Enabling Key Migration Between Non-Compatible TPM Versions

Karlsson, Linus; Hell, Martin

Published in:
Trust and Trustworthy Computing

DOI:
10.1007/978-3-319-45572-3_6

2016

Document Version:
Peer reviewed version (aka post-print)

Enabling Key Migration Between Non-Compatible TPM Versions

Linus Karlsson and Martin Hell

Dept. of Electrical and Information Technology, Lund University,
P.O. Box 118, 221 00 Lund, Sweden
{linus.karlsson,martin.hell}@eit.lth.se

Abstract. We consider the problem of migrating keys from TPM 1.2 to the backwards incompatible TPM 2.0. The major differences between the two versions introduce several challenges for deployed systems when support for TPM 2.0 is introduced. We show how TPM 2.0 support can be introduced while still maintaining the functionality specified by TPM 1.2, allowing a smoother transition to the newer version. Specifically, we propose a solution such that keys can be migrated from TPM 1.2 to TPM 2.0, while retaining behavior with regard to e.g. authorization, migration secrets, PCR values and CMK functionality. This is achieved by utilizing new functionality, such as policies, in TPM 2.0. The proposed solution is implemented and verified using TPM emulators to ensure correctness.

Keywords: Trusted Computing · TPM · Migration

1 Introduction

There are different versions of the TPM, which differ from one another in several ways. In this paper we consider TPM 1.2, introduced in 2003, and TPM 2.0 which was introduced in 2012. TPM 2.0 is not backwards compatible with TPM 1.2, but nevertheless TPM 2.0 chips are now available [4] and have started to ship in devices [5].

We consider the process of migrating from the TPM 1.2 generation chips, to the newer TPM 2.0. As new equipment comes with TPM 2.0 chips, we want to be able to move or copy keys from TPM 1.2 to the new chips, while still maintaining the same functionality. However, because of the lack of backwards compatibility, there is no such support built into the TPM specifications. This presents a problem when we would like to use the same keys even when moving to a newer TPM, for example to be able to decrypt previously encrypted data. In addition, we may want to continue to use these keys with the same functionality, despite the differences between the specifications.

The lack of backwards compatibility means that this migration has to be done manually. Keys have to be converted between different formats, and adapted to the different feature sets of the two standards. Some features in TPM 1.2 have no direct equivalent in TPM 2.0, but identical or similar behavior can be achieved by using new features of TPM 2.0. The goal of this paper is to give a solution for

how to achieve this for all different key types and migration alternatives in TPM 1.2. As an example, in TPM 1.2 there is a concept of a *migration secret*, which authorizes the migration of a key to another TPM. This migration secret has no direct counterpart in TPM 2.0, but the same behavior can be implemented using functionality only available in the TPM 2.0 specifications. Another example is the use of Certifiable Migratable Keys (CMKs) in TPM 1.2, which also requires a non-trivial design by expressing the functionality as policies in TPM 2.0.

We describe a process which allows us to migrate keys from a TPM 1.2 to a TPM 2.0. We start by determining a set of requirements, and present a solution which performs migration according to the presented requirements. We start by implementing the equivalent functionality of TPM 1.2's migration secret in TPM 2.0, using constructions only available in the newest TPM version. We then look at keys bound to Platform Configuration Register (PCR) values, and present a way to handle the incompatibilities in key format between TPM 1.2 and TPM 2.0. We also present a solution for CMKs, such that equivalent behavior is achieved in both TPM versions. We do not consider the case of TPM 2.0 to 1.2 migration, since it is not likely that new TPM 1.2 equipment will be deployed once equipment with TPM 2.0 has been deployed.

The paper is organized as follows. Section 2 presents a brief overview of TPM 1.2 and 2.0. In Section 3 we present our goals and requirements. In Section 4 we describe our proposed solution for different relevant scenarios, which are then extended to the case of CMKs in Section 5. Section 6 describes the implementation. Finally in Section 7, we discuss some related work. Section 8 concludes the paper.

2 Overview of TPM 1.2 and TPM 2.0

This section will give a short introduction to TPM 1.2 and 2.0, with focus on issues related to key migration. For a complete review, consult the specifications [15,16].

2.1 Overview of TPM 1.2 and Certifiable Migratable Keys

A TPM 1.2 provides a key hierarchy of asymmetric keys. Keys can be of different types, for example storage keys, signing keys, or decryption keys (the last called binding key in TPM 1.2). Since the keys are asymmetric, they consist of two parts: one public and one private part. The private part of every key is encrypted with the public part of the parent key. Only a storage key can be the parent of another key.

Certain operations on the TPM, e.g. some commands related to migration, must only be performed by the TPM owner. These operations are authorized by proving knowledge of an *owner secret*, which is set when someone takes ownership of the TPM. To be able to use the private part of a key, e.g. to decrypt or sign data, the user must provide a *usage secret*. This secret is stored inside the key in the TPM, and can be unique for each key.

Copying keys between different TPMs is called *migration*, and was introduced in TPM 1.1[13]. To authorize such an operation the TPM owner must first authorize the destination using the command `TPM_AuthorizeMigrationKey`. We note that the TPM owner can authorize any destination, thus making it possible to migrate the key to any TPM, or even to a keypair generated outside any TPM. In addition, the user performing the migration must prove knowledge of the *migration secret*, which is a secret set on key creation. If this secret is not known, the key is not migratable. This is verified during execution of `TPM_CreateMigrationBlob`, which outputs a data blob which can be transferred to the destination TPM. At the destination, the key can be loaded by `TPM_LoadKey2`, possibly after conversion by `TPM_ConvertMigrationBlob`.

