# Introduction to the TPM 1.2

Mark Ryan
University of Birmingham

DRAFT of March 23, 2009
Comments welcome

## 1 Introduction

The Trusted Platform Module (TPM) is a hardware chip designed to enable commodity computers to achieve greater levels of security than was previously possible. There are 100 million TPMs currently in existence [2], mostly in high-end laptops made by HP, Dell, Sony, Lenovo, Toshiba, and others. The TPM stores cryptographic keys and other sensitive data in its shielded memory, and provides ways for platform software to use those keys to achieve security goals. Application software such as Microsoft's BitLocker and HP's ProtectTools use the TPM in order to guarantee security properties.

TPMs are manufactured by chip producers, including Atmel, Broadcom, Infineon, Sinosun, STMicroelectronics, and Winbond. It is specified by the Trusted Computing Group (TCG) industry consortium, that includes Intel, HP, Microsoft, AMD, IBM, Sun, Lenovo, and about 130 other members, in three documents [1] totalling about 800 pages. This introduction is intended to give a flavour and overview of its operation, and is specifically about TPM version 1.2. Some details are inevitably missing and some issues may be oversimplified here. The TPM specification is, of course, the authoritative source and overrides anything written here.

## 2 The TPM functionality

The TPM offers three kinds of functionality:

- Secure storage. User processes can store content that is encrypted by keys only available to the TPM.
- Platform measurement and reporting. A platform can create reports of its integrity and configuration state that can be relied on by a remote verifier.
- Platform authentication. A platform can obtain keys by which it can authenticate itself reliably.

## 3 Secure storage

To store data using a TPM, one creates TPM keys and uses them to encrypt the data. TPM keys are arranged in a tree structure. After the TPM has been initialised, a process called TPM_TakeOwnership is invoked that creates the *Storage Root Key* (SRK). At any time

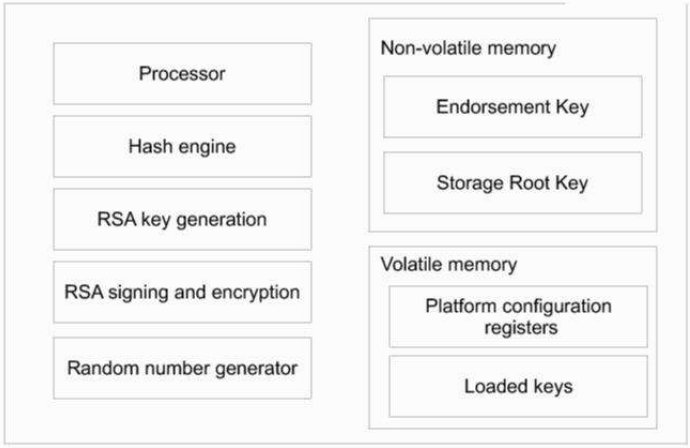


Figure 1: TPM architecture

afterwards, a user process can call TPM_CreateWrapKey to create a child key of an existing key. Once a key has been created, it may be loaded using TPM_LoadKey2, and then can be used in an operation requiring a key (e.g., TPM_Seal).

To each TPM key is associated a 160-bit string called authdata, which is analogous to a password that authorises use of the key. Authdata is specified by the user process at the time the key is created. A processes issuing a command that uses a key provides proof of knowledge of the relevant authdata by including an argument that is the value of an HMAC of other command arguments, keyed on a value based on the authdata.

The TPM_CreateWrapKey command takes arguments that include the parent key of the key to be created, new encrypted authdata of the key to be created, and other information such as the key type (sealing, binding, signature etc.), and the authorisation HMAC based on the authdata of the parent key. It returns a blob consisting of the public part of the new key and an encrypted package; the package contains the private part and the authdata of the new key, and is encrypted with the parent key. The private key of a non-migratable TPM key never leaves the TPM, except encrypted by another key. Thus, the command creates the key but does not store it; it simply returns it to the user process (protected by an encryption). The newly created key is not yet available to the TPM for use.

To use a TPM key, it must be loaded. TPM_LoadKey2 takes as argument the key blob, and returns a handle, that is, a pointer to the key stored in the TPM memory. Commands that use the loaded key refer to it by this handle. Since TPM_LoadKey2 involves a decryption by the parent key, it requires the parent key to be loaded and it requires an authorisation HMAC based on the parent key authdata. SRK is permanently loaded and has a well-known handle value, and therefore never requires to be loaded.

