding of cred with the AK is done with the function symbol makeCredential/3, in line 5 of Figure 3.2. This function outputs the value credBlob which will be used in order to activate the credential cred. We also define the function symbols activateCredential/3 and

$\mathcal{F}_{\texttt{TPM2\_MakeCredential}} :=$

```
1: in(<'MakeCredential', ek_h, cred, ak_h>);   //input: handles of EK and AK & cred to
   be protected and bound to AK
2: lock ek_h; lock ak_h;
3: lookup <publicPart, ek_h> as ek_pk in                    //get EK's public part
4: lookup <publicPart, ak_h> as ak_spk in                   //get AK's public part
5: let credBlob = makeCredential(ek_pk, cred, ak_spk) in
6: out(credBlob);
7: unlock ek_h; unlock ak_h
```

Figure 3.2: Ideal functionality $\mathcal{F}_{\texttt{TPM2\_MakeCredential}}()$

verifyCredential/3, which satisfy the equational theory:

$$\texttt{activateCredential}(\texttt{ak}_{\texttt{spk}}, \texttt{ek}_{\texttt{sk}}, \texttt{makeCredential}(\texttt{ek}_{\texttt{pk}}, \texttt{cred}, \texttt{ak}_{\texttt{spk}})) = \texttt{cred}, \tag{3.1}$$

$$\texttt{verifyCredential}(\texttt{ak}_{\texttt{spk}}, \texttt{ek}_{\texttt{sk}}, \texttt{makeCredential}(\texttt{ek}_{\texttt{pk}}, \texttt{cred}, \texttt{ak}_{\texttt{spk}})) = \texttt{true}. \tag{3.2}$$

In this way we avoid the modelling of the cryptographic operations that are needed in the actual TPM2_MakeCredential command, such as the key derivation functions, asymmetric encryption and HMAC computations. In our model that will be presented in Chapter 4, the EK will be considered as an object that is already loaded into the TPM.

We note that this command does not use any TPM secrets nor does it require authorization. It is a convenience function, using the TPM to perform cryptographic calculations that could be done externally [24]. Indeed, from the point of view of modelling, the party executing this command can simply create a term by invoking the verifyCredential/3 function symbol, and the equational theory defined above will follow. Also, the output of this ideal functionality is just the single object credBlob, rather than the pair of objects credentialBlob, secret returned by a TPM (see Figure 2.2), since the semantics of the theory is not affected.

$\mathcal{F}_{\texttt{TPM2\_ActivateCredential}} :=$

```
1: in(<'ActivateCredential', ek_h, credBlob, ak_h, ak_sh>);   //input: handle of EK,
   credBlob, handle of AK & AK session handle
2: lock ek_h; lock ak_h; lock ak_sh
3: lookup <'policyDigest', ak_sh> as pD in
4: lookup <'authPolicy', ak_h> as aP in
5: if aP == pD then
6:   lookup <'publicPart', ak_h> as ak_spk in
7:   lookup <'privatePart', ek_h> as ek_sk in
8:   if verifyCredential(ak_spk, ek_sk, credBlob) = true then
9:     let cred = activateCredential(ak_spk, ek_sk, credBlob) in
10:    out(cred);
11:  else unlock ek_h; unlock ak_h; unlock ak_sh
12: else unlock ek_h; unlock ak_h; unlock ak_sh
```

Figure 3.3: Ideal functionality $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}()$

Ideal functionality for activating a credential: $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}$

For the ideal functionality $\mathcal{F}_{\texttt{TPM2\_ActivateCredential}}$, no authorization is needed for the EK, since we assume that it is globally defined and already loaded into the TPM. The process of activating a credential that was previously generated by the ideal functionality $\mathcal{F}_{\texttt{TPM2\_MakeCredential}}$ is described in Figure 3.3. Note that the verification of the `credBlob` in step 8 is done using Equation (3.2) and the activation of `cred` in step 9, using Equation (3.1).

Ideal functionality for signing/verifying: $\mathcal{F}_{\texttt{TPM2\_Sign}}/\mathcal{F}_{\texttt{TPM2\_VerifySignature}}$

Our model for the signing command is presented in Figure 3.4. For the creation of the signature `sign` under the secret signing key $\texttt{k}_{\texttt{ssk}}$, referenced in the handle $\texttt{k}_{\texttt{h}}$ we use the built-in `revealSign/2` function of Tamarin [4]. The verification command is summarized in Figure 3.5.

$\mathcal{F}_{\texttt{TPM2\_Sign}} :=$

```
1: in(<'Sign', k_h, k_sh, m>);          //input:  key & policy session handles & message
2: lock k_h; lock k_sh;
3: lookup <'policyDigest', k_sh> as pD in      //get policy digest from policy session
4: lookup <'authPolicy', k_h> as aP in                 //get authPolicy value from key
5: if aP == pD then                          //compare authPolicy against policy digest
6:   lookup <'privatePart', k_h> as k_ssk in                  //get secret signing key
7:   let sign = revealSign(m, k_ssk) in
8:   out(sign);
9:   unlock k_h; unlock k_sh
10: else unlock k_h; unlock k_sh
```

Figure 3.4: Ideal functionality $\mathcal{F}_{\texttt{TPM2\_Sign}}()$

$\mathcal{F}_{\texttt{TPM2\_VerifySignature}} :=$

```
1: in(<'VerifySign', k_h, m, sign>);        //input:  key handle & message & signature
2: lock kH;
3: lookup <'publicPart', k_h> as k_spk in                    //get public signing key
4: let validation = revealVerify(sign, m, k_spk) in
5: out(validation);
6: unlock k_h
```

Figure 3.5: Ideal functionality $\mathcal{F}_{\texttt{TPM2\_VerifySignature}}()$

Ideal functionality for public key encryption: $\mathcal{F}_{\texttt{TPM2\_PK\_Encrypt}}$

The model of the ideal functionality $\mathcal{F}_{\texttt{TPM2\_PK\_Encrypt}}$ is summarized in Figure 3.6. For the decryption process, authorization is needed for using the private part of the key referenced in the handle $\texttt{k}_{\texttt{h}}$. As usual, this is done using the authorization policy of the key and the policy digest of the PCR policy session. Note here that for the creation of the ciphertext in 3 of Figure 3.6 we use the function symbol `aenc/2`, while for the decryption process in step 6 of Figure 3.7, we use the function symbol `adec/2` [4]. These are linked by the relation:

$$\texttt{adec}(\texttt{aenc}(\texttt{m}, \texttt{k}_{\texttt{pk}}), \texttt{k}_{\texttt{sk}}) = \texttt{m}.$$

$\mathcal{F}_{\texttt{TPM2\_PK\_Encrypt}} :=$

```
1: in(<'PK-Encrypt', k_h, message>);          //input:  handle of public key, message
2: lock k_h;
3: lookup <'publicPart', k_h> as k_pk in
4: let outData = aenc(message, k_pk) in
5: out(outData);
6: unlock k_h
```

Figure 3.6: Ideal functionality $\mathcal{F}_{\texttt{TPM2\_PK\_Encrypt}}()$

$\mathcal{F}_{\texttt{TPM2\_PK\_Decrypt}} :=$

```
1: in(<'PK-Decrypt', k_h, k_sh, ciphertext>);        //input:  handle of public key and
   session & ciphertext
2: lock k_h; lock k_sh;
3: lookup <'policyDigest', k_sh> as pD in     //get policy digest from policy session
4: lookup <'authPolicy', k_h> as aP in                //get authPolicy value from key
5: if aP == pD then                          //compare authPolicy against policy digest
6:   lookup <'privatePart', k_h> as k_sk in
7:   let message = adec(ciphertext, k_sk) in
8:   out(message);
9:   unlock k_h; unlock k_sh
10: else unlock k_h; unlock k_sh
```

Figure 3.7: Ideal functionality $\mathcal{F}_{\texttt{TPM2\_PK\_Decrypt}}()$

3.2 Updates of Ideal TPM Functionalities in D3.3

The modelling of the additional TPM commands in the previous section results in the need to perform certain updates on the ideal functionalities that we have modelled in the D3.3 [12]. In

$\mathcal{F}_{\texttt{TPM2\_Create}} :=$

```
1: in(<'Create', k_sh>);                                  //input:  handle of session
2: lock k_sh;
3: lookup <'policyDigest', k_sh> as pD in     //get policy digest from policy session
4: new k_h;                                                 //handle of the new key
5: lock k_h;
6: new k_sk;                                              //private part of new key
7: let k_pk = pk(k_sk) in;                                 //public part of new key
8: insert <'privatePart', k_h>, k_sk;
9: insert <'publicPart', k_h>, k_pk;
10: insert <'authPolicy', k_h>, pD;
11: out(<k_h, k_pk>);
12: unlock k_h; unlock k_sh
```

Figure 3.8: Revised ideal functionality $\mathcal{F}_{\texttt{TPM2\_Create}}()$

particular, we need to revise the modelling of the $\mathcal{F}_{\texttt{TPM2\_Create}}$ command in order to consider

the creation of an asymmetric key pair with a public and private part. The updated version of $\mathcal{F}_{\texttt{TPM2\_Create}}$ is described in Figure 3.8.

