TCG Mobile Trusted Module Specification

Specification Version 1.0
Revision 7.02
29 April 2010

Contact: admin@trustedcomputinggroup.org

TCG PUBLISHED

TCG

Disclaimer, Notices and License Terms

Table of Contents

1. Scope and Audience

The TCG specifications [1][2][3] define a Trusted Platform Module (TPM) and its use. This document is an industry specification that adapts existing TCG technology for use in a mobile phone taking into account its embedded system nature. This specification also defines new commands and structures for enabling applications [4] that the technology must enable in a mobile phone context. New commands and structures have been defined only when necessary, and alignment with the main TCG specifications has been a strong consideration.

1.1 Key words

The key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in the chapters 2-8 normative statements are to be interpreted as described in [RFC-2119].

1.2 Statement Type

Please note a very important distinction between different sections of text throughout this document. You will encounter two distinctive kinds of text: *informative comment* and *normative statements*. Because most of the text in this specification will be of the kind *normative statements*, the authors have informally defined it as the default and, as such, have specifically called out text of the kind *informative comment*. They have done this by flagging the beginning and end of each *informative comment* and highlighting its text in gray. This means that unless text is specifically marked as of the kind *informative comment*, you can consider it of the kind *normative statements*.

For example:

Start of informative comment:

This is the first paragraph of 1–n paragraphs containing text of the kind ***informative comment*** …

This is the second paragraph of text of the kind ***informative comment*** ...

This is the nth paragraph of text of the kind ***informative comment*** ...

To understand the TPM specification the user must read the specification. (This use of MUST does not require any action).

End of informative comment.

This is the first paragraph of one or more paragraphs (and/or sections) containing the text of the kind *normative statements* ...

To understand the TPM specification the user MUST read the specification. (This use of MUST indicates a keyword usage and requires an action).

1.3 References

[1] Trusted Computing Group, *TPM Main Part 1 Design Principles,* Specification Version 1.2 Revision 103, July 2007

[2] Trusted Computing Group, *TPM Main Part 2 TPM Structures,* Specification Version 1.2 Revision 103, July 2007

[3] Trusted Computing Group, *TPM Main Part 3 Commands,* Specification Version 1.2 Revision 103, July 2007

[4] Trusted Computing Group, *Mobile Phone Work Group Use Case Scenarios,* Specification Version 2.7, 2005.

[5] Trusted Computing Group, *TCG Mobile Reference Architecture*, Version 1.0 Revision 1, June 2007

2. Basic Definitions

2.1 Glossary

Abbrevation	Description
AIK	Attestation Identity Key. A key used to sign remote attestations. Defined in [2] and [3].
CBC	Cipher Block Chaining. A special mode for using a symmetric block cipher.
DES and 3DES	Cryptographic symmetric encryption algorithms
DM	Device Manufacturer.
EK	Endorsement Key. A key using which one can enroll certificates for AIK keys. Defined in [2] and [3].
MAC	Message Authentication Code. A cryptographic code for authenticating a message using a secret key.
MTM	Mobile Trusted Module
MLTM	Mobile Local-Owner Trusted Mobile
MRTM	Mobile Remote-Owner Trusted Mobile
RSA	An asymmetric cryptographic algorithm.
RTM	Root-of-Trust for Measurement
RTR	Root-of-Trust for Reporting
RTS	Root-of-Trust for Storage
RTV	Root-of-Trust for Verification
SHA1	A cryptographic hash algorithm.
TPM	Trusted Platform Module

2.2 Representation of Information

Start of informative comment:

The following structures and formats describe the interoperable areas of the specification. There is no requirement that internal storage or memory representations of data must follow these structures. These requirements are in place only during the movement of data from an MRTM or MLTM to some other entity.

End of informative comment.

2.2.1 Endness of Structures

Each structure MUST use big endian bit ordering, which follows the Internet standard and requires that the low-order bit appear to the far right of a word, buffer, wire format, or other area and the high-order bit appear to the far left.

2.2.2 Byte Packing

All structures MUST be packed on a byte boundary.

2.2.3 Lengths

The “Byte” is the unit of length when the length of a parameter is specified.

2.3 Defines

Start of informative comment:

These definitions are in use to make a consistent use of values throughout the structure specifications. The types in sections 2.2.1 and 2.2.2 are reproduced here for the reader's convenience. This document fully re-uses the type definitions from [2]. Section 2.2.3 provides the structure tags for structures defined in this specification.

End of informative comment.