Once the key is loaded, an encryption command such as TPM_Seal can be used. It takes arguments including the handle of the encrypting key, the data to be encrypted, information about PCRs to which the seal should be bound, and encrypted authdata for the sealed blob. Of course, it requires an authorisation HMAC based on the authdata for the encrypting key. It returns a sealed blob. TPM_Unseal works the other way; it requires arguments including the handle and the sealed blob, and it returns the original data. It requires two authorisation HMACs; one is based on the encrypting key authdata, and the other on the sealed blob authdata.

| *Command* | *Main inputs* | *Main outputs* | *Authorisation* |
|---|---|---|---|
| TPM_CreateWrapKey<br>Creates a new TPM key. The new key is returned to the user, with the private part, the authdata, and key attributes encrypted with another TPM key, called the parent key | parent key handle;<br>ADIP-encrypted new authdata;<br>information about the key to be created | wrapped key (i.e. newly created key, encrypted with parent key) | parent key |
| TPM_LoadKey2<br>Given a wrapped key, loads it on to the TPM for usage | wrapped key | key handle | parent key |
| TPM_Seal<br>Given some data, encrypts it with a TPM key. Some PCR values that should hold on unseal may also be specified | key handle;<br>encrypted new authdata for the sealed blob;<br>PCRs for unseal;<br>data to be sealed | sealed blob | key |
| TPM_Unseal<br>Given sealed data, decrypts it. Checks that the PCR values specified in the blob are indeed current | key handle;<br>sealed blob | unsealed data | key;<br>sealed blob |
| TPM_Extend<br>Updates a PCR by "hashing in" a measurement value | PCR;<br>measurement | (none) | (none) |
| TPM_Quote<br>Obtains a signed report of the current PCR values | key handle; PCRs;<br>external data | a signature on PCR values and the external data | (none) |
| TPM_MakeIdentity<br>Create an *application identity key* (AIK) | new encrypted auth<br>info about the identity and the privacy CA | new key blob | owner<br>srk |
| TPM_ActivateIdentity<br>Decrypt an AIK certificate obtained from a Privacy CA | AIK handle<br>blob from Privacy CA | session key to decrypt the certificate | owner<br>AIK auth |

Figure 2: Some commands of the TPM

# 4 Authorisation sessions

All of the commands mentioned above require to be executed within an authorisation session. There are two kinds:

- *Object Independent Authorisation Protocol* (OIAP), which creates a session that can manipulate any object, but works only for certain commands.
- *Object Specific Authorisation Protocol* (OSAP), which creates a session that manipulates a specific object specified when the session is set up.

An authorisation session begins when the command TPM_OIAP or TPM_OSAP is successfully executed.

## 4.1 OIAP

The TPM_OIAP command starts a session with the TPM that may be used for several commands. This session type has the advantage commands may manipulate different objects (that's the "object independence" part), but it has the disadvantage that it cannot be used for commands such as TPM_CreateWrapKey that introduce new authdata to the TPM.

To set up an OIAP session, the user process sends the command TPM_OIAP to the TPM, together with a nonce argument. Nonces from the user process are called "odd" nonces, and nonces from the TPM are called "even" nonces. The user process that called TPM_OIAP receives back an authorisation handle, together with a new even nonce. Then, each command within the session sends the authorisation handle as part of its arguments, and also a new odd nonce. The response from the TPM includes a new even nonce. This system of rotating nonces guarantees new entropy in the commands and responses. All authorisation HMACs include the most recent odd nonce and even nonce. In an OIAP session, the authorisation HMACs are keyed on the authdata for the resource (e.g., key) requiring authorisation.

## 4.2 OSAP

An OSAP session can also be used for several commands, but the commands must manipulate a single object specified at the time the session was set up. The advantage of an OSAP session that it *can* be used for commands that introduce new authdata to the TPM.

To set up an OSAP session, the user process sends the command TPM_OSAP to the TPM, together with the name of the object (e.g., key handle) for the OSAP session, and an OSAP odd nonce. The response includes an authorisation handle, and an even nonce for the rolling nonces, and an OSAP even nonce. Then, the user process and the TPM each computes a secret consisting of an HMAC of the odd OSAP nonce and the even OSAP nonce, keyed on the object's authdata. This secret is called the "OSAP secret." Now commands within the authorisation session may be executed. In an OSAP session, the authorisation HMACs are keyed on the OSAP secret. The purpose of this arrangement is to permit the user process to cache the session key for a possibly extended session duration, without compromising the security of the authdata on which it is based.

### 4.2.1 ADIP encryption

When new authdata is introduced by a command, it is encrypted with a mechanism known as AuthData Insertion Protocol (ADIP). As mentioned, only OSAP sessions can be used to

insert new authdata. Roughly speaking, the new authdata is encrypted by XORing it with the OSAP secret. More precisely, a one-time key is computed as SHA-1$(s, n_e)$, where $s$ is the OSAP shared secret and $n_e$ is the current even nonce, and encryption is done by XORing the new auth data with this key. Some commands introduce two new authdata simultaneously (such as TPM_CreateWrapKey). In this case, the second one is encrypted in a similar fashion using the key SHA-1$(s, n_o)$, where $n_o$ is the current odd nonce.

Because this arrangement could expose the OSAP secret to cryptanalytic attacks if used multiple times, an OSAP session that is used to introduce new authdata with ADIP is subsequently terminated by the TPM. Commands that want to continue to manipulate the object have to create a new session.