In addition, in the ideal functionality $\mathcal{F}_{\texttt{TPM2\_StartAuthSession}}$ of D3.3, for creating a session, we distinguished the type of the session to be created, as "trial" and "policy" session. The type of the session was stored in the field "SESSIONType" (see Figure 2.7 in [12]). Trial sessions are used for creating keys and specifically for linking the authorization policy (authPolicy) of the key to specific policy (i.e. to the policyDigest of the session). Policy sessions are used in order to load the key in the TPM. The difference in the two session types is that in the case of a trial session, the TPM simply updates the policy digest of the session according to the user's input digest value. In the case of policy sessions, e.g. using PCR policy, in order to update the policy digest value of the session, the user's input digest is checked against the corresponding PCR values and the update will be performed if the two values match.

$\mathcal{F}_{\texttt{TPM2\_StartAuthSession}} :=$

```
1: in(<'StartAuthSession'>);
2: new s_h;
3: lock s_h;
4: insert <'policyDigest', s_h>, null;          //policy digest is set to null
5: out(s_h);
6: unlock s_h
```

Figure 3.9: Revised ideal functionality $\mathcal{F}_{\texttt{TPM2\_StartAuthSession}}()$

We update the ideal functionality $\mathcal{F}_{\texttt{TPM2\_StartAuthSession}}$, is such way that when we create a session, we do not consider the type of the session, motivated also by the work of Shao et al. [21]. That is, In our model we will consider all sessions as policy sessions. The evaluation of the policy digest based on PCR values is something that we will be dealing with in the final Deliverable D3.5. The revised ideal functionality for $\mathcal{F}_{\texttt{TPM2\_StartAuthSession}}$ is presented in Figure 3.9.

Chapter 4

Security Modelling of Use Case #3: Create and Certify the AK and TLS keys

The device management use case is described in full depth in Deliverables D4.1 [10], D6.1 [9] and D6.3 [11]. The device management demonstrator consists of three main entities, the Routers which are responsible for routing user traffic, the Network Management System (NMS) which manages the Routers through TLS channels and the RA Server which is responsible for attesting the Routers.

Each new Router that joins the network needs to securely establish trust with NMS. This process is called *enrollment* and it is achieved with the issuance of a TLS certificate that is used to securely communicate with the NMS or with other Routers. Therefore, the enrollment process involves the creation of a TLS by the Router, which is certified by the RA Server and signed by the NMS. The certification of the TLS key is achieved using the AK, which is previously generated by the Router and certified by the RA Server. The certification of the AK is accomplished by the IBM remote, privacy-preserving attestation protocol which we described in Section 2.2 (see Figure 2.8). Once the AK is certified, the Router will generate a TLS signing key pair, which can then be used for the establishment of TLS channel.

Each Router is equipped with a (quantum-resistant) TPM, which is used for certain cryptographic operations, such as for creating cryptographic keys, quotes and Certificate Signing Requests (CSRs). In addition, each Router consists of two components. The Zero Touch Provisioning (ZTP) Agent, which is an agent running on each Router and it is responsible for initiating certification of the AK and the TLS key. In addition, the ZTP Agent is responsible for responding to remote attestation request by the NMS. Each Router also contains the RA Client, which 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 enrollment of each Router and for the verification of the CSRs and quotes that are received from the Router.

In this chapter we present our security model for the creation of the AK and TLS keys. The certification process of the two keys is split in two parts. In this deliverable we present our modelling approach for the certification of the AK as this is described by Huawei in Deliverable 6.3 [11, Chapter 4, HWDU.NO.1], i.e. the model for the IBM remote attestation protocol. The modelling for the certification of the TLS key is currently in progress and it will be presented in detail in Deliverable 3.5. We point out here that both the AK and TLS keys in this particular instance are asymmetric keys, hence they have a public and a private part and they are both used for signing.

For the convenience of the reader, we give a brief description of the AK certification process. This is described in Figure 4.1, which is taken from Deliverable D6.3. In particular, the AK certification

is composed of the following steps:

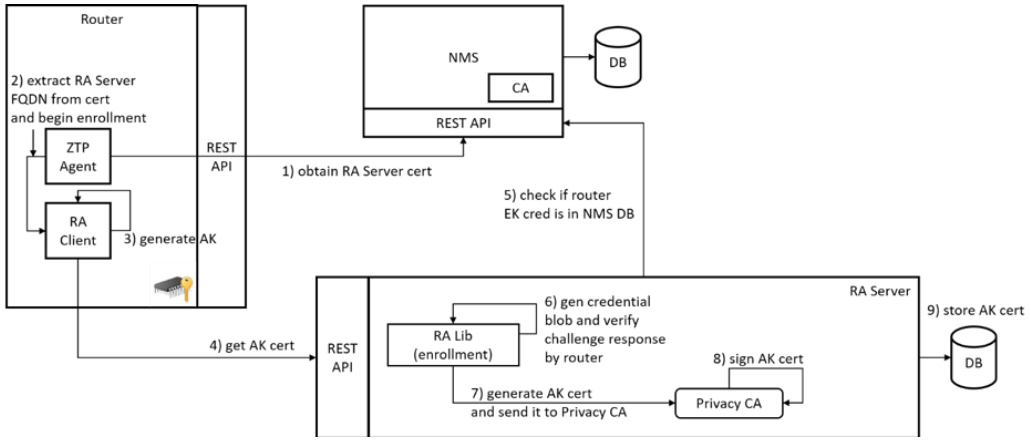

Figure 4.1: Creation of AK by the Router and certification by the RA Server [11]

1. Obtain RA Server cert: The ZTP Agent in the Router obtains the RA Server certificate from the NMS.
2. Extract RA Server FQDN from certificate and begin the enrollment: The ZTP Agent extracts the RA Server FQDN from the certificate and passes it to RA Client.
3. Generate AK: The RA Client generates an AK that will be used to certify the TLS key. In particular, the RA Client executes the TPM commands: `TPM2_StartAuthSession` and `TPM2_Create` in sequence. The first command will create a new session and initiate the policy digest value of this session to zero digest. According to the description of the device management use case, the AK will not be bound to any policy and hence the command `TPM2_Create` will create the AK with `authPolicy` set to zero digest.
4. Get AK cert: The RA Client asks the RA Server to issue a certificate for the AK it generated.
5. Check if Router EK credential is in NMS DB: The RA Server asks the NMS if the EK credential of the Router requesting an AK certificate has been added to the NMS DB by the Network Administrator; this prevents any Router from getting an AK certificate.
6. Generate credential blob and verify challenge response by Router: The RA Lib generates a credential blob and asks RA Client in the Router to prove that the Router possesses the EK. This is achieved through the emulated steps of the command `TPM2_MakeCredential`, which correspond to steps 1–11 in the IBM remote attestation protocol (Figure 2.8). The proof on behalf of the RA Agent created with the command `TPM2_ActivateCredential` (step 12 in Figure 2.8).
7. Generate AK certificate and send it to Privacy CA: The RA Lib generates a certificate for the Router AK and asks Privacy CA in RA Server to sign the certificate.
8. Sign AK certificate: Privacy CA signs the AK certificate; RA Server sends it to the Router. This is done in step 14 in Figure 2.8.
9. Store AK certificate: RA Server stores the signed AK certificate in the DB.

The security modelling of the above user story is presented in Section 4.4. The code in SAPiC is also given in the Appendix A.

4.1 Overview of the Modelling Tools Used

Before delving into the details of the modelling of the scenarios, we briefly present an overview of the tools used. We perform the modelling and analysis of security properties of the use cases using the Tamarin prover and its front-end SAPiC. We refer the reader to [19, 16, 4, 15] for further reference.

Tamarin is a state-of-the-art tool for symbolic verification and automated analysis of security properties in protocols, under the Dolev-Yao model [13], with respect to an unbounded number of sessions. We already discussed similar tools for symbolic verification in Deliverable D3.1 [7], most notably ProVerif [5], where protocols are specified using applied pi-calculus [1]. As opposed to Tamarin, which implements a constraint solving algorithm, ProVerif is based on a resolution algorithm that works at the Horn-clause translation of the protocol under analysis. This translation works reasonably well for protocols with monotonic global state (i.e., protocols that do not "forget" information), but it makes difficult to model protocols with arbitrarily mutable global state, as it occurs for the requirements of the use cases. Several extensions to ProVerif have appeared to tackle this issue [3, 6], but they also come with some limitations (e.g., a finite number of memory cells). In our approach, we have decided to implement our models with Tamarin, since it can handle protocols with unbounded, non-monotonic global state, even under replication, and unbounded sessions. However, we are aware that it comes with the trade-off of loosing some automation, i.e., the user has to provide auxiliary lemmas for complex protocols. In addition, we note that there are related works that have been successful in proving security properties for TPM functionalities using these tools, such as [21, 20].