In TPM 1.2, CMKs were introduced. Their migration is further restricted, such that instead of the migration secret above, an authorization from a trusted entity, called the Migration Selection Authority (MSA), is required. The MSAs are chosen at key creation time. During the migration, the MSA must approve the destination, either implicitly by migrating the key to the MSA itself, or by signing a ticket containing the destination. The signature is done using the private key of the MSA. By signing the ticket, the MSA approves the migration of the specified key to a specific destination. This signature is required by the source TPM to actually perform the migration.

2.2 Overview of TPM 2.0

In TPM 2.0 the asymmetric key hierarchy has been generalized, and has been replaced with an object hierarchy. Objects can be asymmetric or symmetric keys, or data blobs. The type of the object is determined by a set of flags on the object: *sign*, *decrypt*, and *restricted*. An object with the flags *decrypt* and *restricted* set is a storage key, since it can be used to encrypt and decrypt the private parts of child keys, and the *restricted* bit tells the TPM to operate only on data prepared by the TPM (for example keys). However, the storage keys in TPM 2.0 protect its child keys by using symmetric encryption instead of asymmetric. The symmetric key is derived from a seed included in the key itself. In addition to this, TPM 2.0 allows for a wide range of ciphers and algorithms, including different symmetric ciphers and hash functions.

In TPM 2.0, migration has been renamed to *duplication*. Indeed, this is a more appropriate terminology, since keys are not removed from the source when performing a migration. Instead the key will exist in both TPMs. There are two flags connected to the duplicability of a key: *fixedTPM* and *fixedParent*. A key with fixedTPM set can never leave the TPM, and can thus not be duplicated. The other flag, fixedParent, tells us if the key is fixed to its parent. If the flag is set, the key cannot be explicitly duplicated, but it may still be loaded in another TPM if it is possible to duplicate its parent.

Just like in TPM 1.2, use of the private part of a key requires a usage secret, but there is no direct equivalent of the migration secret. Instead, a more general authorization mechanism has been introduced in TPM 2.0, namely policies.

2.3 Policies in TPM 2.0

A major addition in TPM 2.0 is the introduction of policies. A policy can be used to authorize different operations on an object in the hierarchy. The policy is set at creation time, by including a value `authPolicy` in the object. This value is created by repeatedly hashing different values from different policy commands. Possible commands are for example policies based on time, signatures, or secret values. Different policies can also be combined using OR.

Before executing a command using the object, a policy hash must be built in a policy session. The session also includes context specific values which are checked during command execution, for example if we are authorizing duplication or usage of the object, or what authorization method to use. The resulting policy hash of the policy session is then compared to the `authPolicy` in the object to authorize the command execution.

In this paper we are mostly concerned with duplication and authorization. Thus, we are only interested in a subset of the different policy commands:

- `TPM2_PolicyAuthValue` requires the usage secret of the object being authorized, and does the authorization using a HMAC.
- `TPM2_PolicyAuthorize` allows us to modify an existing policy. A new policy is signed using the private key of an authority, and if this signature is valid, the policy is included in the policy session.
- `TPM2_PolicyCommandCode` limits the authorization to a certain command, for example to authorize duplication only. This is done by setting a *command code* in the current policy session.
- `TPM2_PolicyDuplicationSelect` limits the allowed destination parent when performing a duplication.
- `TPM2_PolicyOR` is a logical OR policy, true if the current policy hash matches any of the conditions in this policy.
- `TPM2_PolicyPassword` requires the usage secret of the object being authorized, and does the authorization using the password in clear.
- `TPM2_PolicyPCR` requires the PCRs (see Section 2.4) to have a specific set of values.
- `TPM2_PolicySecret` requires the usage secret of another object on the TPM.
- `TPM2_PolicySigned` requires a digital signature.

2.4 Platform Configuration Registers

Both TPM 1.2 and 2.0 have a number of Platform Configuration Registers (PCRs). Each PCR stores a hash value, which is created by repeatedly calling `TPM_Extend` or `TPM2_Extend`. The extend operation depends both on the previous PCR value, and on the new data. This can be used to store measurements of hardware configuration and software on the host. Keys in both TPM 1.2 and 2.0 can be bound to PCR values, such that the use of a key requires certain PCRs to be in a specified state. This ensures that such keys are only usable in a known environment. In addition, the PCR values can be read by using the commands `TPM_PCRRead` and `TPM2_PCR_Read`.

2.5 Comparing Migration in TPM 1.2 and TPM 2.0

From the descriptions above we see that when it comes to migration, there are several differences between the two TPM versions.

To perform a migration of a (non-CMK) TPM 1.2 key, the following criteria must be fulfilled:

1. The key must have been created with the key flag `migratable` set to TRUE.
2. The migration secret must be known.
3. The TPM owner must authorize the migration destination.
4. The usage secret of the parent key on the source TPM must be known.
5. The usage secret of the parent key on the destination TPM must be known.

In comparison, the following criteria must be fulfilled when migrating a TPM 2.0 key:

1. The key must have `fixedParent` CLEAR.
2. The command code of the policy session must be `TPM_CC_Duplicate`, i.e. the key must have a policy which allows for duplication.
3. The usage secret of the parent key on the source TPM must be known.
4. The usage secret of the parent key on the destination TPM must be known.