### 4.3 Authenticating the TPM

Authdata and the authorisation sessions serve the purpose of ensuring that the calling software is authorised to carry out the command (this is done by ensuring that it knows the value of the relevant authdata). Additionally, authdata is used to enable the calling software to be sure that the response it receives is indeed the response of the TPM. An authentication HMAC is constructed by the TPM to accompany responses to authorised commands. The HMAC message is a selection of the returned values in the response, and the key is based on the relevant authdata in the same way as the authorisation HMAC.

### 4.4 Example

The figure shows the steps to create a key using TPM_CreateWrapKey (figure 3), to load a key using TPM_LoadKey2 (figure 4), using the key to encrypt data TPM_Seal (figure 5).

## 5 Platform measurement and reporting

The TPM contains a number of 160-bit registers called *platform configuration registers* (PCRs) intended to enable a relying party to obtain unforgeable information about the platform state. A component can "measure" another component (compute its hash) and insert that measurement into a PCR. This insertion is an irreversible process, known as "extending" the PCR. A PCR $p$ is extended with the measurement $m$ by the assignment

$$p := \text{SHA-1}(p||m)$$

A given PCR can be extended with any number of measurements. The final value of the PCR represents the accumulation of them all. A secure chain of trust can be established by ensuring that the very first code segment executed on power-up is measured and that measurement is extended into a PCR. Then, every component $A$ that loads another component $B$ and passes control to it ensures that $B$ is first measured and the measurement is extended into a PCR. The PCRs then represent an accumulated measurement of the history of the executed code from power-up to the present.

A TPM signing key (created by TPM_CreateWrapKey) can be used to sign the values of the PCRs, by means of the command TPM_Quote. In this way, application software can send assurance about the dtate of the platform to a third party. Additionally, PCR values can be used to ensure that certain data is accessible only to authorised software. The TPM_Seal command can take some (PCR, value) pairs as argument; then, TPM_Unseal is performed

