Master's Thesis

Securing telecommunication network node data using TPMs

Jelena Mirosavljevic

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, March 2014.

Securing telecommunication network node data using TPMs

Jelena Mirosavljević

Ericsson
Lund

Advisor: Martin Hell & Ben Smeets

March 24, 2014

Printed in Sweden
E-huset, Lund, 2014

Abstract

A telecommunication network is a collection of nodes used by network operators. The two main tasks of a node are to transmit mobile user information and to store sensitive information of the network operator. The node is therefore required to provide a high availability system and trusted computing functionality. The purpose of the thesis was to evaluate how the requirements of such nodes could be achieved by using a key protected by Trusted Platform Module, TPM. This key is used to encrypt sensitive information and in order to provide a high availability system, the TPM protected key is stored in multiple computational units for back up purposes. This requires the key to be migratable or duplicatable.

The aim of this master thesis is to establish use cases for how the migration and duplication in TPM 1.2 and 2.0, respectively, should be performed to provide secure storage for the network operators.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Today, there are more than 6 billion mobile users [1]. The modern phone is today used for various tasks, such as phone calls, messages and internet usage. As such, this also leads to higher demands on networks when it comes to performance and capacity per individual user.

With the increase of usage and dependency on networks, there is no room for downtime. The downtime is the result of any cause that the results in the system not being able to perform the operations and tasks that it is meant for. Thus, the telecommunication network has to be a robust system which means that it is fault-tolerant to some extend and if failures occur, it has to recover quickly. In addition to robustness, the system also has to be secure. The operators that own the networks and its users have a lot of sensitive information which needs to be protected. It also has to be stored in a secure way.

Realizing the features that allow us to implement security can be done in software but may require hardware features if there are high security demands. The modern approach to achieve security is to equip the system with a small subsystem that can be realize in a trusted way with high level of assurance (the reason for why it can be trusted). Thus, the subsystem is today often referred to as a "trusted platform".

In this thesis I address aspects of a specific approach of creating trusted platforms; namely by using Trusted Computing Group technology. I will also investigate how sensitive information in network nodes can be stored in a secure way and also what is needed for the telecommunication network to be a robust system.

1.1 Problem definition

In the thesis a simplified model of the network nodes will be used and presented in this section. To structure our treatment of the protection requirements and solution, three components are distinguished

- *High Availability System*
- *Computational Units*
- *Customer*

The significant part of the telecommunication network system is built around High Availability Systems, hereafter referred to as **HAS**. Each HAS is constructed to ensure that there will be no downtime so that the users of the system are served at any time. This is done by having multiple Computational Units, **CU**, that are organized to take over each other's tasks in case any CU is unavailable or fails. The CUs are manufactured in a CU Factory and their main task is to protect the sensitive information of the customers. The CUs are sent to the HAS factory for initialization and assembly. Also, several CU's are sent to the customer as spare part units, as seen in Figure 1.1. This is in case the CUs that are installed in the HAS Factory malfunction. Each HAS is manufactured for a specific customer.

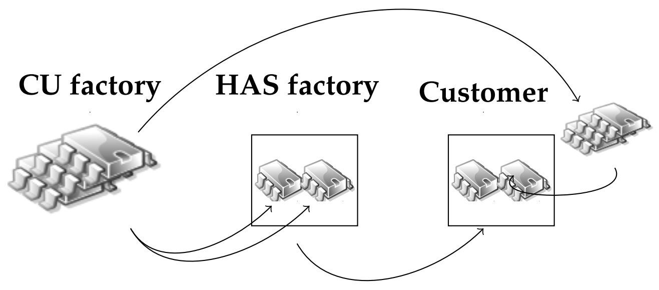

Figure 1.1: The involved parties.

Note that a customer is not the same as a user. Users can be seen as the one who owns the mobile phones and use the telecommunication network for transmitting their information. The customers are the ones who own the network, and are known as the mobile network operators. A network is in turn built on multiple nodes, where the sensitive information of the customer should be stored and protected. This protected information is referred to as a *node blob*. The CU is the component that protects these node blobs and keep them private. Thus, if a CU malfunctions, the information that is kept secret will be lost or become unreachable. This is why the HAS contains multiple CUs.

After the creation of the CUs in the CU Factory, they are sent to the Customer. This means that the CUs have to be constructed in a way that makes it easy for the customer to replace the ones who have failed by new ones. Consequently, a CU has to be generic to be able to synchronize with the system. The meaning of a generic CU is that all CUs shall be realized in the same manner before they are sent to the customer or the HAS Factory.

However, at the same time, the CU has to be customer specific, where only the customer can reach certain information. That is why we also consider a case where the CUs can be more customer specific.

1.2 Possible solutions

There are several solutions for creating a secure environment and secure storage. Trusted Execution Environment technology, TEE [2], is a secure processing environment that consists of processing, memory and storage capabilities. The TEE is isolated from the "normal" environment where the device operating system and applications run. Instead of letting the "normal" environment handle the sensitive operations, the TEE takes care of it and also stores the sensitive data in its memory. One example of this is the TrustZone technology. This technology is developed by ARM [3]. It provides a trusted area that is protected from software attacks. TrustZone consists of a security environment providing code isolation. This technique creates a separation into two parallel execution worlds, the non-secure execution environment, and a trusted certifiable secure world.

There is also virtualization [4] that functions in some sense as TrustZone. This technique separates the virtual and the physical computing platforms via a virtual machine. By creating a virtual environment, an attack by an application will not be able to reach the real physical platform. The virtual machines isolates the subsystems from each other, which is not suitable for embedded systems. Thus, subsystems are not capable to cooperate with each other in this kind of environment. Virtualization also has the draw back that it requires much more memory.

An alternative approach has been developed by the Trusted Computing Group (TCG) and relies on the use of a Trusted Platform Module [5], TPM. The TPM is a cryptographic processor that creates cryptographic keys inside of the TPM. These keys are also protected by the TPM and are used to encrypt the sensitive information before storing it outside the TPM.

The TPM is a component that has been around for years without being used to its fullest. Today, the TPM is used for Microsoft's Bitlocker [6] system. The Bitlocker function is based on locking the encryption key that is used to protect data. This function provides security by hashing the summary of hardware and software configuration. These values are then encrypted to a TPM key and stored in the PCRs, Platform Configuration Register. The values are then checked in BIOS mode, thus when the PC is started and the TPM only decrypts the data if the current PCR values match what was specified when the data was encrypted. This process provides verification that the operating system configuration has not been changed, thus not been tampered with.

1.3 Our approach

The crucial component that will be used in the CU is the TPM. The TPM is a module that can be added to an existing architecture rather cheaply. Comparing to other solutions, the TPM is also a component that does not demand any new memory and can easily be switched in the HAS environment. The other solutions are for example more sensitive to changes in the HAS.

Because the HAS should keep operational even if when replacing the CUs, the TPMs and the information they protect have to be both changeable and reachable. That is why there will be two CU equipped with a TPM in a HAS, so if one of them breaks, the other one can take its place. This is done by using the migration feature of TPM 1.2, or duplication as it is called in TPM 2.0. This feature provides the capability of the key, which the information is encrypted with, being transfered to the new CU/TPM that is inserted.

As indicated above, there are two versions of the TPM. The one that is being used is the TPM version 1.2, and this is the one that is currently on CUs. However, in a few years, the TPM version 2.0 will start to be used instead. This motivates that within this thesis we consider a uniform API, *Application Programming Interface*, that can be used for both versions of the TPM.

1.4 Aim

The aim with this thesis to

- provide secure storage of the sensitive information that a customer wants to protect in the HAS
- construct a robust system by using multiple CU each with a TPM that can take over each other's tasks in case of failure
- provide an easy solution for the customer to switch out the TPM equipped CUs
- create a uniform API that is suitable for both TPM 1.2 and TPM 2.0

1.5 Outline

This thesis is divided in 9 major sections. First an introduction to TPM will be given, both for TPM version 1.2 and 2.0. In the 3rd chapter, the migration in TPM 1.2 will be introduced, which will be more developed with the CMK mechanisms in the 4th chapter. Then, the duplication for TPM 2.0 will be described in the 5th chapter. In chapter 6 and 7 the use cases for CMK migration in 1.2 and duplication 2.0 will be described respectively. In the 8th chapter we analyze and discuss the use cases by comparing them with each other. Chapter 9 will introduce an uniform API for the different use cases from chapter 6 and 7.

Chapter 2

Introduction to the TPM

In this chapter the TPM and its application area will be described in more detail. As presented in the introduction, the main task for the TPM in this thesis is to secure the storage of keys, which are used to protect the node blobs. The two versions of the TPM operates with the encryption in different ways. Therefor, the TPM features, both for the TPM 1.2 and 2.0, and their architectures are shortly described.

2.1 TPM Features

The secure storage is provided by *Shielded Locations* which are areas on a TPM that contain data that is shielded from access by any other entity outside of the TPM. Thus, data is protected against interference from outside exposure. The data itself can be, e.g., a TPM generated cryptographic key. When the private portion of a key is not held in Shielded Locations on the TPM, it is encrypted. To access the objects that are stored in shielded locations, the TPM requires use of a Protected Capability, which is an operation that must be performed correctly. Therefore, all information on a TPM can be seen as being stored in shielded locations.

For example, the unique identity of a TPM, which is the Endorsement Key and the Primary Seed in TPM 1.2 and 2.0 respectively, are held in the shielded locations.

2.2 TPM 1.2

The TPM 1.2 was introduced in October 2003 and this section will give an overview of its components and their functions.

2.2.1 TPM 1.2 Architecture

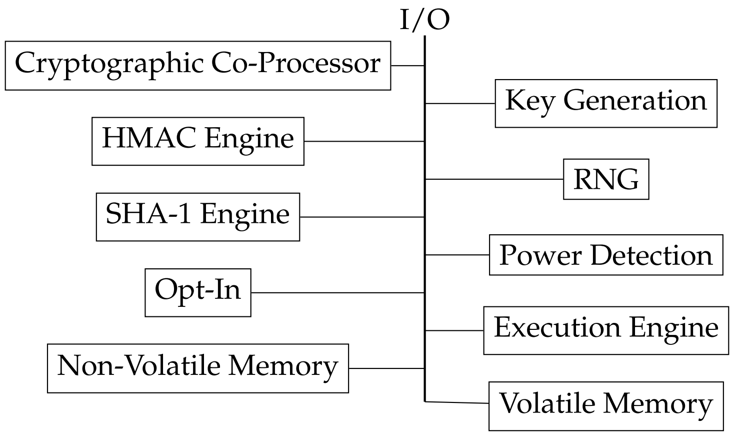

Figure 2.1: TPM 1.2 architectural overview

As mentioned, the TPM is a cryptoprocessor that performs cryptographic operations, such as encryption and decryption. To be able to generate and store keys and information, several components are involved. The components and the design of the architecture can be seen in Figure 2.1.

To start with, the TPM is a command based hardware which means that it operates only on command. Towards this end, the TPM has to have an input/output channel, *I/O*, that supports the communication. For the commands to be executed, an *execution engine*, a CPU, is needed which has the ability to execute sequential operations.

The TPM has also the capability to store its main secrets. This is done in its *non-volatile memory*. This is where the long term keys, the Endorsement Key and the Storage Root Key, which will be described later on in this chapter, are held. Their private portion will never leave the boundaries of the TPM, which means that it will never be publicly visible. Together with the long term keys, the owner authorization can be found. The owner authorization is a secret that is created when a user takes ownership over a TPM. There is also a *volatile memory* that is used as temporary storage for e.g., keys that have been loaded into the TPM for usage, but is cleared when the system reboots.

For more special tasks, like generating a RSA[1] key-pair, the cryptographic components are used. The *key generator* is the main cryptographic engine and is used for, e.g., performing the prime number validation and other RSA-specific requirements. The *RNG*, Random Number Generator, is a TPM feature that produces a unique random number sequence, a bit stream, that is used for the key creation. The RNG is also used in the creation of signatures, password phrases and the nonce which is used as an anti-replay protection. The *cryptographic co-processor* has the ability to encrypt and sign data using externally or internally generated RSA keys.

The functionality of the *HMAC Engine* and *SHA-1 Engine* is involved in the cryptographic operations by hashing data, producing a result defined as digest. The HMAC engine is used for providing information about the proof of knowledge when it comes to authorization, while the SHA-1 engine is used to hash the input data and produce a "one way" 20 byte digest.

Finally, the *Opt-In* component provides mechanisms and protections to allow the TPM to be turned on/off, enabled/disabled and activated/deactivated, while the *power detection* manages the TPM power states in conjunction with platform power states.

2.2.2 The key hierarchy organization

The EK is typically generated when the TPM is manufactured and is not changeable. This is why the key is also called the root-of-trust, which means that it is used for validating the identity of a unique TPM. Although the EK can be seen as the identity for the TPM, its use as such is not really possible. The motivation behind this is that from a privacy point of view, the repeated use of a single identity in all use cases is problematic. This is because of the possibility of linking identities, which means that the observer can trace all actions of an EK back to the owner of TPM. Therefore the TPM gives normally no access to use the EK except for a very limited number of operations. Because of the EK not being used during interaction with other entities, the TPM creates something called Attestation Identity Keys, AIK. An AIK is used as an alias for the EK and one EK can be connected to several AIKs. This is mainly done to maintain anonymity between different services.

The SRK, *Storage Root Key*, is one of the most important keys in the TPM. The SRK is generated when a user takes ownership over the TPM which is done with the TPM_TakeOwnership command and is used as the root of the key hierarchy. Besides generating the SRK, this command also provides the user with an owner secret that is used during authentication. Similar to the EK, the SRKs private part never leaves the boundaries of the TPM as described in 2.2.1. The public part on the other hand, is used to wrap new keys into the TPM. This can only be done by the keys that are of key type *storage*.

[1]RSA is an algorithm for public key-cryptography and is based on having two keys, a public and a private. For a more detailed description of the algorithm see ref [7].

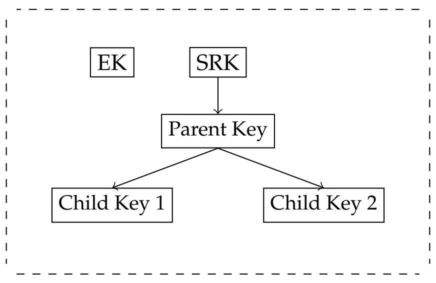

Figure 2.2: Overview of the TPM key hierarchy

As can be seen in Figure 2.2, both the SRK and the Parent Key are keys of type storage, which means that the private part of the *Parent Key* is encrypted with the public key of the SRK, and the private part of *Child Key 1 and Child Key 2* are encrypted with the public key of the Parent Key. This creates a chain of keys, as can be seen in Figure 2.3.

$$\begin{tikzcd}
 & (\text{SRK}) \arrow[d] & \\
 & (\text{Parent Key}_{sk})_{SRKpk} \arrow[dl] \arrow[dr] & \\
(\text{Child Key } 1_{sk})_{ParentKeypk} & & (\text{Child Key } 2_{sk})_{ParentKeypk}
\end{tikzcd}$$

Figure 2.3: Overview of the key storage and its encryption

If, e.g., Child Key 1 is to be loaded and used it has to be decrypted first. To be able to decrypt Child Key 1, the private part of the Parent Key is needed, however, this key itself is encrypted with the private part of the SRK. Since the SRK is not allowed to leave the boundaries of the TPM and is stored inside the TPM, the private part of the SRK is always available. The TPM then loads the Parent Key into the TPM, decrypts it with the private key of the SRK and stores it into the volatile memory, which is a temporary memory as described in 2.2.1. After decryption and storage of the private part of the Parent Key, the Child Key 1 is loaded into the TPM and decrypted.