We first note the similarities, namely that for both TPM versions, the usage secret of the parent key on the source TPM must be known, such that the key to be migrated can be loaded into the TPM. In addition, the usage secret of the destination TPM's parent key must also be known, such that the key to be migrated can be added as a child key.

In TPM 1.2 there is an explicit flag which tells whether or not the key is migratable. This is not the case in TPM 2.0, where there are two flags which control the migratability of a key. If `fixedParent` is SET, then the key has a fixed parent, and cannot be migrated directly (however, it could still be migrated if its parent is migratable). If `fixedTPM` is SET, the key can never be migrated. We note that it is not possible to create a key with `fixedParent` CLEAR and `fixedTPM` SET, so a sufficient condition is that `fixedParent` is CLEAR.

Another difference is the authorization of the migration. In TPM 1.2 this is done by proving knowledge of the migration secret. In TPM 2.0, it is done with a policy session that authorizes the migration. We note that the policy session is a more generic approach, which supports multiple ways of authorizing the migration through the use of any policy command. The only requirement is that there exists a command in the chain of policy commands that explicitly sets the commandCode to `TPM_CC_Duplicate`, since duplication is a special authorization role in TPM 2.0.

Finally, we note that there is no requirement for owner authorization when performing a migration in TPM 2.0.

Looking at the migration of a CMK in TPM 1.2, the following criteria must be fulfilled:

1. The MSA must authorize the migration destination.

2. The TPM owner must authorize the migration destination.
3. The usage secret of the parent key on the source TPM must be known.
4. The usage secret of the parent key on the destination TPM must be known.

Compared to the non-CMK criteria described above, the migration secret criterion is replaced by the approval of the MSA. TPM 2.0 does not have the concept of CMKs, but the behavior can be implemented by the use of policies. Details will be presented later in Section 5.

3 Goals

We want to migrate a *migratable* key from a source TPM (TPM 1.2), hereafter called TPM_S, to a destination TPM (TPM 2.0), denoted TPM_D. The key to be migrated from TPM_S to TPM_D is denoted K.

If the source key is a CMK, then the migration must also be approved by an already existing trusted third-party, called the authority/MSA. This third party may, or may not, have a TPM module installed, but let's assume that this is the case, and call this party TPM_A.

When migrating a key between two TPMs of the same version (i.e. either $1.2 \rightarrow 1.2$, or $2.0 \rightarrow 2.0$) we can immediately import the binary migration blobs produced by the source TPM into the destination TPM. We can also be sure that all features are supported. However, when we do a migration from $1.2 \rightarrow 2.0$ the migration blob must be converted manually, taking into account the differences between the two versions.

We introduce a *conversion authority* which is a trusted entity that performs the actual binary conversion between 1.2 and 2.0, and denote this with TPM_C.

Introducing this trusted entity does not lower the security of our proposed solution. If the key K is a CMK, there is already a trusted third-party (the authority/MSA). If a new, separate, conversion authority is undesirable, it would be possible to extend the MSA to also be the conversion authority.

In the case of a non-CMK, the source key owner is in full control of K. This means that the owner may migrate it to any destination, including a destination outside of a TPM. Thus the owner has full responsibility and opportunity to choose a trusted conversion authority. It is possible to have the conversion authority on either the source or destination, a separate third system is not required. Seeing the conversion authority as a separate entity does however provide a clear separation of concerns, and simplifies reasoning in this paper.

3.1 Requirements

We want our solution to maintain the same functionality with respect to authorization when moving from TPM 1.2 to TPM 2.0. Thus, if an entity is authorized to migrate or use a key at the source TPM, it should have the possibility and authorization to do so also at the destination TPM.

To maintain the functionality when moving between the different TPMs, we identify a number of requirements which must be supported by the conversion authority.

R1. Keep the same private and public part of the RSA key, such that it can be used to decrypt previously encrypted data, or create identical signatures.
R2. Keep the same authorization requirements for key usage.
R3. Keep the same authorization requirements for key migration.
R4. If a key requires a certain state (PCR values) of the TPM, the same state should be required after migration.
R5. Support all key types of the TPM 1.2, i.e., signing, decryption, and storage keys. Both non-CMK and CMK keys should be supported.
R6. Once migrated to a TPM 2.0, it should be possible (if authorized) to further migrate the key to another TPM 2.0.
R7. The migration should be deterministic, such that if the same key is migrated twice, the result at the destination TPM should be identical after both migrations.

The motivation for R7 is that when migrating a storage key in TPM 1.2 or TPM 2.0, its child keys are implicitly migrated as well, since they can just be loaded at the destination TPM with the respective Load-commands. This allows a hierarchy to be moved incrementally, simply by moving the child keys to the destination. However, when migrating keys between TPM 1.2 and 2.0, we will have to perform a conversion step. To be able to perform the migration incrementally at different occasions, the steps involved must be deterministic.

4 Migration Scenarios

We will look at the following different migration scenarios:

1. Migration of a simple, single, key from TPM_S to TPM_D. Only signing keys and decryption keys, without considering PCR values.
2. Migration of a simple, single, key requiring specific values of the PCRs.
3. Migration of a storage key, including its child keys.
4. All of the scenarios above, for CMKs.