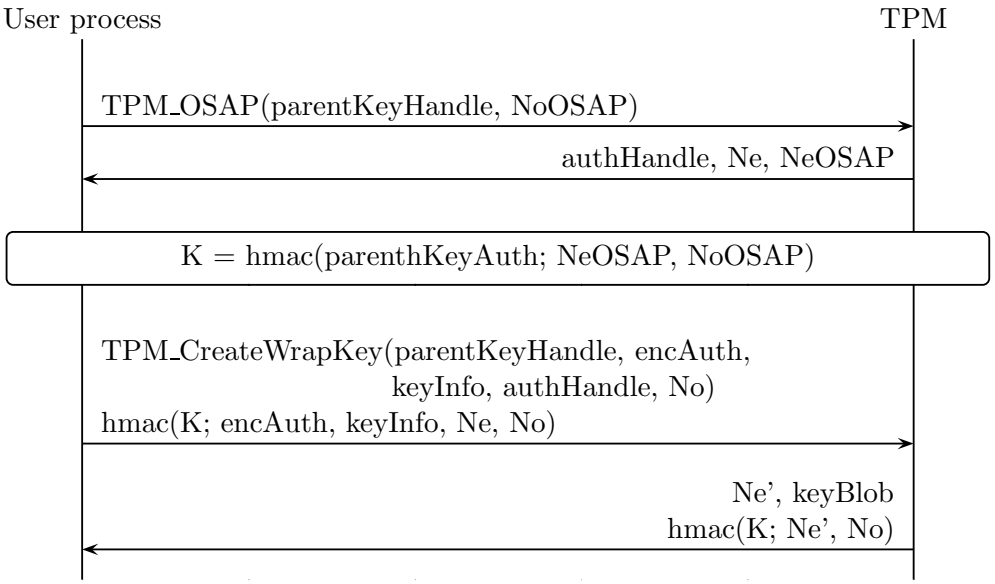


Figure 3: Creating a key

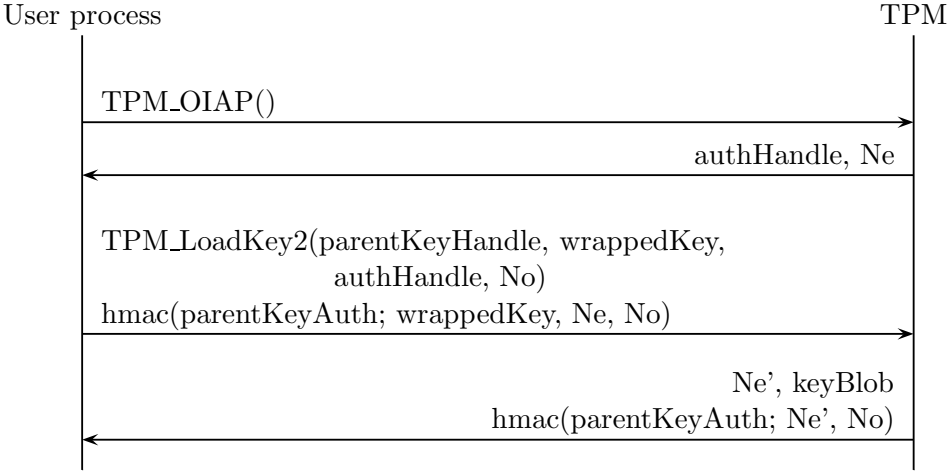


Figure 4: Loading a key

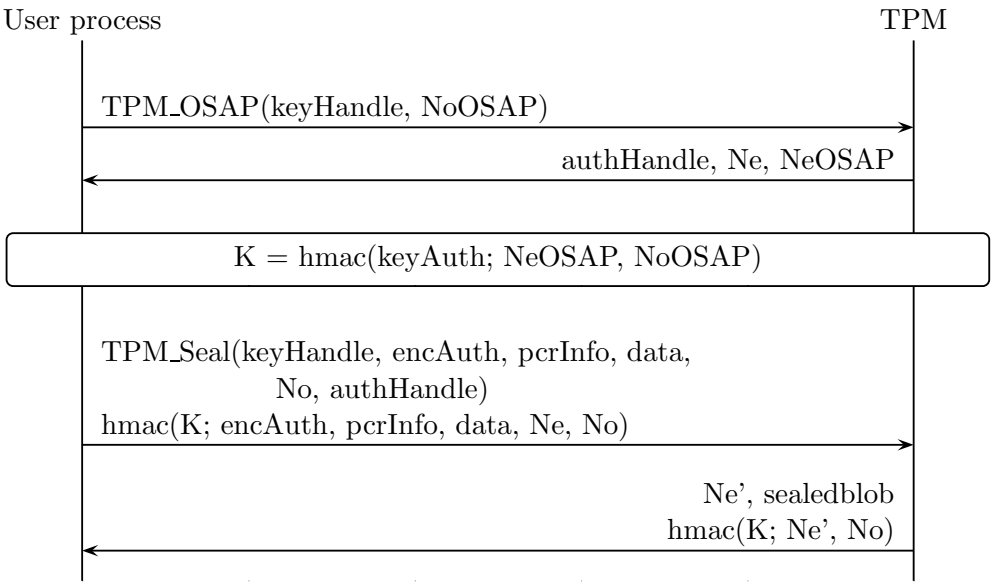


Figure 5: Using a key

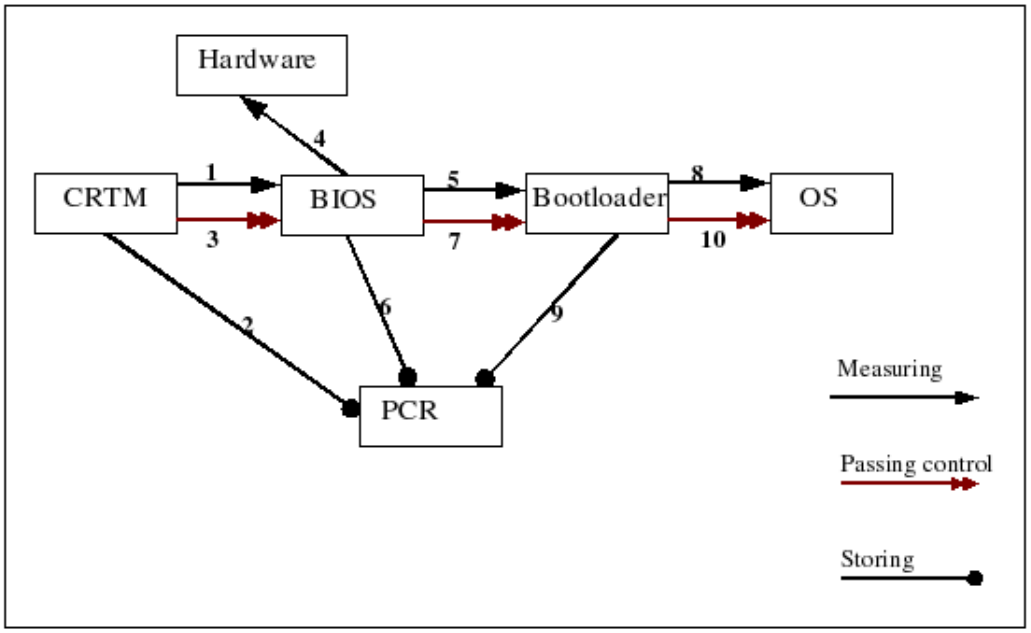


Figure 6: The measurement process, starting with the Core Root of Trust for Measurement (CRTM)