More concretely, we develop our models using the SAPiC front-end, which enables to define protocols in a calculus (similar to applied pi-calculus) rather than directly into multiset rewrite rules, and converts them into *(labeled) multiset rewriting rules* (MSRs) to be analysed by Tamarin.

Fig. 4.2 describes the SAPiC syntax. The calculus comprises an *order-sorted term algebra* with 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.

These rules, when fired, model the state transitions and generate a sequence of labels called *trace*. Every transition is labeled by facts.

The set of facts is defined as $\mathcal{F} = \{F(t_1, \ldots, t_n) \mid t_i \in \mathcal{T}, F \in \Sigma \text{ of arity } n\}$. 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, and A is a set of action facts, modeled by SAPiC events. For a process P, its trace $\mathrm{Tr}(P) = [F_1, \ldots, F_n]$ 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, modeled as trace properties. The construct $F@i$ states that fact F occurs (i.e., is true) at timepoint i. A property can be specified as a *lemma* or as a *restriction*, depending if the property is being verified or enforced [19].

Except for observational equivalence, Tamarin is sound and complete, but it may not terminate, because the verification problem is undecidable in general. In that case, user intervention is required in the form of auxiliary lemmas. Everything SAPiC does can be expressed in MSRs in Tamarin. However, compared to direct MSR encoding, modelling using SAPiC helps to develop

$\langle P,Q \rangle$ ::=	processes
$\mid 0$	terminal (null) process
$\mid P \mid Q$	parallel composition of P and Q
$\mid \texttt{!}P$	replication of P
$\mid \nu\ a;\ P$	binds a to a new fresh value in P
$\mid \texttt{out}(m,t);\ P$	outputs message t to channel
$\mid \texttt{in}(m,t);\ P$	inputs message t from channel m
$\mid \texttt{if}\ Pred\ \texttt{then}\ P\ [\texttt{else}\ Q]$	P if predicate $Pred$ holds; otherwise Q
$\mid \texttt{event}\ F;\ P$	executes event (action fact) F
$\mid P\ \texttt{+}\ Q$	non-deterministic choice
$\mid \texttt{insert}\ m,\ t;\ P$	inserts t at memory cell m
$\mid \texttt{delete}\ m;\ P$	deletes the content m
$\mid \texttt{lookup}\ m\ \texttt{as}\ x\ \texttt{in}\ P\ [\texttt{else}\ Q]$	if m exists, bind it to x in P; otw. Q
$\mid \texttt{lock}\ m;\ P$	gain exclusive access to cell m
$\mid \texttt{unlock}\ m;\ P$	waive exclusive access to m
$\mid [L] -[A]\rightarrow [R];\ P \quad (L,R,A \in \mathcal{F}^*)$	provides access to Tamarin MSRs

Figure 4.2: SAPiC syntax ($a \in FN$, $x \in \mathcal{V}$, $m,t \in \mathcal{T}$, $F \in \mathcal{F}$). Note that, as opposed to the applied pi-calculus [2], SAPiC's input construct $\texttt{in}(m,t);P$ performs pattern matching instead of variable binding.

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 that case SAPiC has a better chance for termination. In general, it is more convenient, although some things, like state machines, are more natural in MSRs, but these are out of the scope of our models.

4.2 Modelling Approach & Challenges

As stated in the introduction, our goal is to show how the ideal functionalities of the TPM that we have described in Section 3.1 can be used to demonstrate security properties in the device management reference scenario. In particular we are interested in modelling the core TPM functionalities which appear in most "real life" TPM-based applications in the literature and which are related to the management of PCRs, the creation of objects and their sealing to PCRs and the remote attestation mechanisms, either DAA or the IBM remote attestation protocol.

To this end, the properties that are within the scope of the model for the Network Management use case are:

- The legitimate certification of AKs,
- The legitimate certification of TLS Keys and
- The successful establishment of a TLS connection between Router and NMS.

Properties that we consider as out of scope in our model are the certification of the EKs, as well as any other certification of keys that are assumed to be trusted. In addition, some components will be aggregated into the same process. Observation of communications between these components will be outside the scope of the adversary. Particularly, we distinguish the following four processes in our model for the device management use case:

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

Figure 4.3 shows an overview of the processes and interactions that comprise our model. Our adversarial model will be mainly a Dolev-Yao model, which allows an adversary to monitor and modify all interactions between the processes. However, the communication between the TPM and the Router has to be considered independently. This will be discussed in Sec. 4.2 below. Our

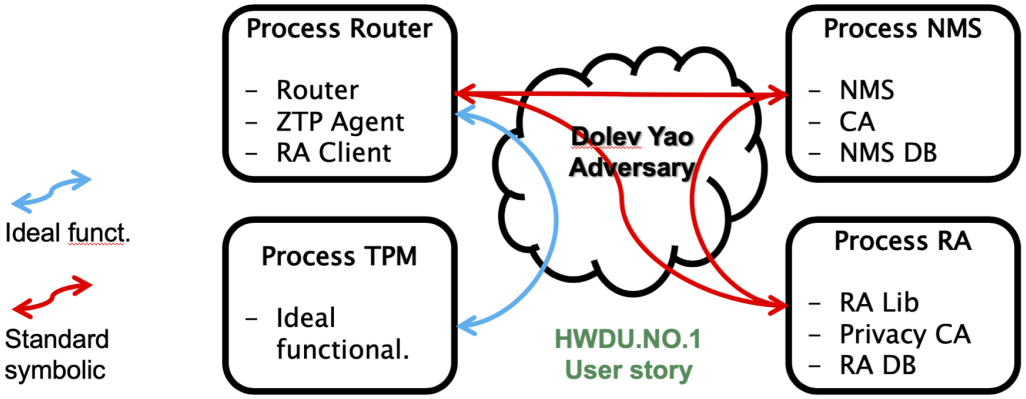

Figure 4.3: Device management use case model overview

model is presented in detail in Appendix A.

4.2.1 Modelling Challenges

Channel modes

One of the main aspects to define when modelling this use case involves how the communication between the different processes takes place, in order to offer a view as close to the reality as possible. Of course, this depends on the threat model, and on what components we assume that are compromised in the scenario.

Ideally, we would like to test for security properties under the presence of the strongest adversary that we can consider for the model, i.e., a standard Dolev-Yao adversary that interacts with the four processes depicted in Fig. 4.3. It is easy to see that this adversary model is overly powerful for practical purposes: the adversary can intercept, drop, replay, etc. any message between the processes. In particular it can drop or send arbitrary commands to the TPM at any time point.

Even with the command abstractions and idealization of TPM-related cryptograpy, the adversary can simply drop all communication between the host and the TPM, reset the TPM, and extend the PCRs with a correct software configuration. At this point, the adversary will be able to forge any protocol execution that relies on the TPM as a root of trust (e.g., the adversary can certify any AK without having to interact with the router software). This would translate, in the real world, as the capability of an adversary to detach, communicate, and reattach a physically soldered TPM in the router at any time.

The TPM Specification Part 1 [22] allows several implementations of the device (discret, integrated, software,...) as long as it is possible to guarantee that it has a separate state from the system which it reports and that the only interaction between the host system is through the interface defined in the specification. For the device management use case, and since the TPM must act as the Root of Trust for Reporting, we assume that the TPM is implemented as a single-chip component attached to the system using a low-performance interface, e.g., the low pin count (LPC) interface, embedded into its motherboard. Several authors [**?**, **?**] model this by assuming that there is a perfectly secure channel between the host device and the TPM (i.e., the adversary cannot read or send messages on it), whereas they impose no constraints between the communication channels between the remaining processes. This is a strong restriction on the capabilities of the Dolev-Yao adversary, but it is indeed appropriate for the use case, since it reflects the security assumptions where the TPM is expected to act as a root of trust. Furthermore, treating this communication separately from the other channels allows for some security assumptions to be relaxed individually in the future. For example, considering scenarios where the router software or the TPM are compromised by the adversary.

Due to the undecidability nature of the protocol verification problem, it is difficult to determine an alternative to implement the channel modes without conducting tests in the model, as some of them might lead to non-termination when querying security properties. In order to assess the feasibility of them, we initially model the LPC bus as a standard public channel, and execute queries for reachability of all the process branches. When provided with the SAPiC translation in this case, Tamarin will usually terminate without much user intervention. Once we are confident that the model is able to reach all branches, there are several alternatives that can be considered to implement the private communication channel. The most straightforward is using the built-in SAPiC construct for private channels as follows:

```
// Create a fresh private channel to model the LPC bus
ν ~lpcBus;

// Sending a command through the private channel (Router process)
out(~lpcBus, ⟨TPM_CommandCode, param_1, param_2, ... param_k⟩); P

// Receiving a command through the private channel (TPM process)
in(~lpcBus, ⟨TPM_CommandCode, param_1, param_2, ... param_k⟩); Q
```

One might be tempted to think that if the verification of a model terminates when all channels are public, it will terminate more easily when some of them are private: intuitively, the adversary has access to less terms to manipulate, and it has also less channels to send information. Unfortunately, this is not the case, and using private channels might slow the verification and even cause non-termination. The reason for that is that private channels are synchronous in SAPiC calculus (as it is usual in traditional process calculus), in contrast to public channels. That is, $\texttt{in}(m, x);\ P \mid \texttt{out}(m, t);\ Q$ reduces to $P[t/x] \mid Q$ in one step. This makes the translation more complicated. It also introduces problems with defining when exactly channels become asynchronous, i.e., when the private information is leaked/deduced.