The encryption and decryption of the child keys is called *wrapping* and *unwrapping*.

Except for the storage key, there are six more key types. A *signing* key is used for signing operations, the *binding* key is used for TPM binding commands, i.e., encryption, and the *legacy* is a key supporting both binding and signing operations. The *migration* key is the key created and used by an authority, which has the ability to actually change the location, i.e., a key that can be displaced from one TPM to another. More of this will be described in the next chapter.

The *authchange* and the *identity* key are keys that can change the authority and also provide the identity for the TPM.

2.2.3 Communication with the TPM

As described in the previous section, the TPM is a simple hardware responding to commands. Hence, the user drives the communication between the user and the TPM, which is command based. The user sends a command associated with the action it wants the TPM to perform and whether the TPM has succeeded or not, it returns a response to the user with its output. Every command consists of three parts, the header, the data and the authorization block, which will be further described in section 2.2.4. The commands can also vary depending on if it is being sent from the user, i.e. command in, or the TPM, i.e. response out. For example, the header always consists of the authorization tag, the size and the command code. However, when being sent from the TPM it also adds a TPM_RESULT structure that returns a value depending on if the command execution succeed or not[2].

The main part of a command is the data section. This is a unique part for all commands where the different structures used in these commands depends on what the user wants the TPM to do. Thus, all the properties belonging to a command are placed in the data section.

More on the data sections of a command and how they are used will be described in the subsection below with an example command, the TPM_CreateWrapKey command.

Creation of a TPM key

When creating a key, the user has to specify what kind of key is to be created and where this key should be placed in the hierarchy. As shown in Table 2.1, the parameters needed as input in this command to create the key are

- *Input*
    - parent key handle: *public part of its parent key*
    - usage secret: *a secret that is needed when the key is being used*
    - migration secret: *a secret that is needed to migrate key*
    - key info: *information about the key to be created*

[2]Besides this there is the notion of locality which is a hardware based signaling to the TPM where the command comes from, this aspect is out of scope for this thesis

Except the input parameters, which belongs to the data section, there is also the header and the authorization part which are the first three parameters respectively the last four parameters. See Table 2.1.

Structure	Name	Description
TPM_TAG	tag	Authorization tag
UNIT32	paramSize	Total number of input bytes including paramSize and tag
TPM_COMMAND_CODE	ordinal	Command ordinal: TPM_ORD_CreateWrapKey
TPM_KEY_HANDLE	parentHandle	Handle of the loaded key that can perform key wrapping
TPM_ENCAUTH	dataUsageAuth	Encrypted usage AuthData for the key
TPM_ENCAUTH	dataMigrationAuth	Encrypted migration AuthData for the key
TPM_KEY	keyInfo	Information about key to be created.
TPM_AUTHHANDLE	authHandle	Parent key authorization. Must be an OSAP session.
TPM_NONCE	authLastNonceEven	Even nonce previously generated by TPM to cover input
TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
BOOL	continueAuthSession	Ignored
TPM_AUTHDATA	pubAuth	Authorization HMAC key: parentKey.usageAuth

Table 2.1: The TPM_CreateWrapKey input command

If the key is created as a migratable key, then the key info which is the TPM_KEY structure, should be modified. The definition of a key structure[10] is

```
typedef struct tdTPM_KEY12{
  UINT16 fill;
  TPM_KEY_USAGE keyUsage;
  TPM_KEY_FLAGS keyFlags;
  TPM_KEY_PARMS algorithmParms;
  UINT32 PCRInfoSize;
     [size is(PCRInfoSize)] BYTE* PCRInfo;
  TPM_STORE_PUBKEY pubkey;
  UINT32 encDataSize;
     [size is(encDataSize)] BYTE* encData;
} TPM_KEY12;
```

The TPM_KEY12 structure contains the TPM_KEY_USAGE and the TPM_KEY_FLAGS fields where information about the key is given. For example, if the key is to be a storage key and migratable, then the keyUsage value is set to 0x0011 and the keyFlag value is set to 0x00000002.

This structure also contains the information regarding the algorithm for this key, the PCR value, and the public and the private portion of the key.

Except from ordinary values as just mentioned, a structure in a command can contain other structures. The TPM_KEY structure contains the TPM_KEY_PARMS structure which in its turn contains the TPM_ALGORITHM_ID structure. This creates a chain of structures which are nestled into each other, which needs to be fulfilled before sending the command.

After the command has been sent to the TPM, the TPM creates the key with the specifics the user asked for, and returns the wrapped key, which is a TPM_KEY structure, as the data section. The output structure can be seen in Table 2.2.

- *Output*
    - wrapped key: *includes the public and the encrypted private key*

Type	Name	Description
TPM_TAG	tag	Authorization tag
UNIT32	paramSize	Total number of input bytes including paramSize and tag
TPM_RESULT	returnCode	The return code of the operation
TPM_COMMAND_CODE	ordinal	Command ordinal: TPM_ORD_CreateWrapKey
TPM_KEY	wrappedKey	The TPM_KEY structure(public and encrypted key)
TPM_NONCE	nonceEven	Even nonce newly generated by TPM to cover outputs
TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
BOOL	continueAuthSession	Continue use flag, fixed at FALSE

Table 2.2: The TPM_CreateWrapKey output command

2.2.4 Authorization Protocol

For the ability to send commands between the user and the TPM, an authorization session has to be created. This authority session can be done in two ways, either by using the Object Independent Authorization Protocol, OIAP or by using the Object Specific Authorization Protocol, OSAP. The purpose of these authorization protocols is to prove to the TPM that the requestor has permission to perform a function and use some of its objects. This is done by providing the *shared secret*, which was created during the TakeOwnership process.

Both of the authorization protocols have their authorization blocks which are attached to the command message block. The OIAP, is an authorization protocol that is not tied to any object, or better said an entity. This means that there will *not* be any authorization blocks in this message. Thus, no authorization is needed.

The OSAP is on the other hand a protocol that does tie into a single entity, meaning that the object used demands authorization. This protocol can be seen as a challenge-response authorization protocol and is based on the "rolling nonce" technique [5]. For example, the TPM creates a nonce and sends it to the user. When the user has received the nonce, it is included into the response command with a new nonce created by the user. This is all hashed with the shared secret which is only known by the TPM and the TPM owner. This technique will prevent replay and man-in-the-middle attacks and is only needed with certain commands.

2.3 TPM 2.0

The TPM 2.0 was introduced in November 2013 with its 0.99 Specification and this section will give an overview of its components and functions.

2.3.1 TPM 2.0 Architecture

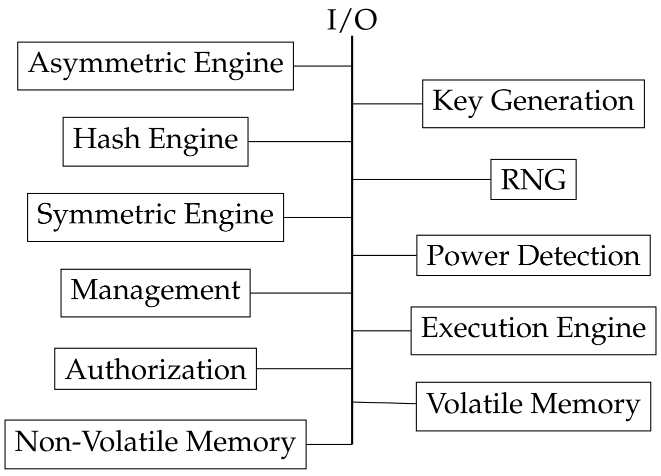

Figure 2.4: Architectural overview, TPM 2.0

The architecture for TPM version 2.0 [9] is rather different compared to the architecture for TPM 1.2 but they share some commonalities. The first four elements mentioned in 2.2.1, which are the *I/O* channel, *execution engine*, *non-volatile memory* and the *volatile memory* have the same function in TPM 2.0.

Other than the four functions that were mentioned, the TPM 2.0 also contains elements such as the *hash engine*, *RNG* and the *power detection* that have the same functions as in TPM 1.2.

One main difference between the versions is that the latter version has both an *asymmetric engine* and a *symmetric engine*. Thus, TPM 2.0 has the capability to create both asymmetric and symmetric keys. This also means the *key generation* is slightly different from the one in TPM 1.2. The keys that are being generated in the asymmetric and the symmetric engine respectively, are generated with the help of a Primary Key. The primary key is derived from a *seed value*, which is a secret value. More about how this works will be explained in section 2.3.2.

When asymmetric keys are being generated, the use for these are the same as in TPM 1.2, i.e., keys for encryption and decryption. However, when the symmetric engine is being used it mostly generates "wrapping" keys that are used to wrap its child keys.

There is also an element called *authorization*. The authorization feature is called at the beginning and at the end of a command execution. This subsystem checks that the correct authorization for use of the shielded locations has been provided. The last element in the architecture, as seen in Figure 2.4, is the *management* function. This feature controls the TPM operational states and the state transitions.

2.3.2 The key hierarchy organization

The Primary Seed is a large random value that is created and permanently stored in the TPM. Thus, this value will never leave the TPM in any form. The primary seed is only used when generating *primary objects*. The method used to do this is the key derivation function (KDF[3]) and the parameters the caller provides of an object to be created.

When the primary object is generated, the TPM uses the parameters of that object and the primary seed to generate a symmetric key to encrypt the sensitive portion of the object, i.e., the private data and authorizations. The primary seed is applied in three TPM key hierarchies, for the endorsement key hierarchy, for the storage root key hierarchy and the platform firmware hierarchy. Each seed value has a different life cycle but the way it seeds the associated hierarchies is approximately the same.

First we have the Endorsement Primary Seed, also known as the EPS, is used to generate EKs. The TPM creates an EPS when the TPM is powered on and there is no existing EPS. The TPM manufactures may inject or replace an EPS, however, in the latter all the objects in the Endorsement Hierarchy are invalid. Unlike in TPM 1.2, TPM 2.0 can contain several EKs in the EPS hierarchy. When generating a Primary Object in the endorsement hierarchy, authentication is required. As seen in Figure 2.5, the EPS hierarchy is empty, this is because the SPS is seen as the root of the key hierarchy that is used for secure storage.

The SPS, Storage Primary Seed, generates the keys that serve as Storage Root Keys. The SRK is used for the protection of storage of keys and its hierarchy is controlled by the platform owner.

The third hierarchy is the Platform Primary Seed, PPS. This hierarchy is used to control the platform firmware and contains keys that are exclusively used by platform firmware and should not be made available to user-installable software.

Similar to the TPM 1.2 hierarchy, the SPS hierarchy creates a tree of encrypted keys. Thus, the parent keys encrypts the child keys. The difference is that in TPM 2.0, the parents encrypt the child keys with a symmetric key that is retrieved from the parameters of the object and a seed stored in the parent.

For example, the SRK has a symmetric key that was created on the basis of the SPS seed value and the properties of the SRK. This key is used to encrypt the

[3]For a more detailed description of this method and the KDF function, see ref. [9].

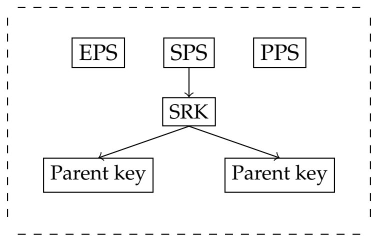

Figure 2.5: Overview of the TPM key hierarchy

Parent Key. The Parent Key in turn creates a symmetric key from its parent key's seed value, in this case the SRKs seed value, and the properties of the Parent Key, and uses this to encrypt its child keys.

However, if the key is a duplicated key, which means it has been attached to the hierarchy, then there is a special duplication symmetric key that will keep the sensitive data of the key encrypted.

There are two different keys that can be generated in TPM 2.0. The first key is the primary object, that is generated below the SPS, EPS and PSP. For example, a SRK that is a child key to the SPS is a primary object. The same goes for an EK that is a child key to EPS.

The other key is an ordinary key that is generated with the help of a random number generator. Compared to the previous version of the TPM, there are only two key types that are used to determine how the TPM may use an object, the *sign* key and the *decrypt* key. The sign key uses its private part in signing purposes while the decrypt key uses its private part to decrypt the entity the public part encrypted. There is also an attribute, *restricted*, that is set together with one of the selected key types and restricts the usage of the key. For example, if set with the decrypt key then the key can only decrypt entities with a certain structure. The restricted decrypt key is also known as the *storage* key.

2.3.3 Communication with the TPM

The communication between a TPM 2.0 and an user is very similar to the communication described in 2.2.3. The communication is still based on commands that are divided in multiple sections. However, unlike the command structure in TPM 1.2, the command structure in TPM 2.0 contains of two parts. The first part is the header and the second part is the data. This can be seen in the example command below, with the TPM2_Create command.

Creation of a TPM key

This command is used to create an object that can be loaded into a TPM using TPM2_Load. If the command completes successfully, the TPM will create the new object and return the objects *creation data*, its *public area* and its encrypted *sensitive area*.

- *Input*
  - @parent key handle: *public part of the parent key*
  - sensitive data: *a secret that is needed when the key is being used*
  - public template *algorithm information about the key to be created*
  - outside info: *buffer area containing algorithm ID and digest*
  - creation PCR: *PCR that will be used in creation data*

As seen above, the input parameters are very similar to the ones used for TPM 1.2. There is a distinction with the parent key handle comparing to the one used in the previous example and that is the "@" symbol in front of the parent key handle. This means that an authorization session is required. More about the authorization sessions will be described in section 2.3.4.

The *sensitive data* parameter defines the initial data value which is placed in the sensitive area of a created object. However, this parameter is only used when creating symmetric keys. If asymmetric objects are to be generated the data in the sensitive data structure should be empty. The *public template* parameter declares a structure that contains all of the fields necessary to define the new object and the *outside info* parameter is used to connect the object to its parent. The last input parameter, *creation PCR*, is connected to a PCR list that indicates which PCR values are associated with the object being created.

Unlike the previous version of the TPM, this version has no keyFlag property that decides if a key is duplicatable, i.e., a key that can be relocated from one TPM to another, as described in section 2.2.3. However, there is an authorization method which is designed to be used in the duplication context. There are also two attributes, the *FixedParent* and the *FixedTPM* that are set when creating an object. The FixedParent value is set if the user wants the object to have a fixed parent. Thus, this key cannot be duplicated. However, if the parent key is duplicated, then all of its children, even though the fixedParent value is set, are duplicated as well. The FixedTPM value determines that the algorithm for this object should be the same as the algorithm that is used by its parent key.

The response to this command is slightly different from the output to the TPM_CreateWrapKey command in TPM 1.2.

- *Output*
  - private part: *the private portion of the object*
  - public part: *the public portion of the created object*
  - creation data: *information about the creation environment for the object*
  - creation hash: *hash of creation data*
  - creation ticket: *a ticket that validates that the key was created in the TPM*

In the output command, the private part and the public part of the created object are separated into their own structures. Thus, when returning the generated object, the public part and an encrypted version of the private part are returned separately when the command has been executed. The creation data parameter contains the information about the created object such as the parent name, the parent algorithm and PCR digest, if used. The creation hash is a digest of the creation data. Also, a creation ticket is given in the response command. This ticket is used to bind the creation data to the object to which it applies. It also contains a proof that the object has been created in a TPM and belongs to a certain hierarchy.

2.3.4 Authorization Area

Unlike TPM 1.2, the latter version does not use any authorization protocols. Instead, the authorization is provided as authorization area in structured data that follows the command data. There are three authorization types; password, HMAC and policy. The authorization area is only present in a command if the authorization tag is set to "session" and not all commands demand authorization.