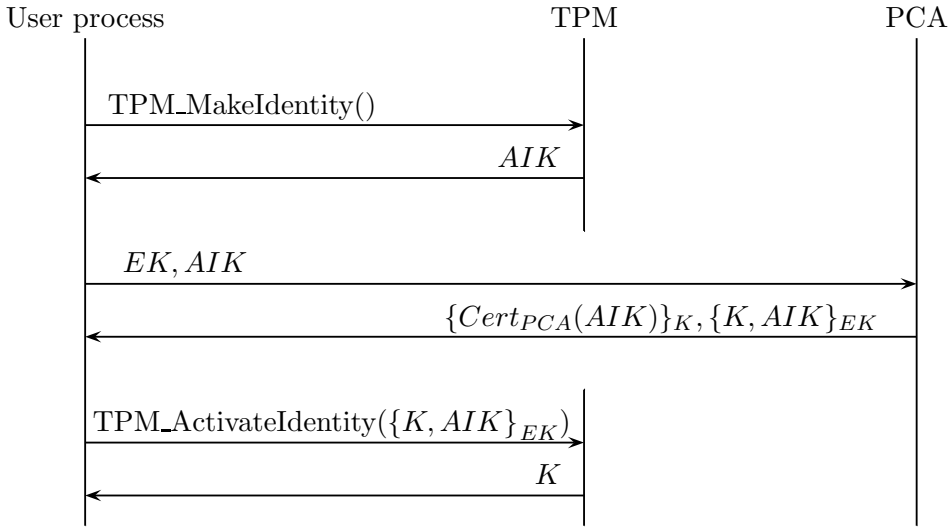


Figure 7: Attestation protocol with a Privacy CA

by the TPM only if the specified PCRs have the values that were stipulated at seal time. Similarly, TPM key can be locked to work only if certain PCRs have certain values (specified at the time the key was created).

# 6 Platform authentication

Each TPM has a unique public/private key pair called the *endorsement key* (EK), set at manufacture time and usually certified by the manufacturer. It may be changed by the platform owner, but this is probably only sensible if the owner is willing to certify the new key and the platform is required only to be trusted by parties that trust that certification (e.g., within a corporation). EK is intended to be a long-term key and can be taken to be the identity of the TPM. In addition to EK, when ownership of the TPM is taken using the command TPM_TakeOwnership, the TPM generates a public/private key pair called the *storage root key* (SRK) which is the root of the tree of storage keys; and it also generates a secret random value called *tpmProof* which is used by the TPM to identify blobs that it creates. SRK and tpmProof can also be changed, by revoking and re-taking ownership, but that destroys the ability to use all existing storage keys and therefore it is probably only done on disposal of the platform. Thus, EK, SRK and tpmProof are long-term keys which are normally not changed during the platform lifetime.

To ensure user privacy, there is no functionality that allows any of these keys to be used as platform identity. TpmProof is a secret known only to the TPM, so it cannot be used as an identity. There is no command to use SRK or EK for signing. They are both used only as encryption keys, but the TPM will only decrypt using the private parts in particular circumstances[1] that are not designed to allow them to be used as platform identifiers.

[1]EK: decryption of owner authdata during TPM_TakeOwnership, and decryption of certificates and cre-

For platform authentication, one may create signing keys known as *application identity keys* (AIKs). These may be used to sign (appropriately tagged) application-specific data, and to sign PCR values using TPM_Quote. For such signatures to be useful, AIKs need to be certified as belonging to a TPM. For reasons of user privacy, the certificate will not specify which TPM they belong to; it will just specify that they belong to a TPM. There are two ways to obtain a certificate on an AIK: using privacy CAs, and using Direct Anonymous Attestation.

### 6.1 Privacy CAs

A Privacy CA is a trusted party that will sign the public part of AIKs; such certificates are used to prove that an AIK really does belong to a valid TPM. The user software uses TPM_MakeIdentity to create an AIK and obtain the public part. It sends the public AIK and the certified public EK to the Privacy CA. The Privacy CA can check the certificate on EK (and check if it is revoked). Then the Privacy CA signs a certificate for the public AIK, and encrypts the certificate with a newly created session key (creating blob B1). It encrypts the session key and the public AIK with EK (creating blob B2), and sends both blobs back to the user software. The user software requests the TPM to decrypt B2, using TPM_ActivateIdentity. The TPM checks that the AIK in B2 really is one of its AIKs, and releases the session key. The user software uses the session to decrypt B1 and obtain the AIK certificate.

The Privacy CA solution works, but it requires a trusted party which makes it unattractive. The Privacy CA is required to be trusted by the platform owner. If the Privacy CA colludes with the relying party, then anonymity is lost. Additionally, the Privacy CA is a potential bottleneck.

### 6.2 Direct Anonymous Attestation

Direct Anonymous Attestation [3] (DAA) was created to avoid requiring the platform owner to trust an external party. In DAA, there are four parties:

- The host, consisting of a platform running operating system and application sofware.
- The TPM, inside the host.
- The issuer. It checks the EK certificate and issues a credential that can be used to sign the AIK. The issuer role is similar to the role of the Privacy CA, but because the protocol ensures that the issuer is not able to link the credential and the EK, the platform owner does not have to trust the issuer.
- The verifer, or relying party, that will check the certificate on the AIK.