4.1 Signing or Decryption Key

In this case we want to migrate a signing or decryption key from TPM_S to TPM_D. Clearly we must retain both the private and public portions of the key when migrating to TPM_D. Furthermore we assume that this key is the child key of the *storage root key* (SRK), but the steps will be identical for any parent key.

Because of the differences between TPM 1.2 and 2.0, both in functionality and in the actual binary migration blob format, we must do a conversion of the binary migration blob before importing it into TPM_D. This means that we cannot simply perform the migration to the SRK of TPM_D. If we did, the migration

blob could only be decrypted by the destination TPM, which would also have to perform the actual conversion. This is not possible, since the conversion cannot be performed inside the destination TPM. Rather, we must use the previously introduced conversion authority, TPM_C. The conversion authority has its own RSA keypair, which will act as an intermediate destination during the migration.

The outline of the conversion is as follows, also depicted in Figure 1.

1. The owner of TPM_S, and the owner of K authorize the migration of K to TPM_C, by proving knowledge of the owner secret and migration secret respectively.
2. A migration blob is created by the command `TPM_CreateMigrationBlob`.
3. The migration blob is first decrypted by TPM_C, and then converted to a TPM 2.0-format, and migrated to the final destination TPM_D.
4. TPM_D imports the migration blob and now has its own copy of K.

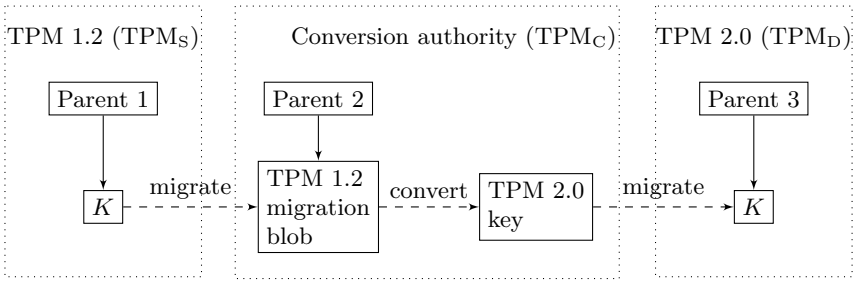

Fig. 1: Overview of migration using the conversion authority.

Conversion The conversion authority will perform the conversion of the key. The following are some important steps in this process.

TPM 2.0 supports a wide range of hash functions, and each key has a property `nameAlg` which stores the algorithm for the key. We set `nameAlg` of the TPM 2.0 key to be SHA-1, since that is the only supported hash algorithm in TPM 1.2. After this, the `usageAuth` in the TPM 1.2 key (which is the SHA-1 hash of some secret) can be moved as-is to the TPM 2.0 formatted key.

Next, we want to move the public and private part of the source key. The public part of the key, which is simply a structure from the TPM 1.2 specification, must be sent separately to TPM_C, since it is not included in the migration blob. This contains the public modulus and exponent.

The private part of the key, which we obtained by manually decrypting the migration blob with the key of TPM_C, can be copied directly to the sensitive structure in TPM 2.0, since both TPM specifications states that the private part of RSA keys is one of the two RSA primes.

Migration of the Migration Secret In TPM 1.2, each key has a migration secret, in addition to usage secret. If the value of this secret is *tpmProof*, no migration is possible since *tpmProof* is a value internal to the TPM. However, if the migration secret is the hash of a secret known to the user, migration is possible.

In TPM 2.0 there is no direct equivalent of the migration secret (called `migrationAuth`) in TPM 1.2. An analysis of the migration secret functionality provides the following four options.

1. Disallow any further migration, that is, once migrated to TPM 2.0, no more migrations will be possible. This violates requirement R6.
2. Always allow migration, that is, anyone can migrate the key. This violates requirement R3.
3. Only allow migration if the user knows the `usageAuth`. This can be implemented through a simple policy. However, this violates requirement R3.
4. Construct a more complex policy, which emulates the `migrationAuth` behavior of TPM 1.2.

Of these options, option 4 is the only one which fulfills our requirements, and most closely resembles the original behavior of TPM_S. Thus, when migrating K to TPM_D, we wish to keep the same migration secret, such that only entities with knowledge of the migration secret can migrate the key further.

In TPM 2.0, migration authorization is performed using policies. Thus, to keep the same migration secret, we must find a policy scheme that mimics the behavior of TPM 1.2.

An initial thought may be to utilize the commands `TPM2_PolicyAuthValue` or `TPM2_PolicyPassword` command in combination with setting the command code with `TPM2_PolicyCommandCode(TPM_CC_DUPLICATION)`, which would allow migration to any destination as long as a secret is known. However, both `TPM2_PolicyAuthValue` and `TPM2_PolicyPassword` use the `authValue` of the key, which is the same secret which is required for regular usage of the key. This would correspond to our discarded option 3 in the list above.

In the general case, the migration and usage secret will be different, and thus these two policy commands do not offer a solution to our problem. Another possibility is to use `TPM2_PolicySecret`. This policy command uses the `authValue` of *another* entity in the TPM. Thus we could imagine a scenario where we could create a new, separate entity whose only purpose is to keep the previous `migrationAuth` as its own usage auth. In this way, we could create a policy with `TPM2_PolicySecret` which uses this extra entity.

However, we have chosen another approach, which somewhat mimics the scenario where we have an MSA that approves our migration. This makes our proposed solution more consistent when we later on start considering CMKs. The proposed solution is depicted in Figure 2.