- To use a password as authorization the path between the caller and the TPM has to be trusted. This is because the password is in plaintext provided to the TPM. This kind of authorization does not require the creation of a session, thus, a password authorization does not use nonces. When creating an object, the authValue is set to the chosen password and it must be provided when the object is being used.
- The HMAC authorization type is similar to the OSAP authorization in TPM 1.2, described in section 2.3.4. This is a session based authorization which are indicated by the TPM2_StartAuthSession command. When executed, the caller indicates, among other things, the size nonce to be used and an initial nonce value. An HMAC is a form of symmetric signature over some data which provides assurance that the protected data was not modified and that it came from an entity with access to the key value. To have usefulness in protecting data, the key value needs to be a secret or a shared secret. For more information about how this secret key is computed, see ref [9].
- The third and the last authorization type is the policy session. This type of authorization is, e.g., used during duplication. In a policy session, the authPolicy entity is used instead of the authValue as in password authorization. A policy is a restriction for an object. For example, a policy may not allow the use of the object after a specific time or unless some PCRs have a specific value. There are several different policies that can be used in different situations.

  When a policy session is created, the policyDigest always starts at zero and increases as the different policy assertions are chosen. A combination of one or multiple assertions are used to construct the authorization policy. These are then given in a specific sequence to the TPM which modifies the policyDigest, i.e., updates it after each given policy. At the end, the policyDigest will have a specific value which is then used as the authPolicy which in turn will be included in an object that is to be created. Thus, if this object is to be used later on, then the same policy sequence has to be executed as an authorization.

For example, when creating an object, the user selects three assertions, *policy A*, *policy B* and *policy C*. The digest of these three policies, e.g. *abc*, will become the authPolicy in the created object. When the production of the object is completed and is to be used, the order of the policies has to be executed once again. If the given digest is *abc*, then the authorization was achieved. However, if the digest has a different value, the authorization will be denied.

A policy may also be expressed in an equation as a set of assertions that must be satisfied before a policy can be valid. There are two equation types, the Policy AND and the Policy OR.

Policy AND requires that all three assertions should be true, this could be written as

$$A\&B\&C$$

Policy OR, however, requires that one of three assertions should be true. This could be written as

$$A|B|C$$

More details about how policies are being used during duplication will be given in Chapter 5.

Chapter 3

Migration in TPM 1.2

The CU entities protect the node blobs that exist in the HAS entities. The TPM in the CU has keys that play a role in this protection and when a CU malfunctions its keys will become unavailable. An approach to have several CU cooperate on the protection of a node blob is to have the same key present in the respective TPMs. One approach to achieve this is called migration in TPM 1.2

In this chapter the migration between TPMs will be described.

3.1 Overview

Key migration can be done in two ways, directly between two TPMs or with the help of a *Migration Authority*, MA.

The MA is an authority that has the ability to perform as an intermediary between two TPMs. This is done with a key that is created with keyUsage set to MIGRATE.

3.2 Commands for migrating

To be able to succeed with the migration of a key from one TPM to another, a chain of commands has to be executed.

In this section, the commands regarding a key migration[12] will be described.

3.2.1 TPM_CreateWrapKey

- *Input*
  - parent key handle
  - owner Secret
  - migration secret
  - key info
- *Output*
  - wrapped key

With the SRK as the parent key, which was created during the TPM initialization, the TPM_CreateWrapKey command is to be used to create its child key, the migratable key. The creation of the migratable key has to be authorized which is realized by providing the owner secret. Also, a migration secret is provided which later will be used as an authentication when the key is to be migrated. The key info among the input parameters, stands for the TPM_KEY12 structure, as described in section 2.2.3. When creating a migratable key it is important to use the the keyFlag field. If the customer wishes to use the migration key to store other keys below it, then the keyUsage value has to be set to *storage* and the keyFlag value has to be set to *migratable*. The migratable storage key can only store other migratable keys.

However, if the key is to be used for encrypting information then the keyUsage value can be set to *binding* and the keyFlag value to migrate. The structure used for the definition of what kind of key it should be is shown in more detailed form in section 2.2.3.

After the execution of the TPM_CreateWrapKey command the TPM sends a wrapped key as the output, i.e., the newly generated key encrypted with its parent key.

3.2.2 TPM_AuthorizeMigrationKey

- *Input*
    - migrate Scheme
    - public key
    - owner Secret
- *Output*
    - migrationKeyAuth

After the key has been created, the TPM has to get the authorization rights to migrate this key which has to be done with the help of the TPM owner. Once this command has been executed, the TPM owner allows the users to migrate without any further involvement of the TPM owner.

This is done by creating a TPM_MIGRATIONKEYAUTH structure

```
typedef structure tdTPM_MIGRATIONKEYAUTH {
  TPM_PUBKEY migrationKey;
  TPM_MIGRATION_SCHEME migrationScheme;
  TPM_DIGEST digest;
} TPM_MIGRATIONKEYAUTH;
```

The TPM_MIGRATIONKEYAUTH structure consists of TPM_PUBKEY that represents the public key of the migration entity, the TPM_MIGRATION_SCHEME that represents the chosen migration scheme and the TPM_DIGEST that represents a digest value of the composition of migration key, migration scheme and the TPMProof.

There are two types of migration schemes, REWRAP and MIGRATE. The migratable key is "re-wrapped" in both schemes. The meaning of "re-wrapped" is when the key is first decrypted with its own parent key, and then encrypted with the parent key of the destination. The REWRAP scheme allows the destination parent key to directly load the migratable key into the destination TPM. If the MIGRATE scheme is chosen, the key has to be converted before it can be loaded into the destination TPM. The latter is done by the TPM_ConvertMigrationBlob command.

The output of this command is the migrationKeyAuth, which provides the authentication for the migration.

3.2.3 TPM_CreateMigrationBlob

- *Input*
  - parent key handle
  - migration scheme
  - migrationKeyAuth
  - owner Secret
  - migration Secret
- *Output*
  - size of a random number
  - random number
  - out data size
  - outData

TPM_CreateMigrationBlob command creates a key blob[1] which is used as an envelope with the migration key and its associated information in it during migration. This is done in the TPM and the input to this command, besides the migrationKeyAuth given in the previous command, is the parent key, the owner secret, migration secret and the migration scheme.

The parameters in the output of this command depends on the chosen migration scheme. If the migration scheme is set to REWRAP, then the size of the random number is set to 0, which means that the random number is not used. However, if the migration scheme is set to MIGRATE, then the random number is used for XOR encryption of the key. In addition to this, the key blob is returned and its size.

[1]"blob" refers here to another type of blob than the node blob discussed earlier.

3.2.4 TPM_ConvertMigrationBlob

- *Input*
  - Private key of the parent
  - Size of data
  - inData
  - Size of a random number
  - Random number
- *Output*
  - OutData Size
  - OutData

After receiving the blob from the sender TPM, the destination TPM uses the TPM_ConvertMigrationBlob command to convert the key blob to data that can be loaded into the TPM. If the MIGRATE scheme is chosen this command removes the envelope the user created with TPM_CreateMigrationBlob. The input parameters for this command is the private key of the parent, the data and the random number. The random number is now used once again to remove the XOR encryption.

The output of this command is the data, i.e., the key that can be loaded by the TPM_LoadKey command.

3.2.5 TPM_MigrateKey

- *Input*
  - MA key handle
  - public key
  - Data size
  - inData
- *Output*
  - Out data Size
  - Out data

As described in the beginning of this chapter, there are two methods for migrating a key, sending it directly from a sender TPM to a destination TPM or use a MA as an intermediary.

The TPM_MigrateKey command is executed in the latter method after the TPM_CreateMigrationBlob command. When a key is being migrated, it is encrypted with the parent key of the destination. In this case the parent key will be a migration authority key. Before the key blob can be forwarded to the destination TPM, the MA has to decrypt the key blob with its private key and encrypt it with the destination TPMs public key.

The input parameters for this command is the MA key that performs the migration, the public key to which the key blob is to be migrated, i.e., destination parent key and the key blob itself.

The output for this command is the re-encrypted blob which can now be forwarded to the destination TPM.

3.3 Migration alternatives

A migration can be orchestrated in different ways. In this section the sequence of the commands described in section 3.2 will be used to demonstrate the different migration methods.

Direct migration

The direct migration, i.e., from one TPM to another, is conducted in two steps.

- The first step is made at the sender TPM where the creation and authorization of the key is done. After successful execution of the two commands, the key blob is created before it is sent to the destination TPM.

  1. TPM_CreateWrapKey
  2. TPM_AuthorizeMigrationKey
  3. TPM_CreateMigrationBlob

- The last step is done at the destination TPM after receiving the key blob. The TPM converts the blob and encrypts it into its own hierarchy. If the key is to be used it is also loaded with the LoadKey command.

  4. TPM_ConvertMigrationBlob
  5. TPM_LoadKey

Migration through a MA

In this example, the migration is conducted with the help of a MA in three steps.

- This step is performed in the same manner as in the previous example. However, the key blob is encrypted with the key of the MA before it is sent.

  1. TPM_CreateWrapKey
  2. TPM_AuthorizeMigrationKey
  3. TPM_CreateMigrationBlob

- When the key blob has arrived at the MA, it is decrypted with the MA private key and encrypted with the destination TPMs public key. The command used for this is the

4. TPM_MigrateKey

- The third and the last step is done in the destination TPM. When the migration blob arrives at the TPM converts the blob and encrypts it in its own hierarchy. If the key is to be used it is also loaded with the LoadKey command.

  - TPM_ConvertMigrationBlob
  - TPM_LoadKey

Chapter 4

CMK migration in TPM 1.2

The difference between using the method of migration described in Chapter 3 and the method that will be presented in this chapter is the creation and migration of Certifiable Migratable Keys, also known as CMKs. The difference between these certifiable keys versus the ordinary keys is that these keys are by guaranty created and protected by the TPM. This can and is verified by two of the most important entities in this chapter, which are the MA, Migration Authority, and the MSA, Migration Selection Authority.

- **MA**: Migration Authority
    - performs the migration of the keys
- **MSA**: Migration Selection Authority
    - controls migration of keys

The first one, the MA, is seen as an intermediary because this is the entity that receives the CMK from the sender TPM before sending it further on to the destination TPM, as briefly mentioned in the previously migration case. The MSA has a much bigger role because the MSA is the authority that approves the migration and is responsible for auditing that the CMK is not sent to an unauthorized TPM. The latter is done with the help of tickets. There are three types of tickets, the authorization ticket, the "export" ticket and the "import" ticket. These tickets contain information about who signed this ticket, where it is headed, where it came from, respectively, and what is being sent. More about these tickets will be described in section 4.1.2.

Simply said, a CMK **must** be a migratable key for which the destination is restricted. During the creation of the CMK a MSA list is attached which lets the key only migrate to one of the MSAs on the list.

4.1 Overview

In this section, some details of the tickets and the MSA will be described. These properties are significant for the CMK migration.

4.1.1 MSA

During the ordinary migration as described in section 3.3, the TPM owner is in charge of which destination TPM the migration key will be migrated to. However, this is not always a secure way of migrating a secret key. That is why the CMK requires the use of a third part.

The third part, also known as the authority, will be the MSA which will be involved in the migration of a CMK with the help of restrictTickets. Without these tickets it would not be possible for the TPMs to perform the migrations because it *must* be done by a trusted authority.

4.1.2 Tickets

When migrating an object, the destination has to be authorized by the MSA. The authority audits if the sender TPM has the rights to send the object to a destination TPM. If this is the case, then the MSA creates a digest by signing the

$$\{\text{MSA, send.TPM, dest.TPM}\}$$

structure with its private key. The three elements stands for

- The public key of the authority
- The public key of the parent key of the sender
- The public key of the parent key of the receiver

The restrict ticket, which is the generated digest, is then sent to the TPM which executes the TPM_CMK_CreateTicket to authorize the MA approval. This in turn produces a signature ticket. The TPM_CMK_CreateTicket command will be further described in section 4.2.4.

The MSA also audits if the destination TPM has the rights to receive the object from the sender. These two types of tickets will be called export respectively import tickets.

4.2 Commands for migrating with a CMK

This section will give a short overview of the commands needed during an execution of migrating a CMK [12]. Each command is described by input and output parameters that are needed for a successful execution of a command.

4.2.1 TPM_CMK_ApproveMA

- *Input*
    - Migration Authority Digest

  - Owner Authentication

- *Output*
  - Authorization Ticket

The TPM_CMK_ApproveMA command is the first command that is executed in the migration process. This command creates an authorization ticket that allows the TPM owner to specify which MSAs are approved. This also allows the users of the TPM to create the CMK without any further involvement of the TPM owner.

The first input parameter of this command is a digest value of the TPM_MSA_COMPOSITE structure.

```
typedef struct tdTPM_MSA_COMPOSITE{
  UINT32 MSAList;
  TPM_DIGEST[] migAuthDigest;
} TPM_MSA_COMPOSITE;
```

The migAuthDigest is an arbitrary number of digest of public keys belonging to Migration Selection Authorities. Thus, it specifies which MAs the TPM approves. The second parameter in the structure is the MSAList which is the number of the authorities approved.

In addition to the MSA_COMPOSITE structure, the owner authorization needs to be provided.

The output of this command is the authorization ticket, which is a HMAC of the migration authority digest using the TPMProof as the secret. This will also be called AuthTicket and is a TPM specific value.

4.2.2 TPM_CMK_CreateKey

- *Input*
  - Parent key
  - Owner Secret
  - Key info
  - AuthTicket
  - Migration Authority Digest
- *Output*
  - Wrapped key

With this command, the creation of the key to be migrated is realized. The different inputs that are needed for executing this command, are the parent key, the owner authorization, i.e., the owner secret, the key information, the authTicket given in the previously command and the digest of the authTicket.

The key information has the same structure as the one described in section 2.2.3. The keyFlag however, **must** be set to migratable. Also, when creating this key an audit will be performed so that the parent key of this key does NOT have the migratable flag set. This is done to insure that this CMK is generated in the TPM and that the private key of the generated CMK is protected by a TPM key that cannot be migrated.

The output of this command is a wrapped key, thus, a CMK encrypted with the parent key.

4.2.3 TPM_AuthorizeMigrationKey

This command is the same command as the one used in the regular migration in the previously chapter, section 3.2. However, there is a difference between these two cases. The parameters, the public key, the migration scheme and the authHandle, are still sent to get the TPM_MIGRATIONKEYAUTH structure but in the CMK case there are two migration scheme types that differ from those used in the migration case. The two TPM_MIGRATE_SCHEME types that can be chosen are either the

- TPM_MS_RESTRICT_APPROVE type

  or the

- TPM_MS_RESTRICT_MIGRATE type

The first type, i.e., APPROVE, includes not only a MSAList that binds the CMK to one or several MSA, but is also in need of tickets. These tickets are created by the MSA that verifies and signs the destination as described in section 4.1.2. This means that a TPM can send a CMK directly to another destination TPM, with the help of an exportTicket and an importTicket from the MSA. In addition to sending the CMK directly between the TPMs, an MA can also be used. However, this requires two extra tickets, an import and export ticket respectively. Which means that the CMK is first sent from the TPM to a MA, and then from a MA to a destination TPM. Hence, the APPROVE scheme needs approval from the MSA for a successful migration.

The MIGRATE type however, does not need any tickets at all. This scheme is used when a CMK is sent between a TPM and a MSA, and because the MSA is already authorized, it is a trusted part.

The output message block given with this command is the same result as in the previous example in Chapter 3, under the TPM_MIGRATIONKEYAUTH command description.