An AIK certificate is obtained as follows. First, the DAA-Join protocol is run, in which the TPM in the host chooses a secret $f$, and the issuer creates a 'credential' *cre* which is a blind signature on the value $f$. The credential *cre* is encrypted by EK and sent to the TPM, which then releases it to the host. The issuer knows the value of *cre*, but because of the properties of the blind signature, he does not know the value of $f$. Next, the host engages in the DAA-Sign

dentials during certification of AIKs; SRK: decryption of child keys during TPM_LoadKey2; ((check if SRK can be used for Unbind or Unseal)).

protocol with the TPM to obtain a DAA signature on AIK using the key $(f, cre)$. The values $(f, cre)$ can be used to sign several AIKs if required. The issuer and the verifier (even if they collaborate) cannot link the DAA signaure and the *cre* used to produce it. [This is the mode $b = 0$ in DAA where the signed value (the AIK) is obtained from the TPM; there is also a mode $b = 1$ where the signed value is from outside the TPM.]

To guarantee anonymity, the platform owner does not have to trust the issuer or the verifier, but she does have to trust the host software. To guarantee the validity of the signature, the verifier has to trust the issuer (and have an authentic copy of its public key), but does not have to trust the host software. DAA employs zero-knowledge proofs and has been the subject of considerable analysis (e.g. [4, 5]).

# 7 Key migration

*This section is not finished!*

Non-migratable keys are those whose private parts never leave the TPM (except under an encryption). This is inflexible, since one might want to migrate encrypted data (and the possibility to decrypt it) from one TPM to another, or to archive and backup data that is encrypted by a TPM. Migrateable keys aim to solve this problem.

There are two types of migratable keys

- Migratable keys, introduced in TPM 1.1. Migration may be of type MIGRATE or REWRAP.
- Certifiable migratable keys, introduced in TPM 1.2. Migration may be of type RESTRICT_APPROVE or RESTRICT_MIGRATE.

If a key is to be migrated, authorisation from the TPM owner is required. TPM_AuthorizeMigrationKey is called with arguments the public part of key to be migrated and the *migration type* (one of the four types mentioned above), and an authorisation HMAC based on ownerAuth. The response is an "authorisation digest" that has tpmProof and the migration type embedded inside.

## 7.1 Migratable keys

Migratable keys are created (like non-migratable keys) by TPM_CreateWrapKey. When TPM_CreateWrapKey is called, the caller specifies whether the key is migratable or not, and if it is, the authdata allowing it to be migrated. (For non-migratable keys, the migration authdata is set to tpmProof, ensuring that the TPM can recognise the wrapped key as its own during TPM_LoadKey.)

The migration process starts with TPM_CreateMigrationBlob, which takes as input a wrapped key, its parent key handle, a migration public key, the migration type, and the authorisation digest. It requires parent key usage authorisation and wrapped key migration authorisation.

- If type is REWRAP, it unwraps the migrating key with the parent key, rewraps it with the migration public key, and returns the rewrapped key. This is appropriate if the migration public key is a TPM key belonging to another TPM.

- If type is MIGRATE, it unwraps the migrating key with the parent key, creates a random password $r$, and XOR-encrypts the private key with $r$. It then wraps the key (still having the $r$-encrypted private part) to form a "migration blob" and returns this blob and $r$.

In the case of MIGRATE, the purpose of $r$ is to avoid having to treat the migration blob as sensitive data. Later (on another TPM), one will use TPM_ConvertMigrationBlob to convert the migration blob and $r$ into a normal wrapped key. The parent key for resulting wrapped key is the same as that for the migration blob.

TPM_MigrateKey can be used to perform further unwrap-rewraps, to support store-and-forward use cases. It takes a migration blob, a public key and a key handle pointing to a key of type TPM_KEY_MIGRATE. The authorisation HMAC is based on the key handle usage auth. It decrypts the blob with the key handle, and returns a new migration blob encrypted with the public key. This function doesn't have the $r$ value so it cannot use the key.

## 7.2 Certifiable migratable keys

Certifiable migratable keys are created with TPM_CMK_CreateKey (rather than TPM_CreateWrapKey), and they are certified with TPM_CertifyKey2 (rather than TPM_CertifyKey). The certificate created by TPM_CertifyKey2 specifies

- one or more Migration Selection Authorities (MSAs), which are trusted parties that restrict migration destinations to be TPMs. Such destinations are "dynamically approved".

- zero or more Pre-approved Destinations (PADs) (also called "statically approved"). They are likely to be backup or escrow entities. A PAD's key can be protected by a TPM ((so I guess that means there is some TPM functionality that supports being a PAD)).

A relying party can inspect the certificate and decide whether he trusts the migration policies of the listed MSAs and PADs.

As well as MSAs and PADs, there are Intermediate Authorities (IAs) that perform store-and-forward roles. IAs don't know the password $r$. ((MSA's pubkey is a signature verif key. PAD's pubkey is an encryption key.))

### 7.2.1 Creating a CMK

- TPM_CMK_ApproveMA: owner of source TPM creates a ticket to approve some MSAs and some PDAs. Result: ticket that is a tpmProof-keyed HMAC of list of MSAs/PADs.