The second alternative is to make use of Tamarin restrictions to limit the usage that the adversary can make of the TPM. This requires using a template for sending/receiving messages to/from the TPM with events placed at the appropriate locations:

// Template for sending a TPM command (Router process)
let pat_tpm_send_command = $\langle$TPM_CommandCode, $param_1, param_2, \ldots param_k\rangle$ 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_receive_command = $\langle$TPM_CommandCode, $param_1, param_2, \ldots param_k\rangle$ in
in(pat_tpm_receive_command);
event TPM_SendCommand(pat_tpm_receive_command);
Q

This requires the following restriction that will limit the adversary capabilities:

$$\begin{aligned}&\forall c, t_1.\texttt{TPM\_ReceiveCommand}(c)@t_1 \Rightarrow \\ &\quad (\exists t_2.\texttt{TPM\_SendCommand}(c)@t_1 \wedge (t_2 < t_1)) \wedge \\ &\quad \neg(\exists t_3.\texttt{TPM\_ReceiveCommand}(c)@t_3 \wedge \neg(t_3 = t_1)))\end{aligned}$$

Namely, the restriction will forbid the adversary from calling the TPM arbitrarily, unless the honest process has first request a TPM call. The main drawback of this approach is that it gives the adversary the capability to replay messages to the TPM, which might not represent a realistic scenario.

The next alternative to model the communication between the TPM and the Router processes consists in using the asynchronous state management capabilities from the calculus. That is, using `insert/lookup` as a means of communication:

// Sending a command through global state (Router process)
insert TPM2_CommandCode, $\langle param_1, param_2, \ldots param_k\rangle$; P

// Receiving a command through global state (TPM process)
lookup TPM2_CommandCode as $\langle param_1, param_2, \ldots param_k\rangle)$ in Q

Finally, the last alternative to model this channel would be using advanced features from SAPiC, as it is constructiong an asynchronous secret channel by using embedded MSRs, that is

// Sending a command through embedded MSR (Router process)
$[\ ] -[\ ]\rightarrow [!\texttt{Chan}(P_{\texttt{Router}}, P_{\texttt{TPM}}, <\texttt{TPM\_CommandCode}, param_1, param_2, \ldots param_k >)]; P$

// Receiving a command through embedded MSR (TPM process)
$[!\texttt{Chan}(P_{\texttt{Router}}, P_{\texttt{TPM}}, <\texttt{TPM\_CommandCode}, param_1, param_2, \ldots param_k >)] -[\ ]\rightarrow [\ ]; Q$

Note that embedded MSRs is an advanced part of SAPiC, and they are connected with ";" just like any other action, and variables bound in them are available to the rest of the process. Above, Q can use the variables $param_i$. Using the persistent state fact "!Chan" allows for replays, which usually speeds up verification. Alternatively, it can be considered whether to avoid this by dropping the "!".

Open chains

Even though we are not explicitly modelling secure communications between the different components (e.g., TLS connections), the protocols in the device management use case make usage of cryptography to protect some secrets exchanged, as expected. For example, the primitive `TPM2_MakeCredential` can be regarded as an AE, as commented above. The adversary, therefore, cannot access encrypted information unless some key is leaked. However, if the adversary forwards an encrypted, unknown secret to a process, then the process might decrypt (or, in general, execute an operation unavailable to the adversary) and output the result. That is, if Tamarin identifies that the process outputs a message later, it might believe that it can use that honest processes as an oracle for arbitrary encrypted terms using a key that the tool tries to derive unsuccessfully. Technically speaking, this causes that the tool has *partial deconstructions* (open chains) left, complicating the proof of a security property by either taking a very long time, or not terminating. 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. Therefore, at this point, even if a process refuses to output a malformed AE, it is not guaranteed the absence of partial deconstructions. We refer the reader to [18] for more details.

A *sources lemma* is one of the possibilities to guide the proof and help Tamarin terminate. For the particular case of the output of `TPM2_MakeCredential`, that is, the `credentialBlob` we have found that Tamarin is unable to determine that it cannot use the honest process as oracles to retrieve useful information. For this particular case, we need to define the events `Source` and `Receive` as follows:

```
// RA server process (excerpt) creates a ν challenge and protects it
// with TPM2_MakeCredential
ν ~challenge;
event Source(~challenge);
let credentialBlob = makeCredential(ek_pk, ~challenge, ak_spk) in
out(⟨'RA_enrollrequest_resp', credentialBlob⟩);


// TPM process (excerpt) receives the output of TPM2_MakeCredential and
// decrypts the challenge
let pat_tpm_command = ⟨'TPM2_ActivateCredential', a_h, a_sh, k_h, credentialBlob⟩ in
in(pat_tpm_command);
event TPM_ReceiveCommand(pat_tpm_command);

let challenge = activateCredential(a_pk, k_sk, credentialBlob) in
event Receive(challenge);
out(challenge);
```

These events allow us to define the following sources lemma that eliminates the partial deconstructions left for this case:

$$\forall s, t_1.\ \texttt{Receive}(s)@t_1 \Rightarrow \\ (\exists t_2.\ K(s)@t_2 \wedge (t_2 < t_1)) \vee (\exists t_2.\ \texttt{Source}(s)@t_2).$$

Indeed, the lemma states the obvious fact that whenever an honest process is able to retrieve a secret s, then s was either known beforehand by the adversary, or it was freshly created by another honest process. Since honest processes in our model do not leak keys, the lemma helps Tamarin 'realising' that the outputs of the protocols can not be used by the adversary to gain any additional information.

4.3 Mapping Create/Load Model to Use Case #3

In D3.3 [12, Chapter 2] we have presented the model for creating a TPM key and loading it into the TPM, using the EA feature of TPM2.0. In this section we will apply this model for the AK and TLS key creation as it is described in the device management use case [11].

Each TLS key is generated by the RA Client, running in the Router, and it is used in order to establish a trusted communication channel between the Router and the NMS. The creation of the TLS key requires its authorization policy `authPolicy` to be asserted with a PCR policy, based on PCR extensions that correspond to the current system state. In order for the TLS key to be used, an authorization check is required, in which the `authPolicy` is checked against the policy digest of the corresponding policy session and if the two values match, the key can be used. On the other hand, the AK is created with an empty authorization policy, in other words its `authPolicy` is set to the zero digest. That is, the AK key is not bound to specific PCR values and hence can be loaded and used without any authorization check. The validation of the AK key is performed through its AK certificate.

Our model for the creation of the AK key is described in Figure 4.4. As pointed out earlier, the only difference from the TLS creation process is that the AK key is not linked to a PCR policy and hence the ideal functionality $\mathcal{F}_{\texttt{TPM2\_PolicyPCR}}$ is not required. This means that the authorization policy of the AK key will be set to `null`. In addition, the AK is an asymmetric signing key and

Figure 4.4: Create an AK key in TPM and use it for signing

hence it has both private and public parts $(\texttt{ak}_{\texttt{ssk}}, \texttt{ak}_{\texttt{spk}})$.

Unlike the EA model that we have presented in D3.3 [12], the authorization check for the TLS key is not performed in the `TPM2_Load` command, but in the command that orders the TPM to use it for some cryptographic purpose during the secure communication between the Router and the NMS. In terms of security modelling, the process of generating the TLS and AK keys is similar to the create/load model that we have described in D3.3 [12]. This is made clear when the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Load}}$ is replaced by the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Sign}}$, which is described in Fig. 3.4

The command flow for creating a TLS key is described in Fig. 4.5. Note that this model is aligned with the update in the ideal functionality $\mathcal{F}_{\texttt{TPM2\_Create}}$, in which the key that will be created is asymmetric and its public part is returned along with the corresponding handle `kH`. The TLS key in the context of the device management use case is a signing key with private part $\texttt{tls}_{\texttt{ssk}}$ and public part $\texttt{tls}_{\texttt{spk}}$. Step 8 in Figure 4.5 demonstrates the authorization check, i.e. the authorization policy `aP` of the key is checked against the policy digest value `pD` of the session