4.2.4 TPM_CMK_CreateTicket

- *Input*
  - Public key
  - Signature
  - SignatureValue
- *Output*
  - sigTicket

This command is used to verify that the restrictTicket given from the MSA is trustworthy. Thus, that an authentic MSA has approven the destination of the CMK.

By using the TPM_CMK_CreateTicket command with provided parameters; the public key of the destination, the public key of the sender and the digital signature of the MSA, the TPM can verify that this is really created and approved by a MSA. This is done by comparing the digest given from the MSA, i.e., the restrictTicket, with the digest that is given by the output of the command, i.e., sigTicket.

4.2.5 TPM_CMK_CreateBlob

- *Input*
  - Parent key
  - Migration scheme
  - AuthTicket
  - Digest of the public key
  - MSA List
  - restrictTicket
  - sigTicket
- *Output*
  - Random number
  - OutData

In this command the user creates a CMK blob that is sent to the destination. The input parameters needed is the parent key, the chosen migration scheme, the authTicket that verifies that the TPM owner has authorized this migration, the MSAList and the digest of the public key of the entity being migrated. Also, for creating this blob, the restrictTicket and the sigTicket are needed. The restrictTicket is the ticket given from the MSA, and the sigTicket is the digest value created to verify the authority. When using the MIGRATE scheme type these two parameters will be zero.

The output of this command is the re-encrypted data which is sent to the destination.

4.2.6 TPM_CMK_ConvertMigration

- *Input*
  - restrictTicket
  - sigTicket
  - public key of the key-to-be-migrated
  - MSAList
  - migrationKeyAuth
  - random
  - randomSize
- *Output*
  - outData
  - outData size

This command is similar to the TPM_ConvertMigrationBlob command described in section 3.2. Some of the input parameters are the public key of the key to be migrated, the MSAList and the migrationKeyAuth, i.e., the authorization of the migration key. Among the input parameters, the restrictTicket and the sigTicket can also be found.

With the input parameters, this command converts the key blob created by the CreateBlob command into a loadable key by removing the *random* parameter and wrapping the key into its own hierarchy, i.e. with a parent key.

The output of this command is the CMK that can be loaded with the TPM_LoadKey command.

4.2.7 TPM_MigrateKey

The input for this command is the blob given in the previously command and is only used when a MA is the middle hand between two TPMs. This means that if the migration is direct between two TPMs or if the MSA is the middle hand this command is **not** used.

For more details about this command, see section 3.2.

4.3 CMK migration alternatives

In this section the various ways of migration with the commands described in this chapter are presented.

The MSA is involved in all of these cases that will be described and is responsible of ensuring that the right TPM is sending the right CMK to the right TPM destination.

Migrating directly between TPMs

When talking about direct migration between two TPMs, the MSA and its involvement can be forgotten. Even though a CMK is directly sent from a sender TPM to a destination TPM, multiple steps involving the MSA are made to make that possible.

- The first stage of the migration begins in the sender TPM, where the owner has to approve the different MAs that can be used as the middle hand even though it will not be needed in this case. On the other hand, the authTicket that is given in the output is needed for the next step which is to create the key. The third and last step is to authorize the migration key, that is, how will this key be migrated and which migration scheme will be used.

  1. CMK_ApproveMA
  2. CMK_CreateKey
  3. CMK_AuthorizeMigrationKey

  After the generation of the CMK is completed, the MSA needs to approve of the CMK migration to the destination. This is done by creating the restrictTickets. To approve the authority, the sigTicket is created by the TPM. If the digest value of these two tickets match, then the CMK blob is created before it is sent to the destination TPM.

  4. CMK_CreateTicket
  5. CMK_CreateBlob

- When arriving at the destination TPM, the CMK blob has to be converted. However, to be able to do this, the destination TPM needs to be certain that this migration is authorized by the MSA. The MSA then sends the restricTicket to the destination TPM, which is once again verified by creating the sigTicket. When this step is performed, the destination TPM wraps the CMK with its parent key.

  6. CMK_CreateTickets
  7. CMK_ConvertMigration

- The CMK can now be used using the TPM_LoadKey command.

Migrating with the help of an MA

In this migration method the use of a MA will be described. Comparing this migration alternative with the previous one, the stages are very similar to the once just noted above with some small differences.

- This step is computed in the same way as step one in the previous example.

  1. CMK_ApproveMA

2. CMK_CreateKey
3. CMK_AuthorizeMigrationKey

- After the generation of the CMK is completed, the MSA needs to approve of the CMK migration to the destination. This is done by creating the restrictTickets. To approve the authority, the sigTicket is created. If the digest value of these two tickets match, then the CMK blob is created before it is sent to the destination.

4. CMK_CreateTickets
5. CMK_CreateBlob

- When the MA receives the CMK blob, the MA needs to re-wrap the CMK blob by decrypting with its own private key and encrypting it with the public key of the destination TPM. This is done by using the MigrateKey command before the CMK blob is further sent to the TPM destination.

6. MigrateKey

- When arriving at the destination TPM, the CMK blob has to be converted. However, to be able to do this, the destination TPM needs to be certain that this migration is authorized by the MSA. The MSA then sends the restricTicket to the destination TPM, which is once again verified by the sigTicket. When this step is performed, the destination TPM wraps the CMK with its parent key.

7. CMK_CreateTickets
8. CMK_ConvertMigration

- The CMK can now be used using the TPM_LoadKey command.

Chapter 5

Duplication in TPM 2.0

Similar to the previous chapter, this chapter will be describing the commands that is needed to duplicate an object from one TPM to another.

5.1 Overview

The main difference between TPM 1.2 and 2.0 is that the migration is no longer called migration, it is now called duplication. Also, the MA and the MSA are no longer needed because in TPM 2.0 there is no CMK. To achieve restrictions on how to handle keys, the main focus in TPM 2.0 is something called *policies*, as described in section 2.3.4. However, in one of the policies that will be used during the duplication, an authority will be needed. More about this will be described further on in this chapter.

5.2 Commands for duplication

In this section the command sequel for a successful duplication [13]will be described. As mentioned in section 2.3.4, the authorization method used during a duplication of an object is the policy method. The policies that will be used in this context are the *TPM2_PolicyAuthorize* and the *TPM2_PolicyDuplicationSelect*.

5.2.1 TPM2_CreatePrimary

- *Input*
    - Data
    - Algorithm
    - AlgorithmID and Digest
    - PCR
- *Output*
    - public portion of the object
    - creationData

- creationHash
- creationTicket
- objectName

When taking ownership over a TPM 2.0 the SPS is created as described in 2.3.2. To create a child key for this hierarchy, the TPM2_CreatePrimary command is used. It creates a primary object, a SRK, which is stored under the SPS. The input parameters for this command is the data, which is an empty buffer if an asymmetric object is being created, the algorithm, for what kind of object is to be created, and PCR value, that will be used in the creation data.

The output of the command is public key of the object, the creationData which includes e.g., the parentName and the parentAlgortihm, the hash of the creationData and the name of the created object. There is one more output, which is the creationTicket. This is to validate that the creation data was produced by the TPM.

This command is very similar to the ordinary CreateKey command, the only difference is that it will **not** return the private part of the key. This is to ensure that the private part of a primary object never leaves the TPM. Also, as a primary object, some attributes has to be set. For example, FixedTPM and FixedParent attributes has to be set to *1*, which means that the key has a definite parent, hence it cannot be duplicated.

5.2.2 TPM2_PolicyAuthorize

- *Input*
  - authPolicy$_1$
  - policyRef
  - signKey
  - checkTicket
- *Output*
  - Response size
  - Response code

The PolicyAuthorize command is used to change an authPolicy value with the help of an authority. The authPolicy value is given by the digest value, as mentioned in section 2.3.4. At the beginning of the session, this digest value is zero.

$$authPolicy_1 = policySession \to policyDigest$$

This command should be executed before the object is created, which means that the digest value included into the object could be changed. However, this command can only change the digest that is created before the PolicyAuthorize command is executed. This means that if there are some policies that are executed before the PolicyAuthorize command, P_{PA}, the digest that was created during the execution of these will be changed. As seen in Figure 5.1, the PolicyAuthorize command can change the digest that has been generated from P_1, P_2 and P_3. Thus, the policies executed after P_{PA}, which are P_4 and P_5, will not be modifiable.

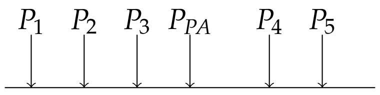

Figure 5.1: PolicyAuthorize process

After the authPolicy$_1$ has been set, the digest is added with a value called *policyRef*, a policy qualifier, which is generated and used by the TPM as its own secret.

$$aHash = HMAC(authPolicy_1|policyRef)$$

The HMAC is calculated depending on which algorithm is used within the hierarchy and what kind of key is to be generated. With the help of the authority, this value can be switched with the name of the sign key of the authority. This is done by signing the *aHash* with the signing key. The result of this action is that the new authPolicy ($authPolicy_2$) will be the name of the signing key.

$$authPolicy_2 = signKey_{name} = (aHash)_{sign}$$

The outcome of the new authPolicy will always be the same as long as the same authority is used for the modification.

To validate this signing process, the TPM2_VerifySignature command is used. This will generate a ticket that will be used as the authorization. Instead of reproducing the digest and its verification, the ticket is provided.

$$Ticket = HMAC(proof|(TPM\_ST\_VERIFIED||aHash||signKey \rightarrow name))$$

As mentioned, the PolicyAuthorize command is executed before the key is created to be able to change the authPolicy value when having the knowledge of which destination the key is being duplicated to.

5.2.3 TPM2_Create

- *Input*
  - Data (empty buffer for asymmetric objects)
  - The algorithm for what kind of object is to be created
  - AlgorithmID and Digest
  - PCR that will be used in creation data
- *Output*
  - private portion of the object
  - public portion of the object
  - creationData
  - creationHash
  - creationTicket
  - objectName

The TPM2_Create command creates the duplication object. The input parameters for this is the data, which is an empty buffer for asymmetric objects, the algorithm, for what kind of object is to be created, and the PRC value that will be used in the creation data. This value is set to zero.

The output parameters is the encrypted private key, the public key, the creationData which includes e.g., the parentName and the parentAlgorithm, the hash of the created data, the name of the created object and the creationTicket.

When the object is created, the symmetric encryption is used to encrypt the sensitive area of the object. Hence, it is encrypted with the symmetric key which is retrieved from the parent key and the parameters of the created object. This command is very similar to the previous command, except as mentioned before, in this command the private portion is returned which the previously command does not. Even though the private portion is returned, it never leaves the TPM decrypted.

5.2.4 TPM2_LoadExternal

- *Input*
  - Sensitive portion of the object (optional)
  - The public area
  - Handle type
- *Output*
  - Handle that refers to the object
  - Name for the entity type

The LoadExternal command is used to load the public area of the key to which the newly created key is to be migrated. The input for this command is the sensitive portion of the object, which is optional, the public key and the handle type depending on which key from which hierarchy is to be loaded.

The name of the new parent key and also the handle for the loaded object is then returned. Also, external objects are temporary objects which means that the next time the TPM is reset, the loaded objects will be deleted.

5.2.5 TPM2_PolicyDuplicationSelect

- *Input*
  - objectName
  - newParentName
- *Output*
  - respond size
  - response code

The PolicyDuplicationSelect command affects the policy value and is used to define a destination for the key being duplicated. Thus, the key will have a fixed destination which will not be modifiable.

Before creating the key, the PolicyAuthorize command should be executed as described in the previous section, 5.2.2. This will set the authPolicy to the key name of the signing key of the authority. After a successful generation of the object, the duplication process is started. The first thing the TPM has to do is to execute the PolicyDuplicationSelect command.

$$nameHash = HMAC(ObjectName||newParentName)$$

$$policyDigest_{new} = HMAC(policyDigest_{old}|| \\ TPM\_CC\_PolicyDuplicationSelect||newParentName||inludeObject)$$

The key is set to a specific destination. When the destination is set, the PolicyAuthorize command is once again used where the authority has to approve the destination before it can sign the new authPolicy with its signing key. Thus, the object cannot be duplicated to an unauthorized TPM. When the authority has approven the destination, the same sign key that was used during the generation of the object has to be used. After the signing, the new authPolicy will be the name of the sign key which means that the same digest value will be accomplished and the duplication can be pursued.

5.2.6 TPM2_Duplicate

- *Input*
  - parent key
  - inner wrapping key
  - symmetric algorithm
- *Output*
  - symmetric encryption for inner wrapper
  - encrypted private area
  - seed of new parent

This command performs the actual duplication of the object. For this command to be executed, the public key of the new parent, the inner wrapping key and the symmetric algorithm used for the inner wrapper have to be given. The inner wrapping is optional, and can be chosen by the user or created by the TPM.

The result of the command is a new sensitive structure that is encrypted by the new parent. The importance with this command is that it sets the command-Code value of the policy context to TPM_CC_Duplicate, which in turn will enable the DUP role, which is an authorization role, and the duplication can be made for the object.

The policy is likely to include cpHash in order to restrict *where* the duplication can occur.

5.2.7 TPM2_Import

- *Input*
  - inner wrapping key
  - public part of the object
  - duplication object
  - symmetric key
  - algorithm for inner wrapping key
- *Output*
  - out private

The import command allows the object to be encrypted using the symmetric encryption values of a storage key. The input for this command is the public part of the object, the duplication object, which is the sensitive area and the symmetric key used to encrypt the duplication object. Also, as seen among the input fields, the duplication object may have an inner wrapping key. However, because the duplication is monitored by an authority, this will not be needed.

The output of this command is the encrypted sensitive data of the duplication object.

5.3 Duplication alternatives

The duplication method is based on the same basis as the migration method in Chapter 3, where it includes only two parties, the sender TPM, which the key is being duplicated from, and destination TPM, which the key is being duplicated to. In this section, two duplication alternatives will be presented, the first one involving an authority, and the latter being a direct duplication alternative.

Duplicating with the help of an authority

This duplication alternative involves an authority and the PolicyAuthorize command, which means that the destination of the duplication key can be modified.

- To be able to create a duplication key, a parent key is needed. After the creation of the parent key, the authPolicy is to be set to the name of the signing key of the authority. Then, the duplication key is generated. When the destination later on is known, the public key of the new parent is loaded into the TPM and the PolicyDuplicationSelect command is executed. Now that the destination is set, the authority has to approve it by signing the new digest. This is done with the same command as previously used, the PolicyAuthorize command. The authPolicy will now be the name of the signing key of the authority once again and the authPolicy can be updated. The last command in this process is the Duplicate command which will send the duplication key to the destination TPM.

    1. TPM2_CreatePrimary
    2. TPM2_PolicyAuthorize
    3. TPM2_Create
    4. TPM2_LoadExternal
    5. TPM2_PolicyDuplicationSelect
    6. TPM2_PolicyAuthorize
    7. TPM2_Duplicate

- When arriving at the destination, the key has to be imported to be able to be used.

    9. TPM2_Import

Duplicating directly between TPMs

Comparing to previous duplication alternative, this alternative does not involve any authority and the destination is set before the generation of the duplication key. This means that the duplication destination can *not* be modified.

- Similar to previous duplication alternative, the parent key needs first to be created. The next step is to create the duplication key, however, before this can be done, the destination of the key should be selected. This is done with the PolicyDuplicationSelect command. The execution of this command will give an authPolicy value in return which will be used during the generation of the duplication key. After the creation of the duplication key, the public key of the destination is to be loaded so that the Duplicate command can be executed and the duplication key can be duplicated.

  1. TPM2_CreatePrimary
  2. TPM2_PolicyDuplicationSelect
  3. TPM2_Create
  4. TPM2_LoadExternal
  5. TPM2_Duplicate

- When arriving at the destination, the key has to be imported to be able to be used.