The `usageAuth` from our TPM 1.2 key is copied directly to the `authValue` field of the TPM 2.0 key. We also copy the `migrationAuth` from the TPM 1.2 key to the `authValue` field of a separate, newly created, signing key, called the

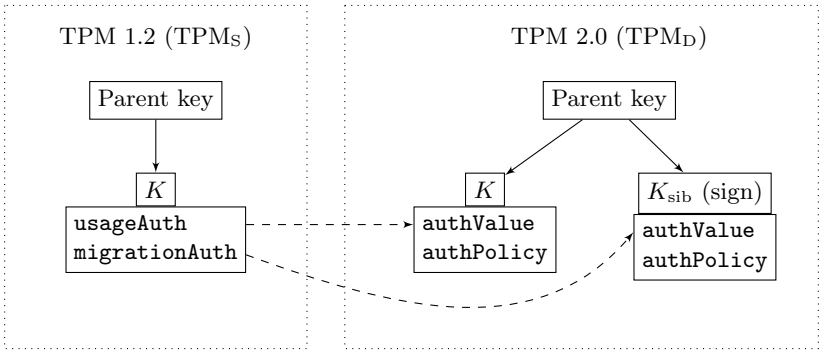

Fig. 2: Migration secret in TPM 2.0

sibling key (K_{sib}), on the TPM 2.0. Thus, to be able to create signatures using the sibling key, we must know the `authValue` of this key (which is the original `migrationAuth`).

Now, to control the migration of the key, we include a policy in the `authPolicy` field of the key K at the destination TPM. We construct the policy such that a signature from the sibling key is required for a migration to succeed. To construct such a signature, the user clearly must have knowledge of the migration secret.

Constructing a policy which validates a signature can be done by using the policy command `TPM2_PolicySigned`. The policy will require the TPM user to present a signature from the sibling key (thus proving possession of the migration secret), and if valid, `TPM2_PolicyCommandCode(TPM_CC_Duplicate)` is used to authorize a migration to any destination, mimicking the behavior of TPM 1.2.

Furthermore, in the `authPolicy` field of the *sibling key* we include a policy which allows migration of the sibling key as long as the `authValue` is known. This allows us to migrate both the sibling key and K to another TPM 2.0 destination, which fulfills requirement R6.

When creating K_{sib}, care must be taken to ensure that we get a deterministic creation. Simply creating a new, random, RSA keypair would violate requirement R7, since every migration of K would result in different K_{sib}, and thus different `authPolicy` in K. Instead, we must base the generation of K_{sib} on K, to ensure that the generation is deterministic, yet unique for all keys. Assuming that the original private part of K, the pair of primes (p, q), is random, we use a hash of (p, q) as the seed to the prime number generator to construct new primes for the sibling key. This is similar to how TPM 2.0 generates primary objects (such as the SRK) using the primary seeds in the TPM. The process is depicted in Figure 3. Since we assumed that the original (p, q) were random primes, our derived seed can also be considered random, thus giving a deterministic, but still secure K_{sib}. Clearly, if someone has knowledge of (p, q) of K, they would be able to derive K_{sib}, and authorize a migration. However, if (p, q) of K is already

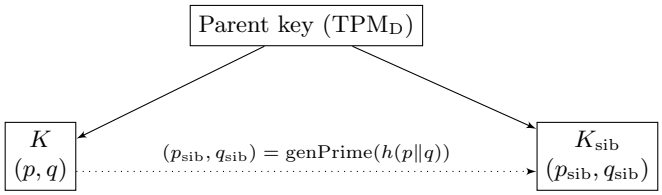

Fig. 3: Generating the primes for K_{sib} based on (p, q) of K.

known, there is no reason for an attacker to do a migration, since the private part of K is already compromised.

Owner Secret In TPM 1.2 the TPM owner is also required to authorize the migration. However this is not the case in TPM 2.0. We propose a solution where an extra signing key is introduced, similar to the sibling key above. However, different from the owner secret, this key is not unique per TPM, but rather per key. In a sense, it becomes an extra migration secret. It does deviate slightly from the behavior in TPM 1.2 since this owner signing key will have to be identical on all TPM 2.0 chips. The secret of the owner signing key is selected during the initial 1.2 to 2.0 migration, and the key will be created by the conversion authority. Just like for the migration key, the actual verification of the signature is done by including a `TPM2_PolicySigned` in the policy chain.

4.2 PCR Bound Keys

In TPM 1.2, key usage can be restricted such that both certain PCR values (through `pcrSelection`) and knowledge of the `usageAuth` is required. In TPM 2.0, this must be implemented through the use of policies. As can be seen in [16, Part 1, Annex A], this can be realized by combining the use of `TPM2_PolicyPCR` and `TPM2_PolicyAuthValue`. When converting the key to TPM 2.0-format, it is important to set the `userWithAuth`-attribute to CLEAR, since otherwise the user could circumvent the PCR requirement by only providing the `authValue`.

When migrating and converting from 1.2 to 2.0, the PCR values need to be moved from the `pcrSelection` structure and instead be included in the `TPM2_PolicyPCR` policy.

However, it is not possible for TPM_C to extract the PCR values from the TPM 1.2 migration blob. This is because the TPM 1.2 PCR structure present in the TPM 1.2 key only contains the hash over a structure containing multiple PCR values. The exact steps to calculate this hash is described in [15, Part 2, Sec. 5.4.1].