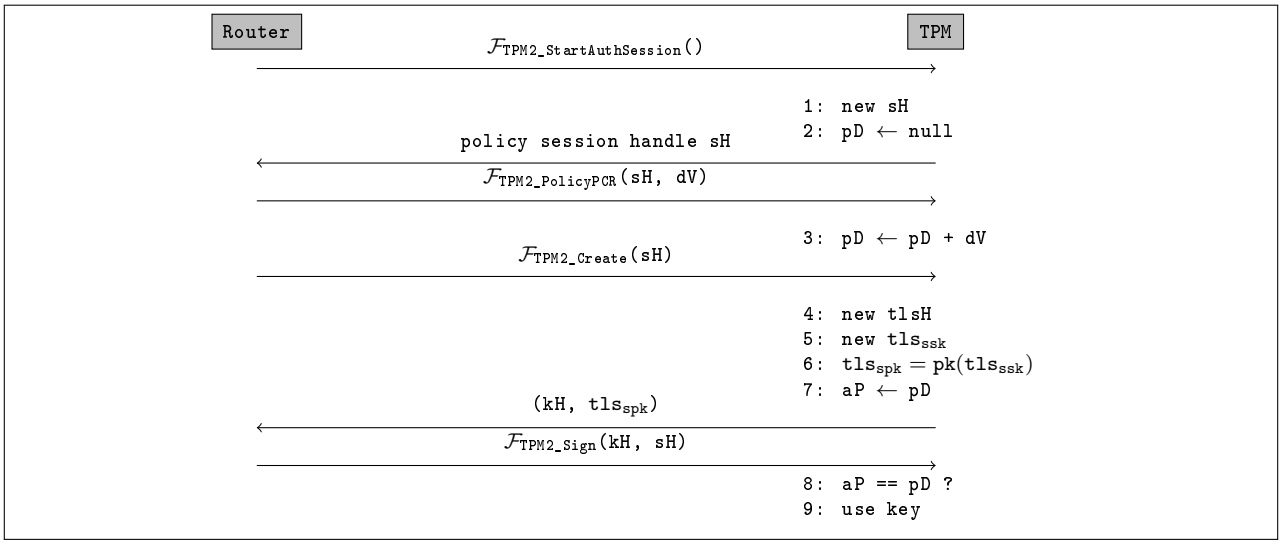

Figure 4.5: Create a TLS key in TPM and use it for signing

with handle sH. For the description of the ideal functionality $\mathcal{F}_{\text{TPM2\_PolicyPCR}}$, we refer to Deliverable D3.3 [12, Figure 2.8, p. 14].

4.4 Modelling the IBM Attestation Protocol

Our model for the AK certification that is described in Figure 4.1, in SAPiC is presented in Figure 4.4. In our modelling approach we consider three main processes, the TPM, the Router and the RA Server. The Router is the entity that interacts with the TPM. This interaction and exchange of messages between the two entities is modelled using private channels. That is, an adversary is not able to monitor the communication between the Router and the TPM. The interaction between the Router and the RA Server is modelled using public channels and hence the adversary has full access to this communication. The IBM attestation protocol requires the use of the EK, which is is an asymmetric encryption key, for certifying the AK. In order to keep our model as simple as possible, we do not model key hierarchies in the TPM, in other words, we consider the EK as a fixed key that is already created and its public part is loaded in the TPM. Hence we set the following values in the TPM process:

$$\begin{aligned}
&\texttt{insert} \;\; < \text{`authPolicy'}, \sim \text{ek}_\text{h} >, \;\; \texttt{null} \\
&\texttt{insert} \;\; < \text{`privatePart'}, \sim \text{ek}_\text{h} >, \;\; \sim \text{ek}_\text{sk} \\
&\texttt{insert} \;\; < \text{`publicPart'}, \sim \text{ek}_\text{h} >, \;\; \sim \text{ek}_\text{pk}
\end{aligned}$$

where ek_h is the handle of the EK and its authorization policy is initialized as null. Additionally, the following variables are also assumed to be fixed throughout the model.

$\sim \text{fqdn}_{\text{router}}$:	the FQDN of the router
$\sim \text{fqdn}_{\text{nms}}$:	the FQDN of the NMS
$(\sim \text{ssk}_{\text{ra}}, \text{spk}_{\text{ra}} = \text{pk}(\sim \text{ssk}_{\text{ra}}))$:	private/public part of RA Server signing key
$(\sim \text{ssk}_{\text{nms}}, \text{spk}_{\text{nms}} = \text{pk}(\sim \text{ssk}_{\text{nms}}))$:	private/public part of NMS signing key

We give a more detailed description of the steps and exchanged messages of Figure 4.4.

1. The Router receives the NMS certificate. This is constructed by the following equation:

$$\texttt{let} \;\; \text{pat}_{\text{cert}-\text{nms}} = \langle \langle \text{spk}_{\text{ra}}, \text{fqdn}_{\text{nms}} \rangle, \text{signature}_{\text{cert}-\text{nms}} \rangle \;\; \texttt{in},$$

where $\mathtt{signature}_{\mathtt{cert-nms}}$ is the NMS signature on the message $\langle \mathtt{spk}_{\mathtt{ra}}, \mathtt{fqdn}_{\mathtt{nms}} \rangle$ with its private signing key $\mathtt{ssk}_{\mathtt{nms}}$. The Router verifies the certificate of the NMS, using the public part $\mathtt{psk}_{\mathtt{nms}}$ and extracts the NMS FQDN. This is accomplished by the `verify` function.

2. The Router uses the TPM to create a new session for the AK. This is done using the ideal functionality $\mathcal{F}_{\mathtt{TPM2\_StartAuthSession}}$. The TPM creates a fresh handle $\mathtt{ak}_{\mathtt{sh}}$ for this session and initializes its policy digest value to the zero digest. It returns the session handle $\mathtt{ak}_{\mathtt{sh}}$ to the Router.

3. The Router asks the TPM to create the AK, using the ideal functionality $\mathcal{F}_{\mathtt{TPM2\_Create}}(\mathtt{ak}_{\mathtt{sh}})$. The TPM will create a fresh handle $\sim\mathtt{ak}_{\mathtt{h}}$ for the AK, and a new signing key pair

$$(\sim\mathtt{ak}_{\mathtt{ssk}}, \mathtt{ak}_{\mathtt{psk}} = \mathtt{pk}(\sim\mathtt{ak}_{\mathtt{ssk}})).$$

   The AK is not linked to any policy and hence its `authPolicy` will be set to `null`, in other words $\mathtt{ak}_{\mathtt{aP}} \leftarrow \mathtt{null}$. Then the TPM will return to the Router the handle of the AK ($\mathtt{ak}_{\mathtt{h}}$) and the public part $\mathtt{ak}_{\mathtt{spk}}$.

4. The Router sends to the RA Server the triple $\langle \sim\mathtt{fqdn}_{\mathtt{router}}, \mathtt{ek}_{\mathtt{pk}}, \mathtt{ak}_{\mathtt{spk}} \rangle$ and initiates the enrollment process. This is equivalent to the Router asking the RA Server to issue a certificate for the AK it created.

5. The RA Server when receiving $\langle \sim\mathtt{fqdn}_{\mathtt{router}}, \mathtt{ek}_{\mathtt{pk}}, \mathtt{ak}_{\mathtt{spk}} \rangle$ it creates a credential using exactly the same steps as described in the ideal functionality $\mathcal{F}_{\mathtt{TPM2\_MakeCredential}}$. That is, it generates a fresh challenge and then it protects this challenge using the public part of the EK $\mathtt{ek}_{\mathtt{pk}}$ and binds it with the public part of the AK $\mathtt{ak}_{\mathtt{spk}}$. This is described by `MakeCredential` that outputs `credBlob`, which is sent to the Router (see Figure 3.2).

6. The Router executes the command $\mathcal{F}_{\mathtt{TPM2\_ActivateCredential}}$, via the TPM, in order to activate the credential that is created by the RA Server. The TPM will perform the validation checks that are described in Figure 3.3 and it will output the initial `challenge` that was created by the RA Server.

7. The Router receives the `challenge` from the TPM and passes it to the RA Server in order for the later to create the AK certificate.

8. The RA Server creates the AK certificate that consists of the public part of the AK and the Router's FQDN, namely $\langle \mathtt{ak}_{\mathtt{spk}}, \mathtt{fqdn}_{\mathtt{router}} \rangle$ and the signature on this message using the secret signing key of the RA Server, $\mathtt{ssk}_{\mathtt{ra}}$. The AK certificate is then transferred to the Router.

9. Finally, the Router verifies the signature $\mathtt{signature}_{\mathtt{cert}_{\mathtt{ak}}}$ of the certificate, using the `verify` function and the public key of the RA Server $\mathtt{spk}_{\mathtt{ra}}$.