  6. TPM2_Import

Chapter 6

Migration methods for TPM 1.2 in telecommunication nodes

For a successful use of the migration functions in the telecommunication nodes, three parties are being involved. First we have the CU Factory which is also known as the board factory. This entity is responsible for the creation of the boards (CUs) which contain the TPMs and the first configuration of each board. This means that the CU Factory will take ownership of each board. After the configuration is done, the CU Factory sends the CUs to the HAS Factory, which is responsible for the creation of the node and the node blob. For example, this can be seen as being created in the base station factory. The third and last part is the customer which receives this node blob with the TPMs installed.

As seen in Figure 6.1, the boards that are made in the CU factory are sent both to the HAS factory and to the Customer. This is because only two boards are being used for the first installation in the HAS factory, and the rest are sent to the customer as spare CUs in case one or both of the CUs in the HAS break down.

The different parties will be involved in all of the use cases presented in this chapter.

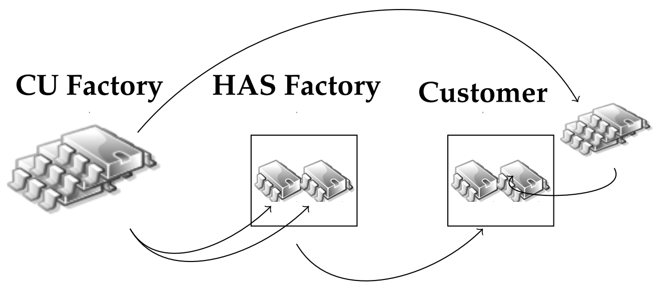

Figure 6.1: The involved parties.

6.1 MSA Setup

With respect to the use case of HAS and CU, the MSA is maintained by the CU Factory. Such a MSA can also be realized as a server. The MSA is responsible for the main parts of the migration in all the use cases and is used for several purposes.

As described in section 4.1.1, the MSA is used for the creation of the restrictTickets. However, in the use cases that will be presented in this chapter the MSA will be much more than just an authority that creates the tickets for the migration. The MSA will also be involved in the configuration of each CU containing the TPMs.

- One of the main tasks involving the MSA, is the creation of the generic CMKs that will be migrated to each TPM board created in the CU Factory. The two CMKs that will be created are the CMK_{f_s} and the CMK_{f_b}. The f_s stands for a *fixed storage key* and the f_b stands for a *fixed binding key*. Because of the unique SRK for each TPM, the MSA has to "re-wrap" the key blobs and create the tickets belonging to it. The key hierarchy, containing the generic keys, after the involvement of the MSA can be seen in Figure 6.2.

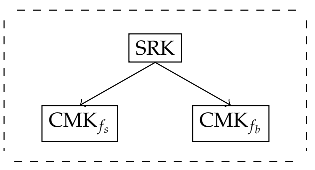

Figure 6.2: Overview of key hierarchy in a generic board

- MSA also creates the certificate for each TPM. This is the certificate that verifies that the TPM is a valid TPM and is used to ensure that the customer is using the correct one. The TPM creates a sign key that will be used in the creation of a CSR, *Certificate Signing Request*. This request is then sent to the MSA, which will create the certificate and approve the TPM by adding the SRK and the certificate to a PKI, *Public Key Infrastructure* tree. The PKI tree will be used during the personalization of the TPM to verify that the TPM is approved by the CU Factory through its MSA. Thus, the certificate has to be given to the MSA to prove validation of the TPM.

- As mentioned, the MSA can also be realized as a server, this is because the MSA will store the public portion of the SRK of each manufactured TPM in the CU Factory in a database. A white list will be created for these SRKs and checked each time the personalization of a board is to be carried out. More details about this will be described in section 6.2.2.

6.2 Migration with a MSA/Server

In this subsection, three use cases will be presented. The first case, *Case A*, is based on creating a generic board which can be used by all customers of the manufacturer. This is done by creating two *fixed* CMKs, CMK_{f_s} and CMK_{f_b}, that will be migrated into each TPM board, as described in previous section. The second and third case, *Case B* and *Case C*, are both based on a customer specific solution of the board for each customer where Case A is used as foundation. This is done by creating a *customer specific* key, CMK_s. The difference between these two cases will be discussed further on in this section.

6.2.1 Case A

In this case, a generic board is to be created. The first step is to initialize the TPMs which is done in the CU Factory. By taking ownership, the SRK is created and will later be used as the parent key for the generic key. The CU Factory will also create a sign key that will be used in the creation of a CSR. This request is sent together with the SRK to the MSA/Server. Hereinafter, the MSA who already created the fixed keys, CMK_{f_s} and CMK_{f_b}, in advance returns this to the TPM on the CU created in the CU factory.

Aside from migrating the fixed keys to the CU, the MSA also creates a certificate from the CSR. This will declare that the TPM is created by the CU manufacture and will always be checked before migration.

When the CUs have been manufactured, two boards are sent to the HAS Factory, where they are installed into the newly created node and some are sent to the customer for backup purposes.

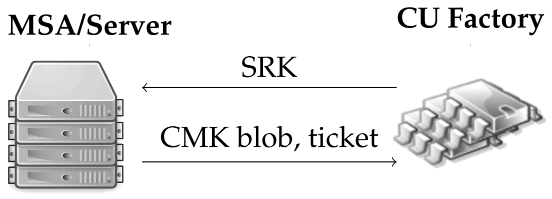

Figure 6.3: Overview of CMK migration in Case A.

Even though both of the fixed keys are in the TPM, thus both keys are available for the customer, this solution will only provide the use of one of the keys, the CMK_{f_b}. This generic binding key is used by the customer to store the sensitive information in the node. However, this also means that all customers of the board manufacturer that choose this solution will share the same binding key.

The properties of this solution are

- Offline

- Generic

The offline property means that the boards can be switched out and used without contacting the MSA/Server, this is because all boards are generic.

The steps and the command that are executed during this solution can be seen in Figure 6.4.

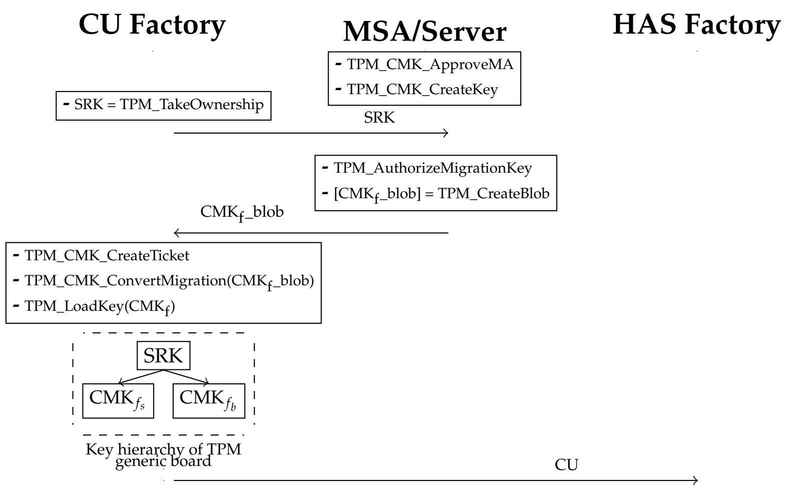
CU Factory
MSA/Server
HAS Factory
- TPM_CMK_ApproveMA
- TPM_CMK_CreateKey
- SRK = TPM_TakeOwnership
SRK
- TPM_AuthorizeMigrationKey
- [CMK$_f$_blob] = TPM_CreateBlob
CMK$_f$_blob
- TPM_CMK_CreateTicket
- TPM_CMK_ConvertMigration(CMK$_f$_blob)
- TPM_LoadKey(CMK$_f$)
SRK
CMK$_{f_s}$
CMK$_{f_b}$
Key hierarchy of TPM generic board
CU

Figure 6.4: Overview of the preparation steps for CMK migration in Case A.

6.2.2 Case B

As mentioned at the beginning of this chapter, Case B is built on a foundation that is constructed during Case A. This means that the generic board is at first generated in the CU Factory with the help of the MSA/Server before proceeding with the personalization of the board which will be done in the HAS Factory. The reason for personalizing the board in the HAS Factory and not in any of the earlier steps, is because during the creation of the node blob the information about which customer this node blob is intended to belong to, has become known.

An overview over the steps of Case B can be seen in Figure 6.5. During the initialization, the SRK of the TPM boards created in the CU Factory are stored in a white list in the MSA/Server's database. When the TPMs later arrive at the customer, at the HAS Factory to be specific, the SRKs are sent to the MSA/Server which checks if the SRKs are in the approved SRK database, if so, the MSA/Server creates a new CMK that will be specific, CMK$_s$, for that node owned by the customer.

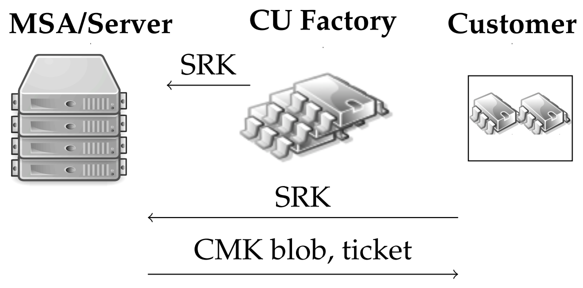

Figure 6.5: Overview of CMK migration in Case B.

After the creation of the CMK$_S$, it is sent to the customer, encrypted with the public key of the SRK and with the tickets that are needed for the migration.

In case one or both CU/-s break, the customer puts one of the spare CUs containing a TPM into the node for installation. The customer has then once again to contact the MSA/Server for the restrictTickets, which are as described in section 4.1.2, needed for the migration of the key. By sending the new SRK of the newly inserted TPM to the MSA/Server, the MSA/Server can then return the saved CMK$_S$ for this particular node blob and the restrictTickets needed for the import of this key.

As noticed, the fixed keys are not used in this solution. A specific key is created for each customer with the help of the MSA/Server, thus each customer gets a personalized board. Also, the migration of a CMK involves the endorsement of the MSA/Server because of the tickets, and due to the need of the communication with the MSA/Server, this becomes an online solution.

The properties of this solution are

- Personalized
- Operator specific

The details of this solution can be seen in Figure 6.6.

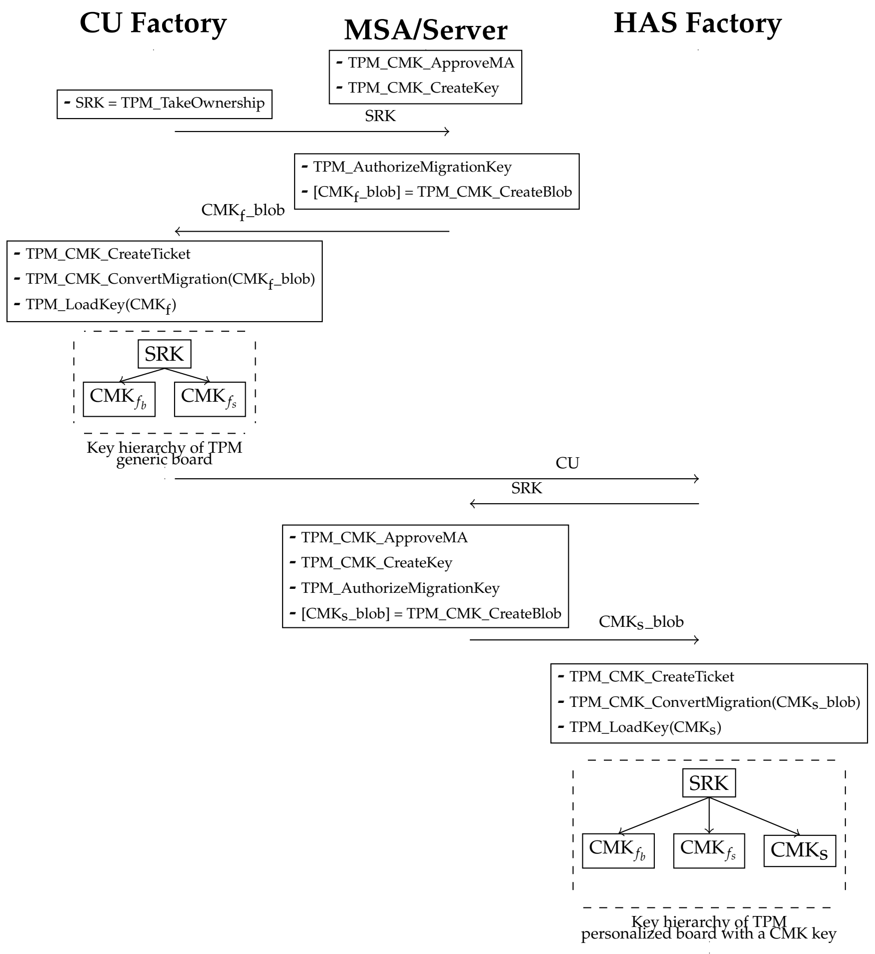
CU Factory
MSA/Server
HAS Factory
- TPM_CMK_ApproveMA
- TPM_CMK_CreateKey
- SRK = TPM_TakeOwnership
SRK
- TPM_AuthorizeMigrationKey
- [CMKf_blob] = TPM_CMK_CreateBlob
CMKf_blob
- TPM_CMK_CreateTicket
- TPM_CMK_ConvertMigration(CMKf_blob)
- TPM_LoadKey(CMKf)
SRK
CMKfb
CMKfs
Key hierarchy of TPM generic board
CU
SRK
- TPM_CMK_ApproveMA
- TPM_CMK_CreateKey
- TPM_AuthorizeMigrationKey
- [CMKS_blob] = TPM_CMK_CreateBlob
CMKS_blob
- TPM_CMK_CreateTicket
- TPM_CMK_ConvertMigration(CMKS_blob)
- TPM_LoadKey(CMKS)
SRK
CMKfb
CMKfs
CMKS
Key hierarchy of TPM personalized board with a CMK key

Figure 6.6: Overview of the preparation steps for CMK migration in Case B

6.2.3 Case C

Although Case C is a very similar to Case B, there is a small difference in how the customer configuration is made. The initialization though, is made in exactly the same way as in the previous cases where the CU factory creates the boards, creates a sign key and receives the CMK$_{f_s}$ and the CMK$_{f_b}$ from the MSA/Server. However, when the board arrives at the HAS Factory a new hierarchy level is added with the help of one of the fixed keys given.

Instead of creating a customer specific CMK, as in Case B, an ordinary migration key, K$_\mathrm{P}$, with

- keyUsage value as *storage*
- keyFlag value as *migrate*
- CMK$_{f_s}$ as the parent key

is generated. Having the customer specific key as a child key, see Figure 6.7, to the generic storage key would mean that a board can easily be switched out when broken by just replacing it with the new one. The reason for making this possible is because every board has the generic keys, which means that having the parent key will instantly give access to the child key, which in this case is the CMK$_{f_s}$ and the customer specific key respectivly. Even though all customers have access to the parent key, no one except the operator itself, and the board owner which is the CU Factory, will have access to the customer specific key. Only the ones knowing the usage secret will be able to use it.

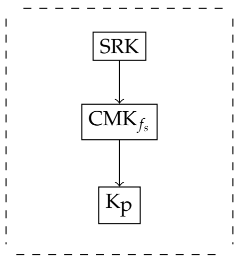

Figure 6.7: Overview of the hierarchy in Case C

Also, one benefit from adding a key for each customer, is that no certifications are needed to be checked when a board breaks and a new one is inserted, the operator can use it instantly.