To be able to convert the PCR values to a format suitable for TPM 2.0, we would require access to each individual PCR value. In TPM 2.0 we will use the hash of the concatenation of all PCR values in the `TPM2_PolicyPCR` command, which is not the same structure that were used in TPM 1.2.

Thus, since we cannot extract each individual PCR value from the composite hash of the key in TPM 1.2, we cannot reconstruct a TPM 2.0 key bound to the exact same PCR values, at least not given only a migration blob. Therefore, the PCR values from TPM_S must be provided separately to the TPM_C during the conversion step.

A migration using `TPM_CreateMigrationBlob` does not require that the PCR values of the TPM are in the expected state. This means that we cannot be sure that reading PCR values using `TPM_PCRRead` returns the PCR values required to use the key. Instead, this must be verified by the conversion authority. Assuming that the PCR values, and the corresponding PCR index, are sent to the conversion authority, it can verify that these are indeed the correct values by calculating the hash in the same way as the TPM 1.2, and then compare it to the hash in the migration blob. If they match, TPM_C can then use the PCR values when converting the key for TPM 2.0.

Assuming the correct PCR values are sent to the conversion authority, we can construct a policy using `TPM2_PolicyPCR` followed by `TPM2_PolicyAuthValue`, which when combined will require both the correct PCR values and the correct usage secret.

However, we must also combine this with the policy for migration authorization in Section 4.1, such that we both can have PCR requirements and migration requirements. This does not mean that a migration requires correct PCR values (this is not required in TPM 1.2 either), but that one of the two policy branches is satisfied.

Thus, we create a policy with two branches, combined with `TPM2_PolicyOR`, as in Figure 4. Either of the two branches can be satisfied, if the left branch is satisfied, key usage is granted (if the PCR values are correct). If the right branch is satisfied, migration is authorized.

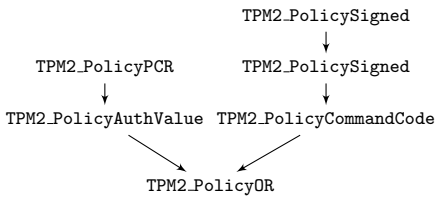

Fig. 4: Policy for PCR combined with migration authorization.

4.3 Key Hierarchies

Up until now, we have only considered the case where K is either a signing or a decryption key. If K is a storage key with child keys, we must be able to migrate the complete hierarchy as well.

Normally, when migrating keys either from 1.2 to 1.2, or from 2.0 to 2.0, there is no need to explicitly migrate the child keys. If the parent key is migrated and thus available in the destination TPM, all child keys can simply be loaded directly with `TPM_LoadKey2` or `TPM2_Load`, using the same encrypted private part on both the source and destination, without any migration.

However, due to the difference in encryption and overall key storage format between 1.2 and 2.0, a more elaborate scheme is required when migrating a hierarchy from 1.2 to 2.0.

Recall that in TPM 1.2, the parent's public key is used to encrypt the child key's private part. Thus, asymmetric encryption is used. However, in TPM 2.0, symmetric encryption is used instead. The child key's private part is encrypted using a symmetric key derived from a *seed* in the parent key. Normally, this seed is generated upon key creation, and is based on data from the RNG in the TPM. However, due to requirement R7, we require a deterministic seed. Otherwise, subsequent migrations of the same hierarchy would yield different seeds, and child keys would be encrypted with different symmetric keys, even though they share the same parent.

When migrating a complete key hierarchy, we introduce extra requirements on our solution:

1. When migrating a hierarchy, only the migration secret of the hierarchy's root key should be required to migrate the root and all of its descendant keys.
2. It should be possible to migrate parts of a hierarchy at different occasions.

Assume the hierarchy of keys given in Figure 5. If we want to migrate K, including its child keys C1 and C2, we first perform a migration of K as usual, i.e. just like if it was a signature or decryption key. However, TPM_C can see that K is a storage key, and if this is the case we include a seed inside the TPM 2.0-version of the key.

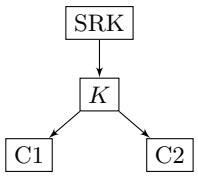

Fig. 5: Key hierarchy

We calculate the seed as $seed = \text{SHA1}(p \| q)$. The reason for using SHA-1 is because the seed must be of the same size as the `nameAlg` of the key, which is set to SHA-1 to be able to use the same `usageAuth` as in TPM 1.2.

When migrating a hierarchy, we also provide TPM_C with the encrypted private parts of the child keys of K, which we wish to migrate to TPM_D. When TPM_C receives this bundle of keys, it can use the private parts of K to decrypt all the other encrypted private parts of the child keys. The child keys can then be

converted to TPM 2.0-format, and re-encrypted using the symmetric key derived from the seed.

This approach will work for hierarchies of any depth. However, the hierarchy must be preserved inside the bundle, since TPM_C must have access to the parent of a child key to be able to decrypt it. We can also migrate only parts of a deep hierarchy, as long as all relevant parents leading to K are included.

When migrating keys in the hierarchy, their migration secret must be preserved just as before. This means that in addition to converted child keys, we will also get sibling keys for each converted child key. The sibling keys are placed so that they share parent with the key that they correspond to, see Figure 6.