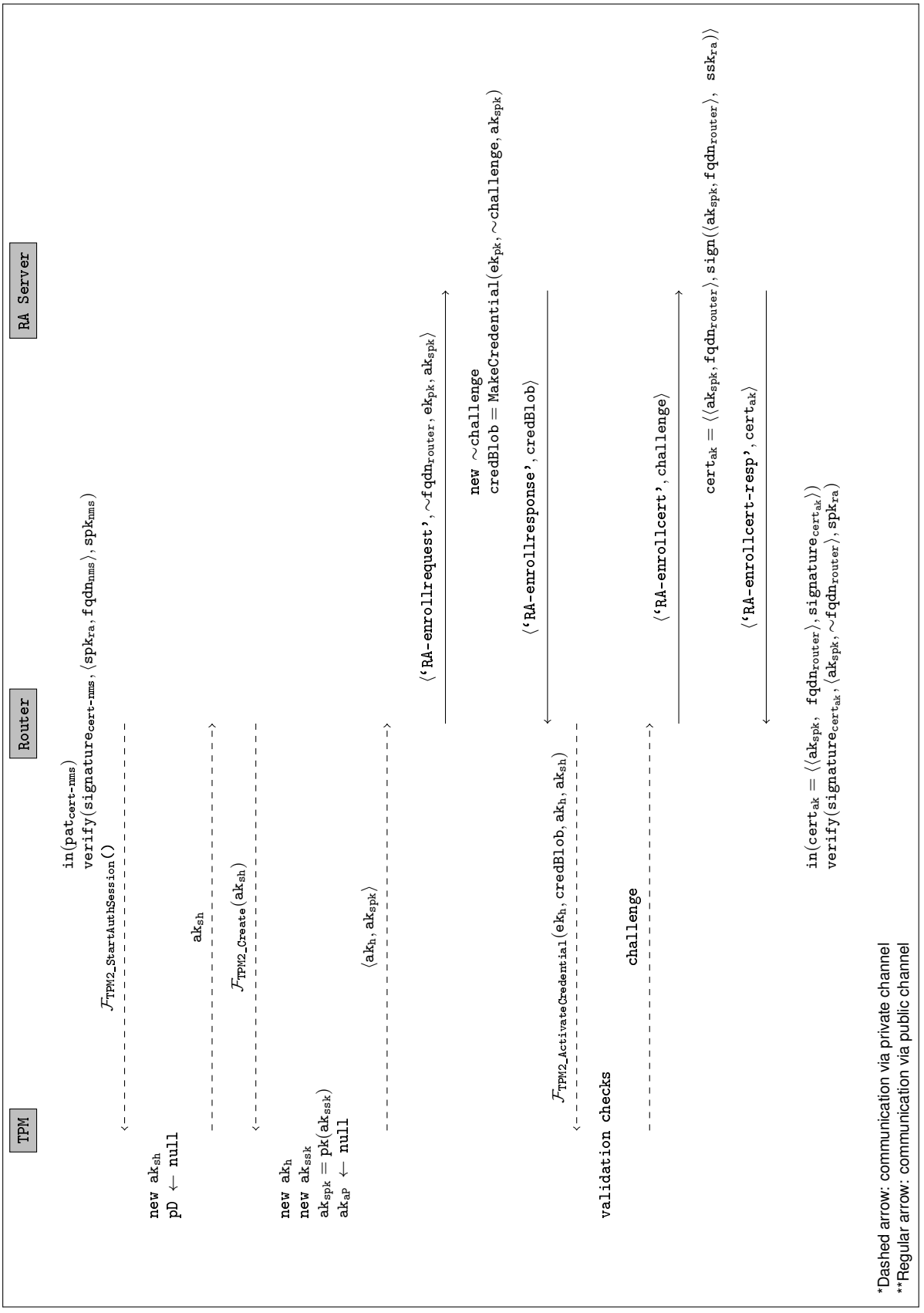

Figure 4.6: Modelling the AK creation and certification process

4.5 Security Properties

In this section we present a set of tentative security properties to be tested in the device management use case. We remark that this set of properties might be subject to changes, depending on the progress and needs of the model. The final set of properties will be documented in Deliverable D3.5.

1. **Sanity-check lemmas.** We consider a number of a sanity-check lemmas that ensure the correctness of the model. These kind of queries are formulated as reachability queries to ensure that the model executes all possible branches. In Tamarin, this correspondds to an "`exists-trace`" lemma

$$\exists t_1, \ldots, t_k.\ \texttt{Reaches}(\text{`}\texttt{label}_1\text{'})@t_1 \wedge \cdots \wedge \texttt{Reaches}(\text{`}\texttt{label}_k\text{'})@t_k,$$

   where `event Reaches('label`$_i$`')` must be located at the end of the ith branch of the model. If this lemma does not verify, then other correspondence properties might be trivially satisfied. Alternatively, we can also specify reachability of each branch in separate lemmas.

2. **Availability of keys at honest processes.** This property states that all honest parties have initial access to the trusted key material required, so that they can build the chain of trust. E.g., RA server signature public key. In order to verify this property, we define the "`all-traces`" lemma

$$\forall key_{id}, id_1, id_2, k_1, k_2, t_1, t_2.\ \texttt{HasKey}(key_{id}, id_1, k_1)@t_1 \wedge \texttt{HasKey}(key_{id}, id_2, k_2)@t_2 \Rightarrow (k_1 = k_2).$$

   It states that if two `HasKey` events are launched with the same key id, then it must be the case that the associated key is the same when the events are invoked. For example, the events

```
// RA server declares that it has its signature public key
event HasKey('spk_ra', id_ra, spk_ra);

// Router declares that it has RA server signature public key
event HasKey('spk_ra', id_router, spk_ra);
```

   must be launched at RA server and router processes respectively, so that the lemma ensures that the key identified by both processes as `'spk_ra'` matches. We require that this property hold for RA server public key, NMS server public key, and EK endorsement key.

3. **Key freshness and secrecy.** This property ensures that the created keys during the protocol execution are fresh and not available to the adversary. We recall that the SAPiC calculus uses the special action fact $K(m)$ to denote knowledge of m by the adversary. We require two lemmas to assert this property

$$\forall key_{id}, tid_1, tid_1, k, t_1, t_2.\ \texttt{GeneratesKey}(key_{id}, tid_1, k)@t_1 \wedge \texttt{GeneratesKey}(key_{id}, tid_2, k)@t_2 \Rightarrow (tid_1 = tid_2),$$

$$\forall key_{id}, id, k, t_1.\ \texttt{UsesKey}(key_{id}, id, k)@t_1 \Rightarrow \neg(\exists t_2.\ K(ptk)@t_2).$$

   The first lemma states that once key k with key identifier key_{id} is generated at two different thread executions, tid_1 and tid_2, then this is the case that $tid_1 = tid_2$. In other words,

there is no other thread execution where the same key value is generated. The second lemma asserts that once the key k is consumed by a process, then the adversary has no knowledge of it at *any* time point. We remark that authentication below can only be asserted when key secrecy is guaranteed.

4. **Authentication.** We consider the strongest authentication property from Lowe's hierarchy [17], namely, mutual, injective agreement:

$$\begin{aligned}\forall X, Y, pars, t_1.\ \texttt{Commit}(X, Y, pars)@t_1 \Rightarrow \\ ((\exists t_2.\ \texttt{Running}(Y, X, pars)@t_2 \wedge (t_2 < t_1)) \\ \wedge \neg(\exists X', Y', t_2.\ \texttt{Commit}(X', Y', pars)@t_2 \wedge \neg(t_2 = t_1))).\end{aligned}$$

   Using the standardized events `Running` and `Commit`, we verify that: "for each `Commit` event executed by a supplicant (resp. authenticator) X, associated to authenticator (resp. supplicant) Y, then Y executed the corresponding Running event earlier, and for each run of the protocol there is a unique `Commit`." In order to capture full agreement on the parameters of the protocol all generated key material must be included in the parameter vector $pars$, e.g., keys and certificates. `Commit` events are placed as late as possible on the X side, ideally, at the end of the protocol. `Running` events have to be executed as earlier as possible, when all the parameters to agree are available to Y.

5. **Transfer of information as generated.** This property ensures that cryptographic material, such as the AK certificate or the TPM quotes, are received at the destination process as generated by the process of origin. We verify the property through the lemma

$$\begin{aligned}\forall msg_{id}, id_1, m, t_1.\ \texttt{Receives}(msg_{id}, id_1, m)@t_1 \Rightarrow \\ (\exists id_2, t_2.\ \texttt{Generates}(msg_{id}, id_2, m,)@t_2 \wedge (t_2 < t_1)).\end{aligned}$$

   That is, the generated material m, identified by tag msg_{id}, and received by process id_1 was earlier created by process id_1 at an earlier time.

We note that the tentative properties that we have listed above do not consider the situation where we have compromised components that leak secrets. In that case, the lemmas have to be updated with a disjunction indicating that either the property verified, or a secret was leaked at some point. For example, for the case of the agreement lemma stated above,

$$\begin{aligned}\forall X, Y, pars, t_1.\ \texttt{Commit}(X, Y, pars)@t_1 \Rightarrow \\ ((\exists t_2.\ \texttt{Running}(Y, X, pars)@t_2 \wedge (t_2 < t_1)) \\ \wedge \neg(\exists X', Y', t_2.\ \texttt{Commit}(X', Y', pars)@t_2 \wedge \neg(t_2 = t_1))) \\ \vee (\exists C, t_3.\texttt{Reveal}(C)@t_3 \wedge \texttt{Honest}(C)@t_1).\end{aligned}$$

the disjunction on the right of the implication indicates that either there was a unique `Running` event for each `Commit`, or some party C believed to be honest revealed some of its secrets at a certain timepoint.
Again, the properties presented in this section are tentative security propertis for the use case, and they will be revised and documented in their final form in Deliverable D3.5.