This solution is an offline solution which means that none of the three parties are involved in case of a TPM board malfunctions. However, this also means that the HAS Factory has to be a trustworthy party and the customers has to believe that the HAS factory will not record the migration secret after the creation of the key. This is to insure that the key will never be migrated to an unauthorized authority.

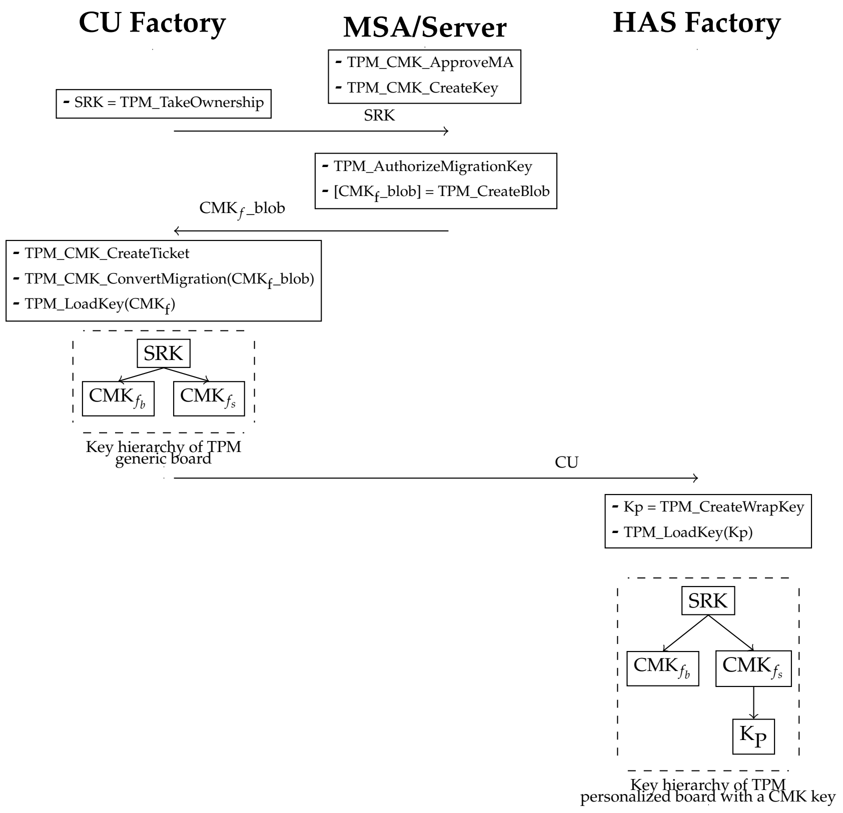

Figure 6.8: Overview of the preparation steps for CMK migration in Case C

As seen in Figure 6.8, the steps done in the MSA/Server and the CU factory are standard, which means that they are always done. However, the last step that is done in the HAS factory can be chosen, depending on if the customer wants a customer specific board with an online or an offline solution. Thus, an offline solution would mean that the customer specification would take place in the HAS Factory.

The properties of this solution are

- Operator specific
- Offline

6.3 Migrating offline with a node specific MSA

In this section, an other offline solution will be presented. As in previous case, three parties are involved; the MSA/Server, CU Factory and the HAS Factory. However, one more part will be added in this use case and it is called the Node Specific MSA. The node specific MSA will play an important role in the HAS Factory and will be the crucial part of this solution.

To begin with, the TPMs are mounted and initialized in the CU Factory, which will take ownership over the TPMs and create a sign key that will be used for the creation of the certificate. The SRK and the CSR are then sent to the MSA/Server which will create the certificate and approve the TPMs by adding the SRK and the certificate to a PKI tree. After the initialization is done, CUs are sent to the customer as the backup CU and to the HAS Factory for node initialization. So far, the use case has been the same as in section 6.2.1.

In the next step, which is the node initialization in the HAS factory, the node specific MSA is involved. The node specific MSA will be created as a software and run in a TEE, in the node. TEE is used so that the private keys of the node specific MSA, which will be encrypting the CMK, will be protected. In this way, the node specific MSA can "play" the MSA/Server and perform all the steps that are needed for a successful creation and migration of the CMK. The steps that are being performed by the node specific MSA can be seen in Figure 6.9 under HAS Factory.

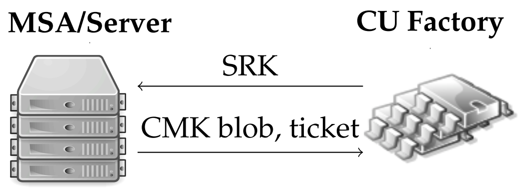

Figure 6.9: Overview of CMK migration in Case A.

This means that when one or both TPM/-s break, the customer only has to switch it out with the spare board/-s and the node will configure itself, which contributes to being an offline solution. Another positive feature of this solution is that each node will in a way be personalized, due to the creation of a new CMK_S in each and every node.

However, when using the node specific MSA it has to be trusted enough to store the customer specific key.

This may be solved by letting the TPM create its own personalized CMK in the HAS factory, which would mean that the CMK_S would be encrypted with the public key of the TPMs SRK. This key is then to be migrated so the TPM has to create a key blob that will be encrypted with the public key of the destination which in this case would be the node specific MSA. Before encrypting the key blob with the MSA's public key, the CMK_S can be encrypted with a random **r**. This r can be kept secret and only sent to the customer, which means that even though the node specific MSA has the power to decrypt it at any time, the private part of the key will not be revealed. So when a TPM breaks, the customer puts in a backup TPM, receives the key blob from the node specific MSA and decrypts it with the r. This would mean that if the inserter does not have the knowledge of what r is, the key blob will never be decrypted and the private key will stay secret. This also means that MSA does not have to be fully trusted, but it does mean that the software performing these commands do and also that the customer has to keep track on which r belongs to which node.

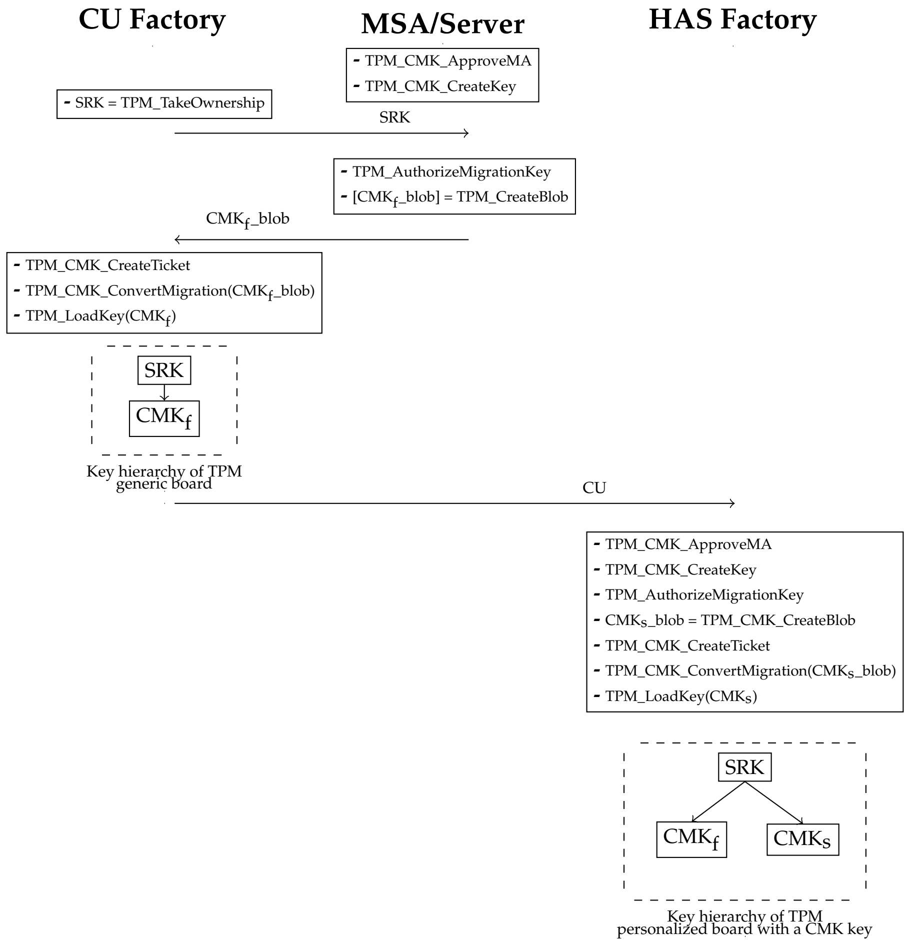
CU Factory
MSA/Server
HAS Factory
- TPM_CMK_ApproveMA
- TPM_CMK_CreateKey
- SRK = TPM_TakeOwnership
SRK
- TPM_AuthorizeMigrationKey
- [CMKf_blob] = TPM_CreateBlob
CMKf_blob
- TPM_CMK_CreateTicket
- TPM_CMK_ConvertMigration(CMKf_blob)
- TPM_LoadKey(CMKf)
SRK
CMKf
Key hierarchy of TPM generic board
CU
- TPM_CMK_ApproveMA
- TPM_CMK_CreateKey
- TPM_AuthorizeMigrationKey
- CMKS_blob = TPM_CMK_CreateBlob
- TPM_CMK_CreateTicket
- TPM_CMK_ConvertMigration(CMKS_blob)
- TPM_LoadKey(CMKS)
SRK
CMKf
CMKS
Key hierarchy of TPM personalized board with a CMK key

Figure 6.10: Overview of the CMK migration method in the HAS Factory

Chapter 7

Duplications methods for TPM 2.0 in telecommunication nodes

Because of the similarities between TPM 1.2 and TPM 2.0, the use cases that were created for TPM 1.2 in Chapter 6, will be recreated for TPM 2.0 in this chapter. The use cases will still depend on three parties, the CU factory, the HAS factory and the Customer, and their course of action will be the same.

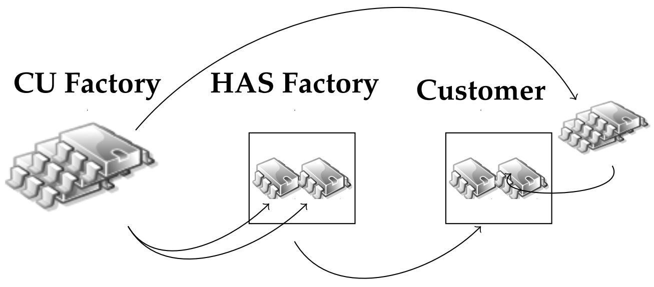

Figure 7.1: The involved parties.

However, during the duplication there is no MSA/Server that is needed, but there is something very similar to it. The MSA will in these cases be called an *authority*, and instead of dealing with tickets, the authority is needed for the policies as described in Chapter 5.

Also, because of the two cryptographic engines offered in TPM 2.0, symmetric and asymmetric keys can be created. This means that a symmetric key can be used for the encryption and decryption of the sensitive information, which would increase the speed of the encryption and decryption receptively.

7.1 Duplication with an authority

The three use cases will be presented in this subsection. The first case, *Case A*, is based on creating a generic board containing a fixed storage key, K_{f_s}, and a fixed decrypt key, K_{f_d}, which can be used by all customers of the manufacturer, as in section 6.2.1. *Case B* and *Case C* are then based on Case A, but are also more developed solutions which handle personalization by creating a customer specific key.

7.1.1 Case A

The TPM boards have to be created and initialized which is done in the CU factory. When completed, the three TPM hierarchies, which include the SPS that is needed for this use case, are established. SPS is then used as a parent key to create a SRK, also known as a primary object, which, in turn, is used as the parent key for the generic keys.

Also, as in 6.2.1, a signing key is generated. Which will be used to create a CSR, which in turn will be used as an identification of that the TPM boards is created in the CU Factory.

The SRK and the CSR is sent to the authority, and in return the key blob is given, see Figure 7.2.

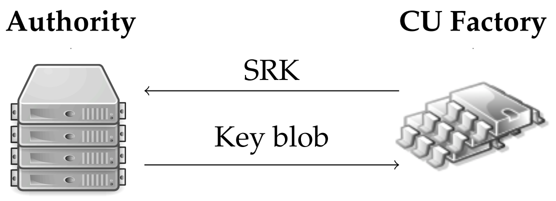

Figure 7.2: Overview of duplication in Case A

The generic key creation is realized in multiple steps. First of all, when creating the key, the authPolicy has to be given which is done by the use of policies. As described in section 2.3.4, the execution of the different policies will establish a digest that will be used to make sure that the object will not be duplicated to an unauthorized authority. Also, before creating the key the fixedParent value *must* be set to "0", otherwise the key cannot be duplicated to another parent and an error will occur. This initialization is done for both generic keys created, K_{f_s} and K_{f_d}, respectively. K_{f_s} stands for a *fixed* manufacture storage key and will be an asymmetric key, while K_{f_d} stands for a *fixed* manufacture decrypt key and will be a symmetric key. After the creation of the generic keys, they are duplicated to the each individual board. This is done by changing the authPolicy digest value, which now includes which destination the keys are meant for, and signing them with the key of the authority.

After the TPM initialization is completed, the generic boards are sent to the HAS factory and to the customers.

Very much alike the solution in section 6.2, a generic solution as in Case A will be provided, see Figure 7.3.

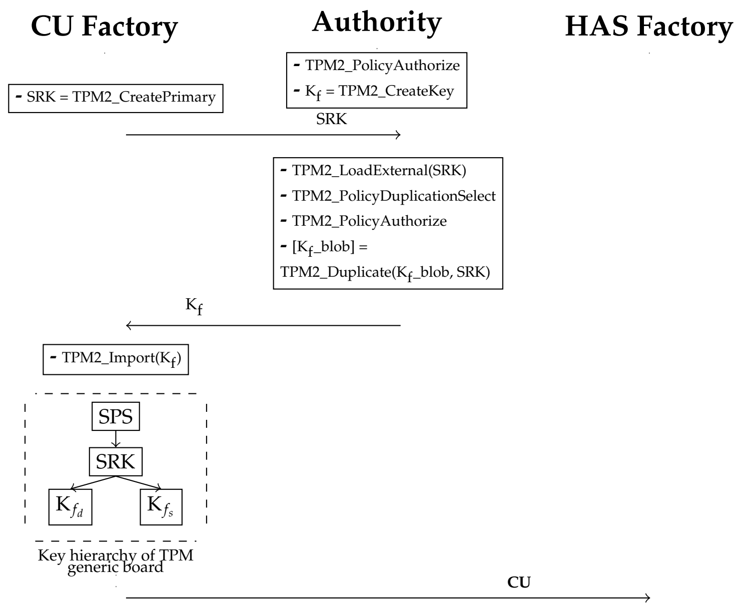

Figure 7.3: Overview of the preparation steps for Case A

The properties of this solution are

- Offline
- Generic

The offline and the generic properties provides simplicity for the customer in case that a TPM board malfunctions by just changing the failed board with a new one.

7.1.2 Case B

In Case B, the personalization will be performed when the generic boards arrive at the HAS Factory. At this stage, the details of which customer the board is meant for is known. To make it personalized, the customer has to send the SRK to the authority, in similarity to Case B in section 6.2.2. The authority then audits with its database to make sure the TPM has been created in the CU Factory.