- TPM_CMK_CreateKey: like TPM_CreateWrapKey, but specifically for CMKs. Takes ticket from TPM_CMK_ApproveMA. Result is a migratable key, but whose migrationAuth is an HMAC of the migration authority and the new key's public key, signed by tpmProof (instead of being a user chosen value or tpmProof).

- TPM_AuthoriseMigrationKey: owner of source TPM approves migration of this particular key. Result: migrationKeyAuth.

So now we have the wrapped migration key, and migrationKeyAuth.

### 7.2.2 Exporting a CMK

The export can be to a PAD or an MSA. This is determined by the scheme specified in migrationKeyAuth, which may be RESTRICT_APPROVE (implies MSA) or RESTRICT_MIGRATE (implies PAD).

- In the case of MSA: We obtain some kind of signed input called restrictTicket from an MSA. Then TPM_CMK_CreateTicket takes a public key, some signed data, and a signature. It checks the signature using the public key as verification key. Result: sigTicket, consisting of an HMAC of the data keyed with tpmProof.[2]
- TPM_CMK_CreateBlob. Takes as argument
  - the CMK wrapped key
  - the migrationKeyAuth; in particular, this specifies whether the export is to MSA or PAD.
  - the MSA list (which must agree with the digest in migrationKeyAuth)
  - [in case of MSA] the sigTicket and the restrictTicket (that contains the digests of the pubkeys belonging to the Migration Authority, the destination parent key and the key-to-be-migrated)

  In the case of MSA, the TPM checks that sigTicket is the tpmProof-keyed-HMAC of one of the items in MSA list, and that the restrictTicket.destinationKey is SHA1 of migrationKeyAuth.migrationKey, and that restrictTicket.sourceKeyDigest is the digest of the public key being migrated.

  In the case of PAD, the TPM verifies that the intended migration destination is an MA (??PAD), i.e., that SHA-1[migrationKeyAuth.migrationKey equals one of the MSA list digests.

### 7.2.3 Handling exported CMKs

### 7.2.4 Importing CMKs

# 8 Delegation

*This section is not finished!*

The requirement that the calling environment possesses the authdata for resources that it uses is inflexible. It does not support use cases in which

- the TPM owner wants to delegate some (but not all) the privileges of owner to another party, and possibly withdraw those privileges later;
- the owner of a TPM key wants to delegate some (but not all) of the capabilities afforded by that key to another party.

To support use cases like this, the TPM allows the owner of a resource to create another authdata value for the resource, specifying a list of commands on the resource that the possessor of the new authdata is allowed to execute.

[2] Looks fishy. It doesn't do any check on the public key. An attacker can specify any data, signed with any throw-away key, and get it signed by tpmProof. Need to check this out.

### 8.1 How to create delegation

The owner of a key can use TPM_Delegate_CreateKeyDelegation to create a blob that specifies

- a set of commands that can be performed using the key;
- a new authdata value.

The blob is returned encrypted with a key known only to the TPM. The blob is not confidential data, but the new authdata value is. Later, anyone that can demonstrate possession of the new authdata (and that supplies the blob) can execute the specified commands on the key.

A similar command called TPM_Delegate_CreateOwnerDelegation can be used to create a blob supporting delegation of TPM owner capabilities. Again, specific commands can be stipulated.

### 8.2 How to use a delegation blob

The authorisation protocols OIAP and OSAP are supplemented by a third one called *Delegate-specific Authorisation Protocol* (DSAP). Similarly to OSAP sessions, DSAP sessions are restricted to a single object. To set up a DSAP session, the user process sends the command TPM_DSAP to the TPM, together with the name of the object (e.g., key handle or owner) for the session, and a DSAP odd nonce. Unlike in the case of OSAP, in DSAP the caller additionally sends a delegation blob. The TPM verifies that the object matches the one included in the blob. The TPM's response includes an authorisation handle, and an even nonce for the rolling nonces, and a DSAP even nonce. Similarly to OSAP, a secret is calculated from the the odd and even DSAP nonces, keyed on the delegation authdata extracted from the blob. Now commands within the authorisation session may be executed, using the session secret in place of authdata for the object. The TPM will check that the commands are permitted according to the delegation blob.

### 8.3 The family and delegate tables

As well as being presented as encrypted blobs stored off the TPM, *owner* delegations may also exist as rows in a table held in non-volatile memory of the TPM, called the *delegate table*.

Revocation of delegations is provided by means of a counter called *verificationCount* that is specified at the time the delegation is created, and is stored inside the blob or the delegation table row. When a delegation blob or delegation table row is used, the value of verificationCount must match a value stored inside the TPM. If it does not, the delegation has been revoked.

To permit fine-grained revocation, each delegation (whether presented as a blob or as a delegate table row) is associated to a delegation family that is specified when the delegation is created. The *family table* in non-volatile memory inside the TPM lists the delegation families that exist, and each family has associated with it a value of verificationCount that is the current one for delegations in the family. As well as providing the currently accepted value of verificationCount, the family table also groups delegations for management purposes. *Entities identified in the delegate table rows as belonging to the same family can edit information in the other delegate table rows with the same family id*[3].