Chapter 5

Conclusion

In this deliverable, we define new TPM commands and the corresponding ideal functionalities which are needed in the modelling for use case 3. We consider the operations related to credential management, encryption/decryption and sign/verifysign. These operations are the core functionalities of TPM and they are the cryptographic related functions. We notice that use case 3 fully captures those functionalities and the successful modelling of use case 3 will help us seamlessly and easily model other use cases. Via the definition, we guarantee that the security of these operations can be reduced to the secure implementation of the ideal functionalities. This matches the philosophy of D3.3 and the whole work package - analyzing the security for each use case when the designed TPM is interacting with outside surroundings. Finally, we present the security modelling for use case 3 for AK and TLS key creation and AK key certification. We will put the TLS key certification in D3.5.

Chapter 6

List of Abbreviations

Abbreviation	Translation
AE	Authenticated Encryption
AK	Attestation Key
CA	Certification Authority
CSR	Certificate Signing Request
EA	Enhanced Authorization
EK	Endorsement Key
NMS	Network Management System
PCA	Privacy Certification Authority
PCR	Platform Configuration Register
RA	Remote Attestation
SAPiC	Stateful Applied Pi Calculus
TLS	Transport Layer Security
TPM	Trusted Platform Module
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] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 104–115, London (UK), January 2001. ACM.

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

[4] 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/.

[5] 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.

[6] Vincent Cheval, Véronique Cortier, and Mathieu Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in ProVerif. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 344–358. IEEE, 2018.

[7] The FutureTPM Consortium. First report on security models for the TPM. Deliverable D3.1, FutureTPM, September 2018.

[8] The FutureTPM Consortium. FutureTPM use case and system requirements. Deliverable D1.1, FutureTPM, June 2018.

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

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

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

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

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

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

[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] 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.

[23] Trusted Computing Group (TCG). TPM 2.0 library specification - part 2: Structures. Available at: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part2_Structures_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.

Appendix A

SAPiC Code for AK Certification

```
/*****************************************************************************

        FutureTPM - WP3
        ---------------

        This file comprises the model for the device management use case. All
        processess are under the DY model, and the interaction with the TPM process
        is done as ideal functionalities.

        Use case:       Device management
        User stories:   HWDU.NO.1

*****************************************************************************/

theory DeviceManagementScenario
begin

builtins:
        asymmetric-encryption,
        signing,
        revealing-signing,
        hashing,
        multiset

functions:
        myenc/2,
        nil/0,
        kdf/2,
        mac/2,
        verifyMac/3,
        makeCredential/3,
        activateCredential/3,
        verifyCredential/3,
        pcrH/0

equations:
        verifyMac(mac(m,k), m, k) = true,
        activateCredential(n, k, makeCredential(pk(k), m, n)) = m,
        verifyCredential(n, k, makeCredential(pk(k), m, n)) = true

/*****************************************************************************
        TPM Process.
        It integrates the ideal functionalities of the TPM.
```

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

        insert ⟨'PCRList', pcrH⟩, nil;

        !(
                //TPM2_Create
                (
                        let pat_tpm_command = ⟨'TPM2_Create', authPolicy⟩ in
                        in(pat_tpm_command);
                        event TPM_ReceiveCommand(pat_tpm_command);
                        ν ~k_h;
                        lock ~k_h;
                        ν ~k_sk;
                        let k_pk = pk(~k_sk) in
                        insert ⟨'authPolicy',  ~k_h⟩, authPolicy;
                        insert ⟨'privatePart', ~k_h⟩, ~k_sk;
                        insert ⟨'publicPart',  ~k_h⟩, k_pk;
                        out(⟨~k_h, k_pk⟩);
                        unlock ~k_h
                ) +
                //TPM2_StartAuthSession
                (
                        let pat_tpm_command = ⟨'TPM2_StartAuthSession'⟩ in
                        in(pat_tpm_command);
                        event TPM_ReceiveCommand(pat_tpm_command);
                        ν ~s_h;
                        lock ~s_h;
                        insert ⟨'policyDigest', ~s_h⟩, nil;
                        event CreateHandle(~s_h);
                        out(~s_h);
                        unlock ~s_h
                ) +
                //TPM2_ActivateCredential
                (
                        let pat_tpm_command
                        = ⟨'TPM2_ActivateCredential', a_h, a_sh, k_h, credentialBlob⟩ in
                        in(pat_tpm_command);
                        event TPM_ReceiveCommand(pat_tpm_command);
                        lock a_h; lock a_sh; lock k_h;
                        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, credentialBlob) = true then
                                        let challenge
                                        = activateCredential(a_pk, k_sk, credentialBlob) in
                                        event Receive(challenge);
                                        event Debug3();
                                        out(challenge);
                                        unlock k_h; unlock a_sh; unlock a_h
                                else
                                        unlock k_h; unlock a_sh; unlock a_h
                        else
```

```
                                unlock k_h; unlock a_sh; unlock a_h
        ) +
        )
/*******************************************************************************
        Router process.
        It integrates functionality of ZTP Agent and RA Client.
*******************************************************************************/
let Router =
        ν ~swHash;
        ν ~fqdn_router;

        !(
                //Workflow (AK certificate)
                (
                        //1. Obtain RA Server cert
                        let pat_cert_nms = ⟨⟨spk_ra, fqdn_nms⟩, signature_cert_nms⟩ in
                        in(pat_cert_nms);

                        //2. Extract RA Server FQDN from cert and begin the enrolment
                        if verify(signature_cert_nms, ⟨spk_ra, fqdn_nms⟩, spk_nms) = true then

                        //3. Generate AK
                        let pat_tpm_command1 = ⟨'TPM2_Create', nil⟩ in
                        event TPM_SendCommand(pat_tpm_command1);
                        out(pat_tpm_command1);
                        in(⟨ak_h, ak_spk⟩);

                        //4.1 Get AK cert
                        out(⟨'RA_enrollrequest', ~fqdn_router, ek_pk, ak_spk⟩);

                        //6.2 Generate credential blob (RA) and verify challenge response
                        //by router
                        in(⟨'RA_enrollrequest_resp', credentialBlob⟩);

                        let pat_tpm_command2 = ⟨'TPM2_StartAuthSession'⟩ in
                        event TPM_SendCommand(pat_tpm_command2);
                        out(pat_tpm_command2);
                        in(ak_sh);

                        let pat_tpm_command3
                        = ⟨'TPM2_ActivateCredential', ak_h, ak_sh, ~ek_h, credentialBlob⟩ in
                        event TPM_SendCommand(pat_tpm_command3);
                        out(pat_tpm_command3);
                        in(challenge);

                        out(⟨'RA_enrollcert', challenge⟩);

                        //8.1 Sign AK cert. Privacy CA signs the AK certificate;
                        //RA Server sends it to the router.
                        let cert_ak = ⟨⟨ak_spk, ~fqdn_router⟩, signature_cert_ak⟩ in
                        in(⟨'RA_enrollcert_resp', cert_ak⟩);
                        if verify(signature_cert_ak, ⟨ak_spk, ~fqdn_router⟩, spk_ra)
                        = true then

                        event Debug1();

                        0
                )
```

```
        )

/*******************************************************************************
        RA server process.
        It integrates functionality of Privacy CA and RA Lib.
*******************************************************************************/
let RAServer =
        !(
                //Workflow (AK certificate)
                (
                        //4.2 Get AK cert
                        in(⟨'RA_enrollrequest', fqdn_router, ek_pk, ak_spk⟩);

                        //6.1 Generate credential blob and verify challenge response
                        //by router
                        ν ~challenge;
                        event Source(~challenge);
                        let credentialBlob = makeCredential(ek_pk, ~challenge, ak_spk) in
                        out(⟨'RA_enrollrequest_resp', credentialBlob⟩);

                        //7. Generate AK cert and send it to Privacy CA
                        //and
                        //8.1 Sign AK cert. Privacy CA signs the AK certificate;
                        //RA Server sends it to the router.