When the audit is performed, the authority starts the process of generating a symmetric decrypt key. First, the authority has to execute the policy command PolicyAuthorize. This is because the destination of the key can vary depending

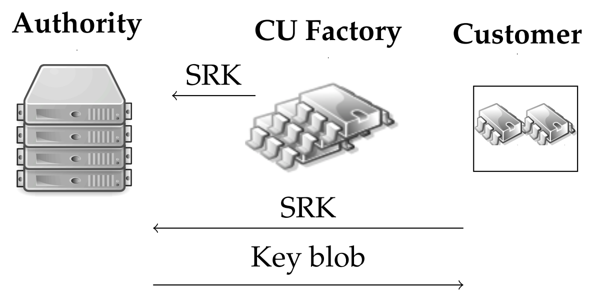

Figure 7.4: Overview of duplication in Case B

on which TPM the key is to be migrated to, which means that the authPolicy digest will need to change. Also, the fixedParent value is set to "0". With this value set, the object will not be able to be duplicated to an unauthorized authority. One more thing worth mentioning is that the parent key of the object being created will be the SRK, which is stored within the borders of the TPM. Thus, the SRK cannot be duplicated, which also leads to that the object *cannot* in any way be transfered in an unauthorized way.

After the key has been generated, the DuplicationSelect command is used to decide the destination and with that the authority also has to create a new digest and update this within the key. After the authority has signed and approved of the new authPolicy, the personalized key is both saved at the authority side and returned to the customer. The customer then imports it into its board.

In case the board malfunctions, the customer has to contact once again the authority to get the personalized key. The customer sends the SRK of the new board and the server changes the authPolicy once again as described in 5.2.2. The key blob is then returned to the customer.

The properties of this solution are

- Personalized
- Operator specific

The steps and commands that are needed to be executed for this solution can be seen in Figure 7.5.

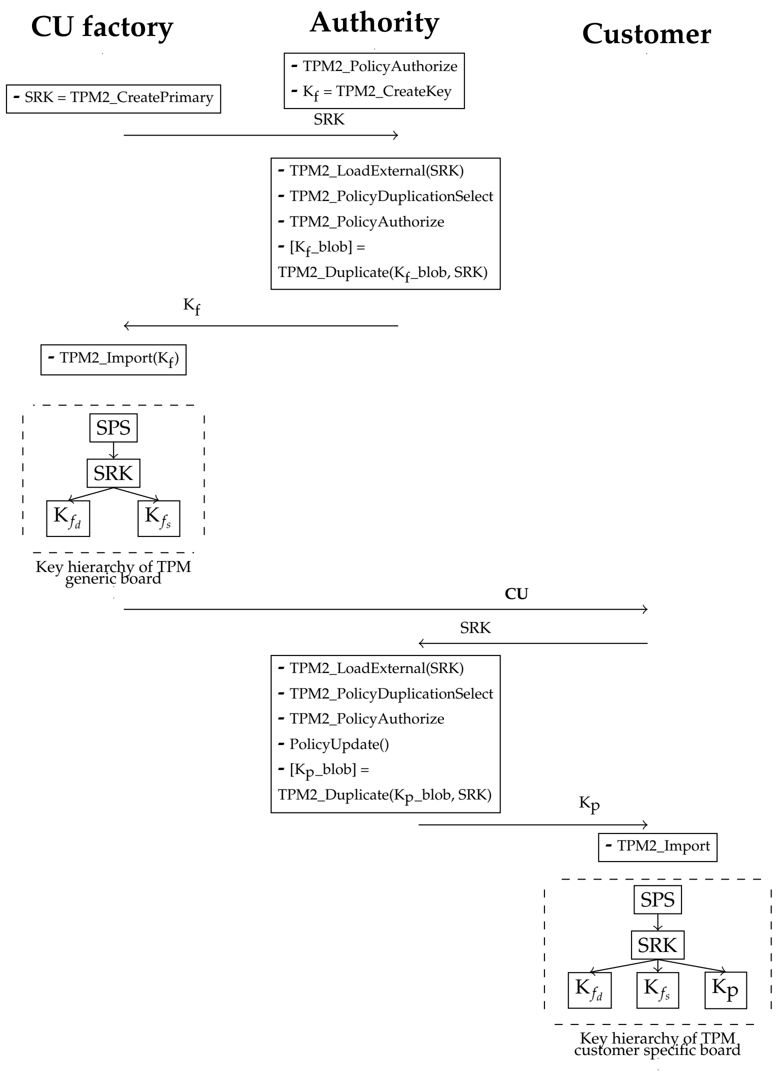

Figure 7.5: Overview of the preparation steps for Case B

7.1.3 Case C

Case C is a customer specific solution that is literally built on the solution of Case A. This is because Case C will take advantage of the key hierarchy that is given in Case A, see Figure 7.6. That is why the steps that are made in section 7.1.1 are important to follow.

When the initialization of the TPMs is done, the boards are sent to the HAS Factory for the creation of the node blob. This is where the operator specific key, K_s is to be created. This key will be a symmetric key and is created with the Create command, but first it has to be ensured that this key will not be able to be duplicated to an unauthorized authority. This is done by setting

- the PolicyDuplicationSelect command
- the public key of K_{f_s} as the parent key
- the fixedParent value is set to "1"
- the key type value as *decrypt*

Implicitly, this means that the key created will not be a duplication key. However, because the parent key is a duplication key, the keys belonging to it will also be duplicated. Thus, having access to the K_{f_s} key results in having access to the operator specific key, K_S.

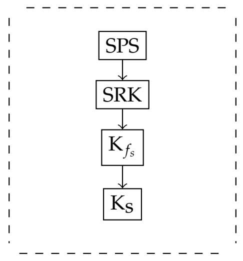

Figure 7.6: Overview of the hierarchy after the customer specific key creation

As seen in Figure 7.6, the key hierarchy will be constructed of four keys, where three of them belong to the generic TPM. With the help of this, the customer will be able to reach their own customer specific key at anytime. If one or both of the boards break down in the node, the failed TPMs are switched out with new ones by the customer and these will take over their tasks. As mentioned, the CU factory will send generics boards directly to the customer for these purposes.

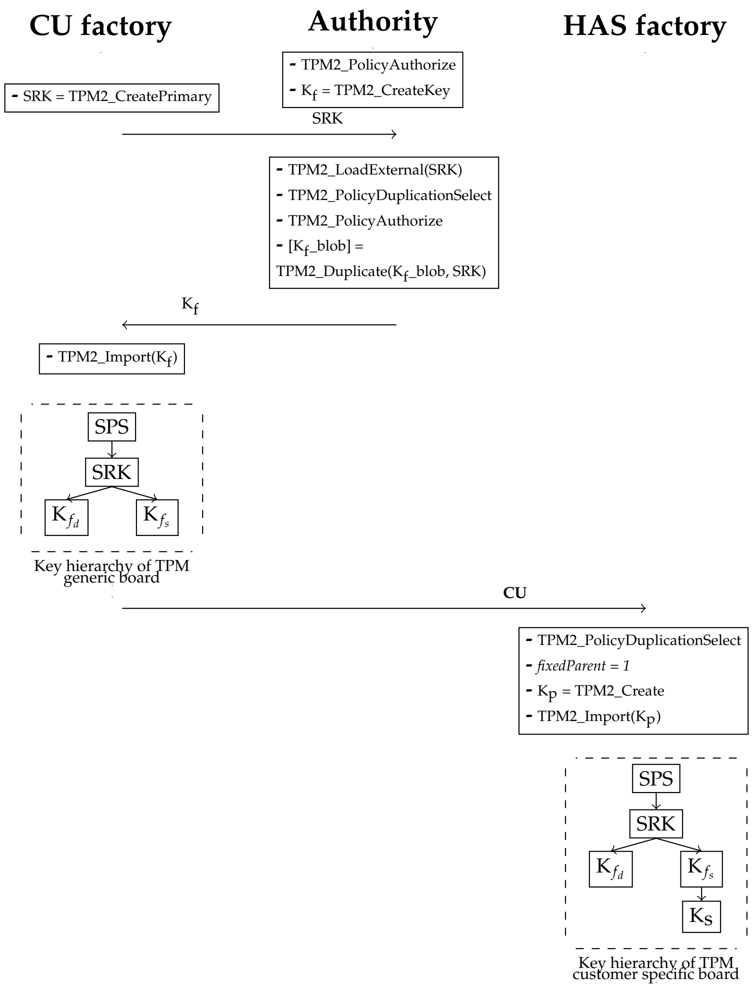

Figure 7.7: Overview of the preparation steps for Case C

As seen in Figure 7.7, this method is very much alike the method that is done in 6.2.3. The main difference between these two solutions is that a CMK is used in the first method, while a ordinary duplication key is used in the latter solution.

The properties of this solution are

- Offline
- Operator specific

In this case, the boards are operator specific and also provides an offline solution, which means that an authority does not have to be involved. The operator specific key will be below the K_f key which means that it will be reachable from every generic board. However, the HAS Factory has to be trusted.

Chapter 8

Summary of Results

In this chapter, the main differences between the use cases when deploying TPM 1.2 and TPM 2.0 will be discussed. Because of the many similarities between the uses cases in version 1.2 and 2.0, the reflection in this chapter will not set these two versions against each other. This chapter will be more of a discussion of the differences between the use cases themselves, i.e., Case A, Case B and Case C.

The advantages and disadvantages will be mentioned and some feedback is given on what could be changed to facilitate the migration/duplication.

With the help of Table 8.1, Table 8.2 and the properties of each solution, the most efficient versus inefficient solution for the three involved elements will be chosen.

	Method 1			Method 2
	Case A	Case B	Case C	
Personalized	0	1	0	1
Operator specific	0	1	1	1
Offline	1	0	1	1
No HAS Factory trust	1	0	0	0

Table 8.1: Differences in use methods in TPM 1.2

	Case A	Case B	Case C
Personalized	0	1	0
Operator specific	0	1	1
Offline	1	0	1
No HAS Factory trust	1	0	0

Table 8.2: Differences in use methods in TPM 2.0

The different properties provided in Table 8.1 and Table 8.2 represents;

Personalized
a customer specific key is created with the help of an authority

Operator specific
a customer specific key is created

Offline
an offline solution is provided

No HAS Factory trust
the node factory, which is the creator of the HAS, is not needed to be trusted

8.1 Use case overview

Even though the solutions described in Chapter 6 and 7 are very similar, they have their differences too. As seen in the tables, no method can completely fulfill the requirements.

Method 1

- Comparing these cases, the most inefficient use case is Case A of Method 1. Case A does not include any operator specific or personalization solution, only a solution that provides a common fixed key. This means that all customers will be using the same key for storing their sensitive information. The solution is not optimal from a customers point of view. However, at the same time it is an easy solution if a CU in a HAS fails.

- Case B is a solution that is more customer specific. That is, each customer has its own key to encrypt their sensitive information. In addition to this, the key will also be personalized which means that it involves an authorized authority. Thus, Case B is a more secure solution from a customer's point of view. However, because this solution involves the MSA/Server or an authority, this will mean that it is an online solution. This may be more demanding from a customer's point of view as they occasionally must contact the server.

- The latter solution, Case C, is a solution that is most suitable from a customer's point of view. This is because the solution is both customer specific and offline. Which means that if a board fails, the customer can easily switch it out with a new one, but still have it customer specific which will keep its sensitive information private. The only downside with this solution is the question if the HAS Factory is to be trusted.

Method 2

- The node specific solution provides a personalized and operator specific key. Also, using this solution would mean that each node would have its own key to protect the sensitive information of the node, which also would mean a higher security, which is an advantage. It is also an offline solution which facilitates the board switch in case a board fails. However, this solution also has its disadvantages. First, the HAS Factory has to be a trusted part. Second, which is the major problem with this solution, is that the customer has to assume that the TEE is capable in protecting a private key. Then one might wonder why not use the TEE only. Thus, the customer actually gained nothing by using a TPM.

8.1.1 Offline solution

In the offline methods, the HAS factory has a crucial role in the different use cases. This means that the HAS factory needs to be trusted.

For example, if Case C in section 6.2.3, is used with the operator specific key, the customers has to trust that the migration secret has been thrown away so that the key cannot be migrated to an unreliable board. The same goes for Case C in the duplication method in section 7.1.3. If the customers cannot trust that the HAS factory will set the fixedParent value to 1, then this method cannot be used.

8.1.2 Online solution

In contrast to the offline solutions, the online solution is to be fully trusted. This is because the MSA/Server, or authority, side is connected to the CU Factory. However, the online solutions also has their disadvantages as a connection has to be established between the MSA/Server and the Customer each time a board fails. Also, the MSA/Server has to store the key blobs from each customer which demands more management efforts in the MSA.

8.2 Modification possibilities

When looking at the different use cases and their properties, each use case is missing one or another property to be completely fulfilled when it comes to both security and making it easy for the customer. This section will view the different modifications that could be done to increase the number of properties for some of the cases.

TPM 1.2: Migration with a MSA

The main disadvantage with CMK migration is that it involves a MSA which contributes to an online communication. This would mean that each time a TPM breaks, the customer has to contact the MSA for tickets that would allow the migration of a CMK.

- Having a "MSA" key in each TPM hierarchy could solve the online problem. This would offer the TPMs to sign the tickets themselves. However, at the same time the TPM should not have the power to send the CMK to an unauthorized authority, so the certification of a TPM should still be checked.

TPM 1.2: Node specific MSA

One setback on the use cases is that the TPMs of the customer are not being grouped. That is, the MSA does not know which TPM belongs to which customer. This could lead to a customer stealing another customer's key and in that way also the information that the key is protecting.

- Having the node specific MSA on an USB or a smart card would provide a more secure solution but still being an offline solution. By just inserting the USB/smart card into the node when the CU is to be changed. The MSA/Server will start its execution and the key will be migrated to the TPM/-s. Also, a password could be used so every time the key blob is to be migrated to a new TPM, a password has to be given. This would ensure that no unauthorized authority can obtain the key blob.

TPM 2.0: Case C

The disadvantage with Case C is that the HAS Factory in some cases cannot be trusted. In the duplication case the HAS Factory has to set a fixedParent value into the customer specific key.

- This can be solved by adding a *fixedChild* value. By setting this value, the customer specific key that is being created can be set to a fixed parent as well. This could be done by having a fixedChild value into the generic key which will decide that its child keys cannot be duplicated. By doing this, when the key is then added, they cannot be duplicated further.

Chapter 9

Uniform API

Today, TPM 1.2 is the version that is being used, however, in the near feature TPM 2.0 will be in the market and ready to replace the older version.

The migration and duplication steps with a TPM 1.2 and TPM 2.0 respectively will involve a similar communication between the involved parties. Thus, with some changes to the software, the system can, mutatis mutandis, accommodate both TPM versions.

This chapter will describe an API which will suit both the chosen methods from TPM 1.2 and TPM 2.0.

We characterize the API through message flow charts. The plain line represents the commands needed to be executed for TPM 1.2, while a dashed line represents the commands needed to be executed for TPM 2.0.

9.1 Case A

As seen in Figure 9.1, which is a simplified high level API for Case A, the communication flow between the three involved entities can be summed into two steps.

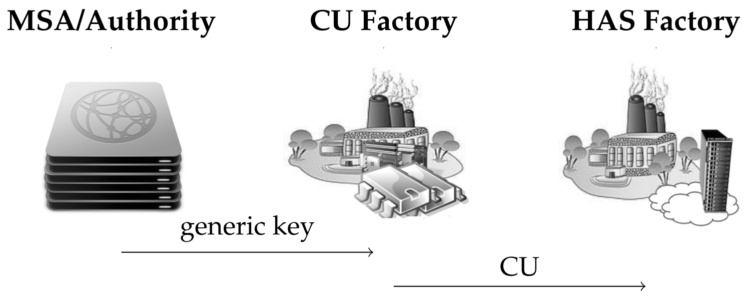

Figure 9.1: High level API for Case A

1. The first step is done after the generic keys have been generated where they are then sent from the MSA/Authority to the CU Factory
2. The second step is made after the key blob has been imported and the TPMs have been initialized. The TPMs are now ready to be sent to the HAS Factory

More details about which commands are executed in the different steps in the communication flow can be seen in Figure 9.2.