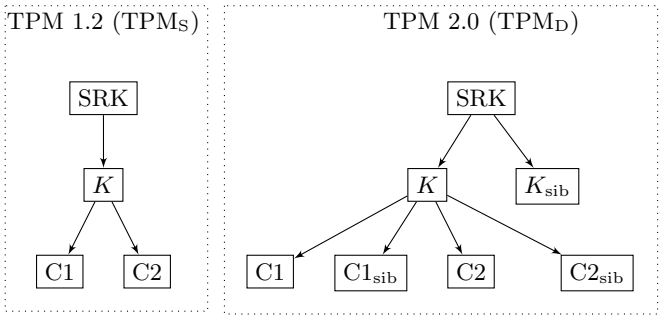

Fig. 6: Key hierarchy and sibling keys

5 Certifiable Migratable Keys

In TPM 1.2, a CMK can only be migrated with the approval of both the TPM owner and a third-party Migration Selection Authority (MSA).

In TPM 2.0, there is no direct equivalent of CMK, but the behavior can be achieved by using policies as in Figure 7. `TPM2_PolicyAuthorize` allows us to replace the previous commands in the policy chain, in this case, it allows us to replace `TPM2_PolicyDuplicationSelect` with another destination, as long as we can present a valid signature of the policy hash. This signature is done by the authority (MSA in TPM 1.2 terminology).

In this way the MSA must approve the destination before any migration can be performed, and the approval is only valid for a specific destination.

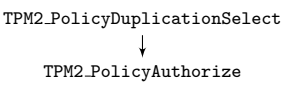

Fig. 7: Policy for CMK.

A complication introduced by CMKs is that TPM 1.2 introduces restrictions on the place of CMKs in the key hierarchy. A CMK cannot be the child of a migratable key, nor can it be the child of another CMK. When we convert a CMK into TPM 2.0 format, we must ensure that these restrictions still hold. Otherwise we would violate requirement R3, since we would be able to further migrate the child CMK if we were authorized to migrate the migratable parent.

Thus, when migrating a CMK, we must ensure that the destination parent is not a migratable key. This is the responsibility of the MSA, and is not discussed any further.

We consider the three cases in the previous section, and construct the required policy for each case.

5.1 Signing or Decryption Key

When using CMKs, there is no migration secret that the key owner needs to present. In Section 4.1 we presented a solution where two `TPM2_PolicySigned` commands were included in the `authPolicy` of K. In the CMK case, we can remove one of the signatures, since there is no migration secret. This also means that no sibling key is required, we can consider the key of the MSA as our (remote) sibling key.

Since there is no built-in requirement in TPM 2.0 for the owner to authorize a migration, we introduced an owner signing key. This signature is still required in the CMK case.

We can do this by simply adding the `TPM2_PolicySigned` command to the end of the chain. Note that adding it to the start of the chain would make it possible for the authority to override the owner authorization, which we want to avoid. Thus the chain now look like in Figure 8. `TPM2_PolicyDuplicationSelect` will set the command code to `TPM_CC_Duplicate`, so no explicit call to set the command code is required after `TPM2_PolicySigned`.

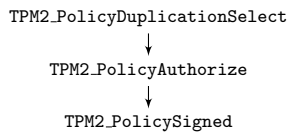

Fig. 8: Policy for CMK, with owner authorization.

5.2 PCR Bound Keys

We start with the policy from the previous section, and add a PCR policy, similar to what we did in Section 4.2. Again, we get two different branches of the policy, one for usage, and one for migration, see Figure 9. Just like before, either of the two branches can be satisfied. If the left branch is satisfied, key

usage is granted (if the PCR values are correct). If the right branch is satisfied, migration is authorized, because `TPM2_PolicyDuplicationSelect` will set the correct command code for migration.

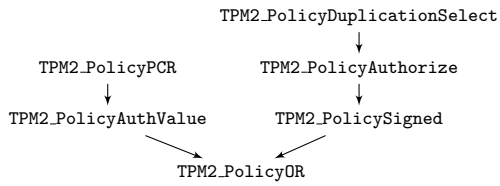

Fig. 9: Policy for PCR combined with migration authorization and CMK.

5.3 Storage Keys

Recall the restrictions on CMKs in the key hierarchy. A CMK may not have a migratable parent, neither a regular migratable key nor a CMK. The effect is the only possible key hierarchy which includes CMKs is a hierarchy where the root node is a CMK. This means that we can proceed as in Section 4.3, with the additional requirement that the root CMK key gets a policy just like in Section 5.1.

6 Implementation

To ensure that our conversion process works as intended, we have implemented all the above test cases, and verified their behavior. The TPMs have been emulated in software. For TPM 1.2, IBM's Software TPM version 4720 [3] has been used. For TPM 2.0, Microsoft's TPM2 Simulator version 1.1 [7] has been used.

To simplify the implementation, we have assumed the following:

- All TPM 1.2 keys are in the `TPM_KEY12`-key format.
- K is 2048 bit RSA, two primes. Two primes and RSA is a requirement for migratable keys according to [15, Part 2, Sec. 10.7].
- The default RSA exponent ($2^{16} + 1$) is used for all keys. For storage keys this is also required by the TPM 1.2 specification.

The TPM 1.2 specification in [15] has no defined formats on how to send migration packages between the different entities. It does, however, exist a specification [14] which describes an XML schema for supplying information about keys during the migration phase. This specification is, however, not fully updated for TPM 1.2, but rather based on TPM 1.1, and thus we have not used this XML-based approach in our implementation.