                        in(⟨'RA_enrollcert', ~challenge⟩);

                        let cert_ak
                        = ⟨⟨ak_spk, fqdn_router⟩, sign(⟨ak_spk, fqdn_router⟩, ~ssk_ra)⟩ in
                        out(⟨'RA_enrollcert_resp', cert_ak⟩);

                        //9. Store AK cert
                        //insert ⟨'ra_database', ak_spk⟩, cert_ak;

                        event Debug2();
                        0
                )
        )

/*******************************************************************************
        NMS process.
        This process integrates the NMS CA.
*******************************************************************************/

let NMS =
        ν ~fqdn_nms;
        out(~fqdn_nms);
        let cert_ra = ⟨⟨spk_ra, ~fqdn_nms⟩, sign(⟨spk_ra, ~fqdn_nms⟩, ~ssk_nms)⟩ in
        out(cert_ra);
        0

/*******************************************************************************
        Main process
*******************************************************************************/
//TPM objects
ν ~lpcBus;

ν ~ek_sk;
```

```
let ek_pk = pk(~ek_sk) in
out(ek_pk);

ν ~ek_h;
out(~ek_h);

//RA objects
ν ~ssk_ra;
let spk_ra = pk(~ssk_ra) in
out(spk_ra);

//NMS objects
ν ~ssk_nms;
let spk_nms = pk(~ssk_nms) in
out(spk_nms);

(
Router
| RAServer
| NMS
| TPM
)

/*******************************************************************************
        Lemmas
*******************************************************************************/
lemma SourcesLemma [sources]:
        "All m #i. Receive(m)@i ==⟩
                ( (Ex #j. KU(m)@j & (j⟨i))
                | (Ex #j. Source(m)@j)
                )"

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))
        )
"
```

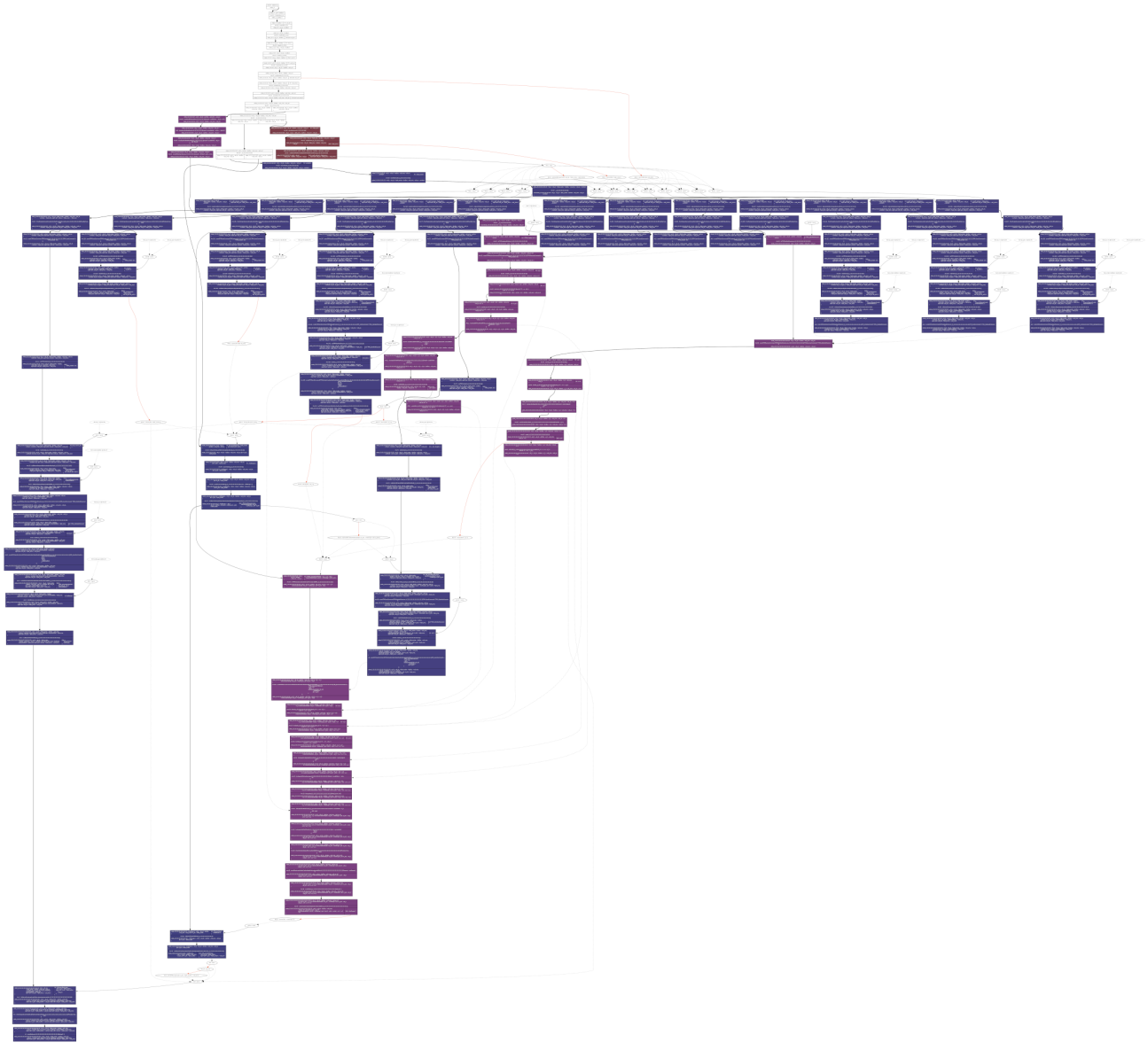

Figure A.1: Example of a reachability trace of AK certification (Router side)
The high-resolution image can be found at: https://futuretpm.technikon.com/03-WPs/WP3/D3.4/working-directory/figures/trace_ak_router_HI-RES.png

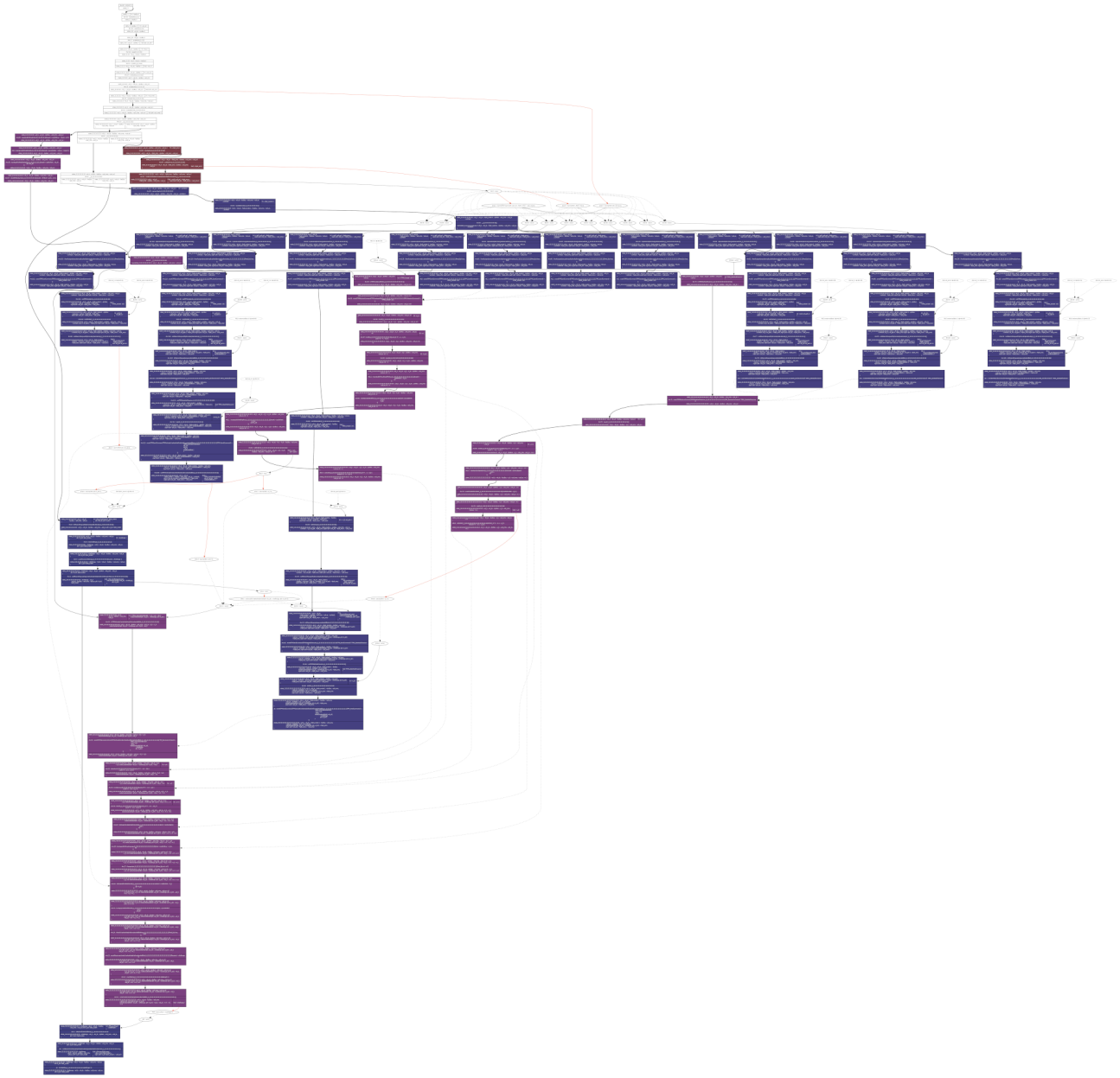

Figure A.2: Example of a reachability trace of AK certification (RA Server side)

The high-resolution image can be found at: https://futuretpm.technikon.com/03-WPs/WP3/D3.4/working-directory/figures/trace_ak_ra_HI-RES.png