[3]Is this really true? To be checked.

| *Command* | *Main inputs* | *Main outputs* | *Authorisation* |
|---|---|---|---|
| TPM_Delegate_CreateOwnerDelegation<br>Creates a delegation for the owner. A boolean input indicates whether the specified family verificationCount should be incremented first. | delegation permissions and family id;<br>increment;<br>encrypted auth | delegation blob | owner |
| TPM_Delegate_CreateKeyDelegation<br>Creates a delegation for usage of a key. | key handle;<br>delegation permissions and family id;<br>encrypted auth | delegation blob | key usage |
| TPM_DSAP<br>Create an authorisation session based on a delegation. The TPM checks that the delegation is still valid, by comparing its verificationCount with the relevant family member verificationCount. | object (key handle or owner);<br>delegation blob or row | authorisation session handle | (none) |
| TPM_Delegate_UpdateVerification<br>Sets a delegation's verificationCount to the current family value. If the input is a delegation blob, returns a new blob. | delegation blob or row | delegation blob | owner |
| TPM_Delegate_LoadOwnerDelegation<br>Loads an owner delegation blob into the delegate table | blob | (none) | owner, or none if no owner is installed |

Figure 8: Delegation commands of the TPM. ((Check CreateOwnerDelegation in case that no owner installed.))

A delegate's authorisation may include the commands necessary to create further delegations. But in this case, the TPM checks that the created delegate's permissions are a subset of the primary delegate's permissions. ((If a delegation includes the command TPM_UpdateVerification, then ???))

((Think about everlasting delegations. From the spec:

> *TPM_Delegate_CreateOwnerDelegation includes the ability to void all existing delegations (by incrementing the verification count) before creating the new delegation. This ensures that the new delegation will be the only delegation that can operate at Owner privilege in this family. This new delegation could be used to enable a security monitor (a local separate entity, or remote separate entity, or local host entity) to reinitialize a family and perhaps perform external verification of delegation settings. Normally the ordinals for a delegated security monitor would include TPM_Delegate_CreateOwnerDelegation (this command) in order to permit the monitor to create further delegations, and TPM_Delegate_UpdateVerification to reactivate some previously voided delegations.*
>
> *If the verification count is incremented and the new delegation does not delegate any privileges (to any ordinals) at all, or uses an authorisation value that is then discarded, this family's delegations are all void and delegation must be managed using actual Owner authorisation.*
>
> *))*

TPM_Delegate_Manage is used for managing the entries of the delegation and family tables. Families may be created and enabled or disabled. TPM_Delegate_LoadOwnerDelegation loads an existing delegation blob into the delegate table. Both these commands require owner authorisation if an owner is installed. They may be run without authorisation (e.g. to initialise tables by the manufacturer) if there is no owner installed.

## Acknowledgments

Most of my knowledge of the TPM was acquired during my visit to the System Security Lab at Hewlett Packard Labs in Bristol, UK, from January to June 2008. I was also a member of the TCG TPM Working Group during that time. I warmly thank colleagues at HP including Liqun Chen, Graeme Proudler, Dirk Kuhlmann, Boris Balacheff, Martin Sadler, David Palquin, Serdar Cabuk, Rich Smith and many others for interesting discussion and explanation, as well as colleagues in the TPM Working Group, including David Grawrock, David Wooton, Paul England, Ari Singer, Carsten Rudolph, and others. Additionally, Liqun Chen and Ben Smyth made comments on drafts of this document, helping me to improve it a lot (esp. w.r.t. DAA). I continued to learn a lot while explaining and further exploring the TPM specification with Stephanie Delaune, Steve Kremer and Graham Steel during a one-month visit to LSV, Cachan, France.

## Video

A ten minute video about the TPM is available at: `www.youtube.com/???`

## References


[1] Trusted Computing Group. TPM Specification version 1.2. Parts 1–3. www.trustedcomputinggroup.org/specs/TPM/

[2] Quote from Brian Berger in TCG press release by Wave Systems.
www.trustedcomputinggroup.org/news/press/member_releases/WAVETCGPROMOTIONMW5_31_FINAL_.pdf
See also "TCG timeline", revised April 2008.
www.trustedcomputinggroup.org/about/corporate_documents/TCG_timeline_rev_april_2008.pdf

[3] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In B. Pfitzmann and P. Liu (Eds.), Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS 2004), pp. 132-145, ACM press, 2004.

[4] B. Smyth, M. Ryan, and L. Chen. Direct Anonymous Attestation (DAA): Ensuring privacy with corrupt administrators. In proceedings of the Fourth European Workshop on Security and Privacy in Ad hoc and Sensor Networks (ESAS'07). Lecture Notes in Computer Science (LNCS), volume 4572, pp. 218-231, Springer-Verlag.

[5] M. Backes, M. Maffei, and D. Unruh. Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol. in Proceedings of 29th IEEE Symposium on Security and Privacy, May 2008.