Instead, since our implementation was primary meant for testing and evaluation purposes, we have simply passed files with binary content between the different entities.

7 Related Work

While there are few widespread applications that rely on the functionality provided by the TPM, there are examples of existing pieces of software, and some other proposed use cases. From Microsoft we have both Bitlocker [6], used for full-disk encryption, and Virtual Smart Cards [8], which uses the TPM instead of physical smart cards to store private keys. Examples of proposed use cases for the newer TPM 2.0 are for example the use of TPM for tamper-proof logging [11], or the use of TPM 2.0 for electronic identities [9].

Related to the challenge of providing consistent behavior between the two TPM versions, in [2], the authors design a unified API which implements their functionality on both TPM 1.2 and 2.0. In contrast to this work, they consider the functionality for a certain use case, and then create two different and separate implementations, one for each TPM version, with no possibility of key migration between them.

The use of TPMs to provide trusted computing functionality within cloud computing is an area where there also has been development and research. In [10] the use of trusted computing in cloud platforms is discussed, and in [12] trusted snapshots of running virtual machines is discussed. Related to migrating keys between TPMs are ways of sharing keys between different TPMs. A cloud-based solution is proposed in [1].

8 Conclusions

We have proposed a solution to make it possible to move or copy key material from TPM 1.2 to TPM 2.0. Even though the two TPM versions differ significantly in functionality, and offer no backward compatibility, we have presented a design which allows the migration of keys between different versions, while still maintaining the same functionality. This allows users of the current TPM 1.2 version to start using the newer TPM 2.0 chips, still keeping the same encryption keys and functionality. In this way, previously encrypted data can be decrypted with the same set of authorization requirements as before. The required functionality was first identified and organized as a set of requirements. After this we looked at several different cases, where each case corresponded to different properties of the source key on the TPM 1.2.

We presented a way to provide the migration secret functionality of TPM 1.2 also in TPM 2.0. By introducing sibling keys and using policies, we can maintain the same authorization requirements in both TPM versions. We also handle migration of PCR bound keys from TPM 1.2 to TPM 2.0. Because of the differences in key format between the two versions, the migration requires PCR values to be sent to the conversion authority. The conversion authority can then verify the values against the source key before including them in the destination key. In addition to this, we showed how the TPM 1.2 CMK functionality can be expressed in terms of TPM 2.0 policies, and combined this with the previous results so that migration of all key types of TPM 1.2 are covered. Finally the different proposed solutions were implemented and tested using TPM emulators.

Acknowledgments. The authors would like to thank the anonymous reviewers for their helpful and valuable comments.

References

1. Chen, C., Raj, H., Saroiu, S., Wolman, A.: cTPM: A Cloud TPM for Cross-Device Trusted Applications. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). USENIX Association, Seattle, WA (Apr 2014)
2. Hell, M., Karlsson, L., Smeets, B., Mirosavljevic, J.: Using TPM Secure Storage in Trusted High Availability Systems. In: Trusted Systems: 6th International Conference, INTRUST 2014, Beijing, China. pp. 243–258. Springer International Publishing (2015)
3. IBM: IBM's software trusted platform module. `http://ibmswtpm.sourceforge.net/`
4. Infineon: Infineon Advances Trusted Computing with New OPTIGA™ TPM Family: Security Chips Serve Industrial/Embedded Environments and Support Next Generation TPM 2.0 Firmware. `http://www.infineon.com/cms/en/about-infineon/press/press-releases/2013/INFCCS201309-062.html`
5. Infineon: Infineon Expands its Trusted Computing Expertise to Mobile Devices: OPTIGA™ TPM 2.0 Chips Secure Microsoft Surface Pro 3 Tablet. `http://www.infineon.com/cms/en/about-infineon/press/press-releases/2015/INFCCS201502-026.html`
6. Microsoft: BitLocker Drive Encryption Overview. `https://www.microsoft.com/en-us/download/details.aspx?id=29076`
7. Microsoft: TSS.MSR v1.1 TPM2 simulator. `http://research.microsoft.com/en-US/downloads/35116857-e544-4003-8e7b-584182dc6833/default.aspx`
8. Microsoft: Understanding and Evaluating Virtual Smart Cards (July 2014)
9. Nyman, T., Ekberg, J.E., Asokan, N.: Citizen Electronic Identities Using TPM 2.0. In: Proceedings of the 4th International Workshop on Trustworthy Embedded Devices. pp. 37–48. TrustED '14, ACM, New York, NY, USA (2014)
10. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards trusted cloud computing. In: Proceedings of the 2009 conference on Hot topics in cloud computing. USENIX Association (2009)
11. Sinha, A., Jia, L., England, P., Lorch, J.R.: Continuous Tamper-Proof Logging Using TPM 2.0. In: Trust and Trustworthy Computing: 7th International Conference, TRUST 2014. pp. 19–36. Springer International Publishing (2014)
12. Srivastava, A., Raj, H., Giffin, J., England, P.: Trusted VM Snapshots in Untrusted Cloud Infrastructures, pp. 1–21. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
13. Trusted Computing Group: Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b (February 2002)
14. Trusted Computing Group: Interoperability Specification for Backup and Migration Services, Specification Version: 1.0 Final, Revision 1.0 (June 2005)
15. Trusted Computing Group: TPM main specification, Version 1.2, Revision 116 (March 2011)
16. Trusted Computing Group: Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.16 (October 2014)