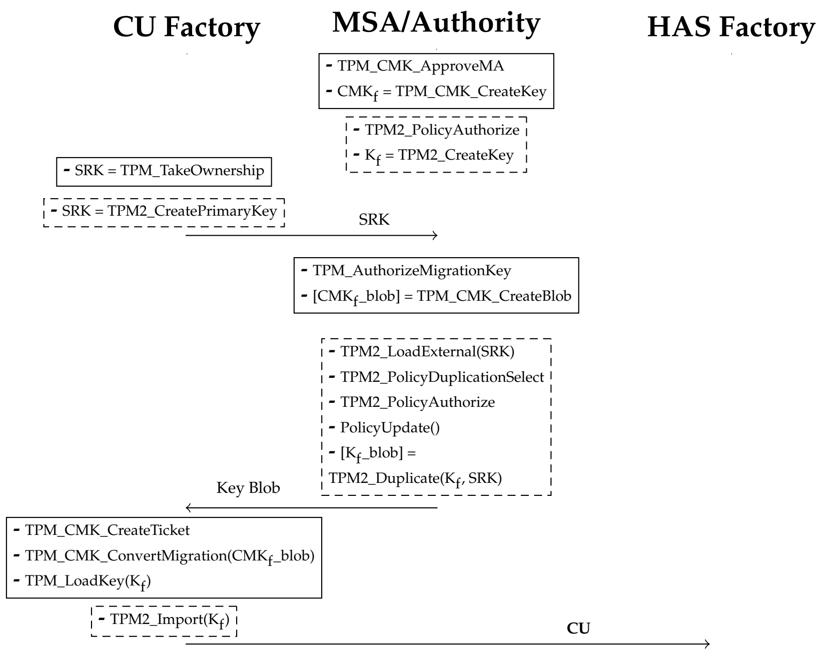

Figure 9.2: API Case A

9.2 Case B

As seen in Figure 9.3, the communication flow between the three involved entities are made in four steps for Case B, where the two latter steps are made to create a customer specific key.

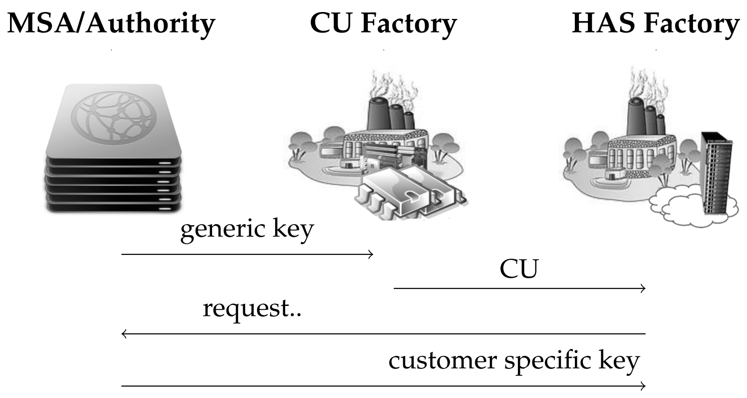

Figure 9.3: High level API for Case B

1. The first step is done after the generic keys have been generated where they are sent to the CU Factory
2. The second step is made after the key blobs have been imported and the TPM has been initialized. They are now sent to the HAS Factory
3. The third step is made when the CUs have arrived at the HAS Factory and a request for a customer specific key is sent
4. The fourth and the last step is made when the MSA has generated a customer specific key for a specific customer which is then returned to the HAS Factory

More details about which commands are executed in the different steps in the communication flow can be seen in Figure 9.4.

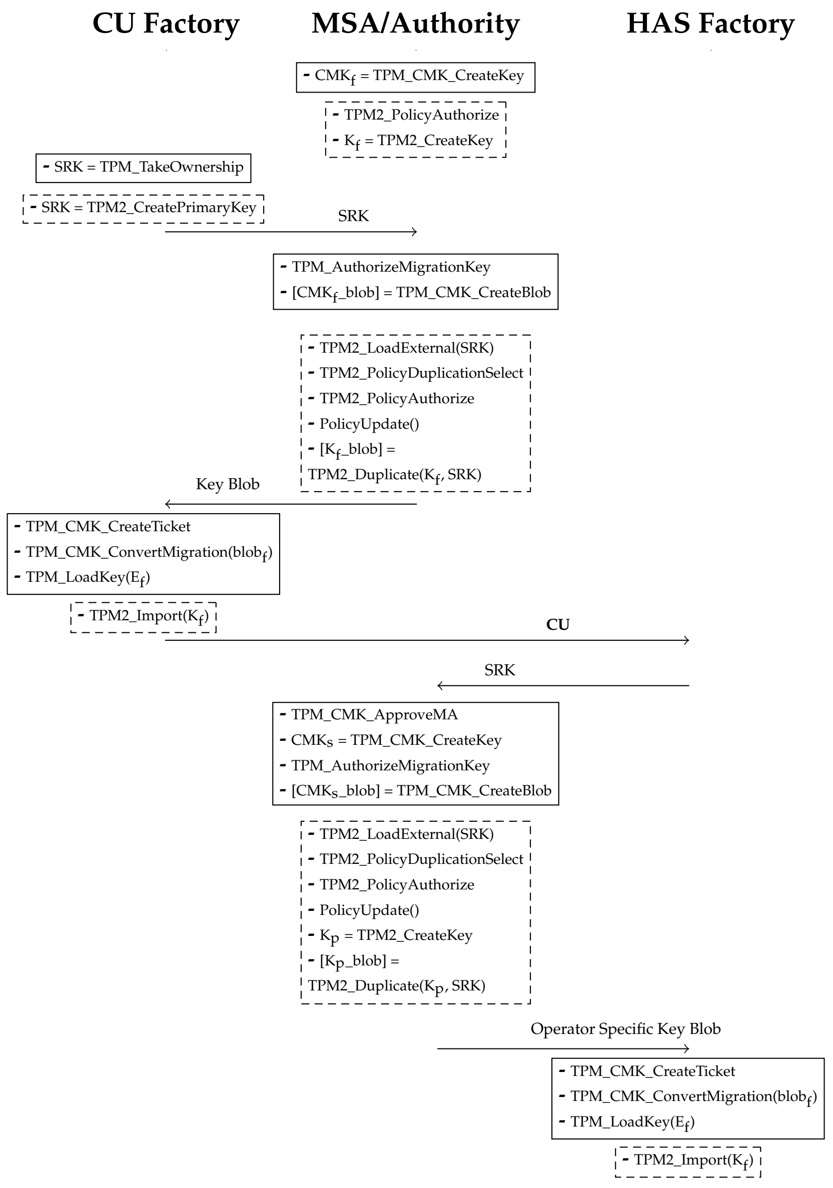
CU Factory
MSA/Authority
HAS Factory
- CMK_f = TPM_CMK_CreateKey
- TPM2_PolicyAuthorize
- K_f = TPM2_CreateKey
- SRK = TPM_TakeOwnership
- SRK = TPM2_CreatePrimaryKey
SRK
- TPM_AuthorizeMigrationKey
- [CMK_f_blob] = TPM_CMK_CreateBlob
- TPM2_LoadExternal(SRK)
- TPM2_PolicyDuplicationSelect
- TPM2_PolicyAuthorize
- PolicyUpdate()
- [K_f_blob] =
TPM2_Duplicate(K_f, SRK)
Key Blob
- TPM_CMK_CreateTicket
- TPM_CMK_ConvertMigration($blob_f$)
- TPM_LoadKey(E_f)
- TPM2_Import(K_f)
CU
SRK
- TPM_CMK_ApproveMA
- CMK_s = TPM_CMK_CreateKey
- TPM_AuthorizeMigrationKey
- [CMK_s_blob] = TPM_CMK_CreateBlob
- TPM2_LoadExternal(SRK)
- TPM2_PolicyDuplicationSelect
- TPM2_PolicyAuthorize
- PolicyUpdate()
- K_p = TPM2_CreateKey
- [K_p_blob] =
TPM2_Duplicate(K_p, SRK)
Operator Specific Key Blob
- TPM_CMK_CreateTicket
- TPM_CMK_ConvertMigration($blob_f$)
- TPM_LoadKey(E_f)
- TPM2_Import(K_f)

Figure 9.4: API Case B

9.3 Case C

As seen in Figure 9.5, the communication flow between the three involved entities are made in two steps. In broad terms, the communication flow is the same as in Case A.

However, there is one more step that is needed to be done in Case C which does not involve the communication flow itself. The third added step is made at the HAS Factory where the customer specific key is generated.

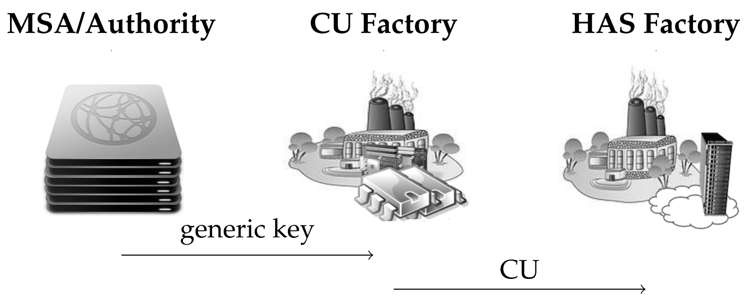

Figure 9.5: High level API for Case C

1. The first step is done after the generic keys have been generated and are sent to the CU Factory
2. The second step is made after the key blobs have been imported and the TPMs have been initialized. The CUs are then sent to the HAS Factory
3. The third step is done when the HAS Factory have received the CUs. The customer specific key is then generated and imported

More details about which commands are executed in the different steps in the communication flow can be seen in Figure 9.4.

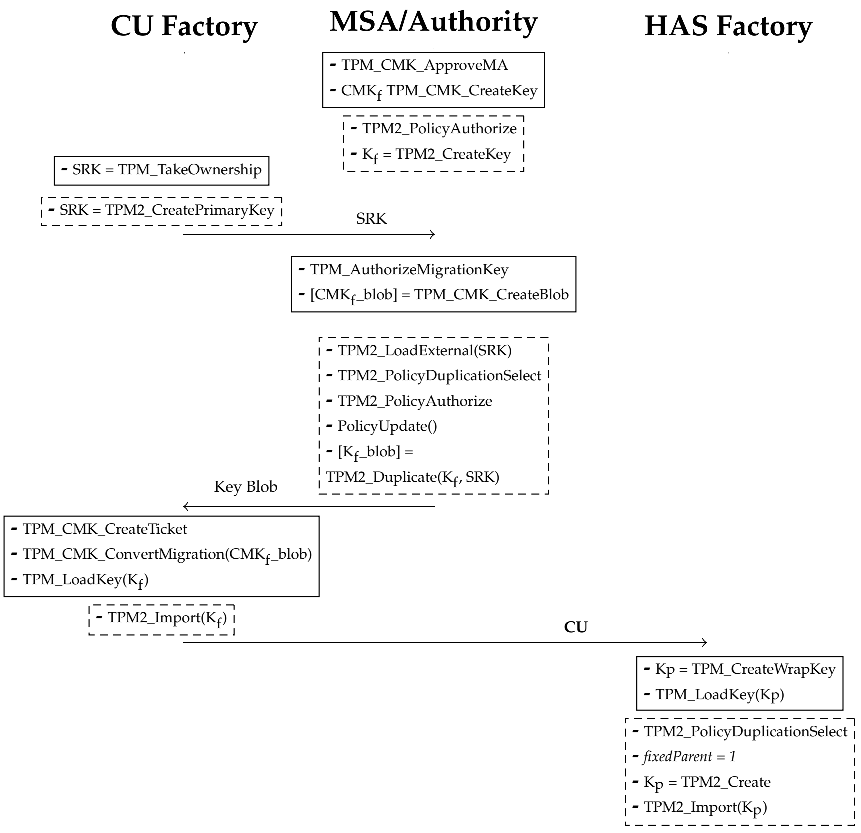

Figure 9.6: API Case C

As shown in the three different figures, the API for Case A, Case B and Case C, the communication flow between the involved parties will be the same when replacing the TPM version 1.2 to TPM version 2.0.

References

[1] International Telecommunication Unit, *Measuring the Information Society 2012*, `http://www.itu.int/ITU-D/ict/publications/idi/index.html`

[2] J.E. Ekberg, K. Kostiainen and N. Asokan, *Trusted Execution Environment in Mobile Devices*

[3] T. Alves and D. Felton, **ARM**, *TrustZone: Integrated Hardware and Software Security - Enabling Trusted Computing in Embedded Systems*

[4] G. Heiser, *The Role of Virtualization in Embedded Systems*

[5] Steven Kinney, *Trusted Platform Module Basics: Using TPM in Embedded Systems*

[6] S.X. Wang, Y.C. Wang and W.Z. Tian, *Research on Trusted Computing Implementations in Windows*, 2010

[7] R.L. Rivest, A. Shamir and L. Adleman, *Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, `http://people.csail.mit.edu/rivest/Rsapaper.pdf`

[8] Trusted Computing Group (TCG), *Trusted Platform Module Library, Version 1.2; Part 1: Design Principles*, `http://www.trustedcomputinggroup.org`

[9] Trusted Computing Group (TCG), *Trusted Platform Module Library, Version 2.0; Part 1: Architecture*, `http://www.trustedcomputinggroup.org`

[10] Trusted Computing Group (TCG), *Trusted Platform Module Library, Version 1.2; Part 2: Structure*, `http://www.trustedcomputinggroup.org`

[11] Trusted Computing Group (TCG), *Trusted Platform Module Library, Version 2.0; Part 2: Structures*, `http://www.trustedcomputinggroup.org`

[12] Trusted Computing Group (TCG), *Trusted Platform Module Library, Version 1.2; Part 3: Commands*, `http://www.trustedcomputinggroup.org`

[13] Trusted Computing Group (TCG), *Trusted Platform Module Library, Version 2.0; Part 3: Commands*, `http://www.trustedcomputinggroup.org`

Appendix A

Acronyms & terminology

authPolicy	a digest value that is included into an object when being generated
authTicket	a ticket providing authorization for migrating
CMK	Certifiable Migratable Key
CSR	Certificate Signing Request; A request for a certificate that proves that the TPM is valid
CU	Computational Unit
EK	Endorsement Key; unique identity of TPM 1.2
EPS	Endorsement Primary Seed; the root of EK hierarchy in TPM 2.0
exportTicket	ticket that is created when an object is to be exported
HAS	High Availability System
importTicket	ticket that is created when an object is to be imported
key blob	all information belonging to the key
keyFlag	value defining the state of an object
keyUsage	value defining the usage of an object
MA	Migration Authority
MSA	Migration Selection Authority

node blob	all information belonging to the node
OIAP	Object Independent Authorization Protocol
OSAP	Object Specific Authorization Protocol
Policy	an authorization type used in TPM 2.0
PPS	Platform Primary Seed; the root of platform firmware hierarchy in TPM 2.0
Primary Seed	unique identity of TPM 2.0
restrictTicket	a ticket created to verify the TPM
Shared Secret	a "password" provided when taking ownership
Shielded Location	area on the TPM protected against interference from the outside exposure
sigTicket	a ticket created to verify the authority
SPS	Storage Primary Seed; a storage key which is the root of the key hierarchy
SRK	Storage Root Key; the root of SRK hierarchy in TPM 2.0
unwrapping	decryption of a child key
wrapping	encryption of a child key

LUND UNIVERSITY

http://www.eit.lth.se