2.3.1 Basic data types

Typedef	Name	Description
unsigned char	BYTE	Basic byte used to transmit all character fields.
unsigned char	BOOL	TRUE/FALSE field. TRUE = 0x01, FALSE = 0x00
unsigned short	UINT16	16-bit field. The definition in different architectures may need to specify 16 bits instead of the short definition
unsigned long	UINT32	32-bit field. The definition in different architectures may need to specify 32 bits instead of the long definition

2.3.2 Boolean types

Name	Value	Description
TRUE	0x01	Assertion
FALSE	0x00	Contradiction

2.3.3 Structure Tags

Start of informative comment:

This section defines TPM_STRUCTURE_TAG values for the structures defined in this specification.

End of informative comment.

Name	Value	Structure
TPM_TAG_VERIFICATION_KEY	0x0301	TPM_VERIFICATION_KEY
TPM_TAG_RIM_CERTIFICATE	0x0302	TPM_RIM_CERTIFICATE
MTM_TAG_PERMANENT_DATA	0x0303	MTM_PERMANENT_DATA
MTM_TAG_STANY_FLAGS	0x0304	MTM_STANY_FLAGS

2.3.4 Return codes

This specification extends the meaning of certain TPM error return codes to include new error scenarios that arise in MTM specific commands defined in this specification.

These new meanings are described in the table below.

Name	Description
TPM_BAD_COUNTER	New conditions causing this error: A TPM_VERIFICATION_KEY or TPM_RIM_CERTIFICATE had counterReference->counterSelection set to a value greater than MTM_COUNTER_SELECT_MAX. A TPM_VERIFICATION_KEY or TPM_RIM_CERTIFICATE had a counterReference->counterValue set and it was less than the referenced counter. TPM_ReadCounter failed to read the actual counter value from MTM_PERMANENT_DATA->counterBootstrap or the counter MTM_PERMANENT_DATA->counterRimProtectId.
TPM_AUTHFAIL	New conditions causing this error: TPM_VERIFICATION_KEY or TPM_RIM_CERTIFICATE has an illegitimate parentId set (e.g. TPM_VERIFICATION_KEY_ID_NONE when it is not allowed). The integrityCheckData in a TPM_VERIFICATION_KEY or TPM_RIM_CERTIFICATE is invalid when verifying using the defined verification key. The parentId of a TPM_VERIFICATION_KEY or TPM_RIM_CERTIFICATE does not match the myId of the verifying TPM_VERIFICATION_KEY.
TPM_INVALID_KEYUSAGE	New condition causing this error: The TPM_VERIFICATION_KEY_USAGE_SIGN_RIMCERT, TPM_VERIFICATION_KEY_USAGE_SIGN_RIMAUTH or TPM_VERIFICATION_KEY_USAGE_INCREMENT_BOOTSTRAP bits are not set as required in the usageFlags field in TPM_VERIFICATION_KEY.
TPM_KEYNOTFOUND	New condition causing this error: A TPM_VERIFICATION_KEY_HANDLE key handle is not defined or the key it points to is not present in the MTM.
TPM_WRONGPCRVAL	New condition causing this error: MTM_VerifyRIMCertAndExtend detected that TPM_PERMANENT_DATA was not in the state required by TPM_RIM_CERTIFICATE->state.
TPM_NOSPACE	New condition causing this error: There is insufficient room to load a verification key into an MTM.
TPM_BAD_PARAMETER	This error code can be returned if the input is syntactically incorrect.
TPM_BAD_LOCALITY	This error code can be returned by TPM_Extend if a PCR that is set to be a verified PCR (i.e. PCR index selection bit in MTM_PERMANENT_DATA->verifiedPCRs is set) is being extended using TPM_Extend.

2.3.5 Structures and Datatypes

All other type and structure definitions used in this specification that are not defined in this specification are defined in [2].

2.4 Strength of Cryptographic Hash Algorithms

Start of informative comment:

There is a need to bind a configuration (e.g. a set of public keys) to a mobile phone platform. There are many ways to do this, including placing the entire configuration into ROM on the platform. Possible optimizations to this approach are to simply place a hash of the configuration, a public key or a hash of a public key into the mobile phone platform. A configuration can then be validated by comparing its hash against the on-platform hash or checking that it was signed by a key that has a corresponding public key bound to the mobile phone platform.

The intent is to allow for these optimizations to be compliant with this specification and therefore the following definition of an *acceptable* hash function is made in the context of this spec (and for the above use).

Birthday-attack collisions are not relevant in the scenarios for which the minimal strength requirements in this section are set.

End of informative comment.

This specification uses the term *acceptable* cryptographic hash function to refer to any cryptographic hash function that meets the following criteria:

- The hash algorithm used has been standardized for use in a TPM in any version following the specifications [1][2][3].

3. Introduction

Start of informative comment:

This TCG Mobile Phone Specification, together with [5], abstracts a trusted mobile platform as a set of trusted engines, meaning constructs that can manipulate data, provide evidence that they can be trusted to report the current state of the engine, and provide evidence about the current state of the engine. This abstraction enables designers to implement platforms using one or more processors, each processor supporting one or more engines.

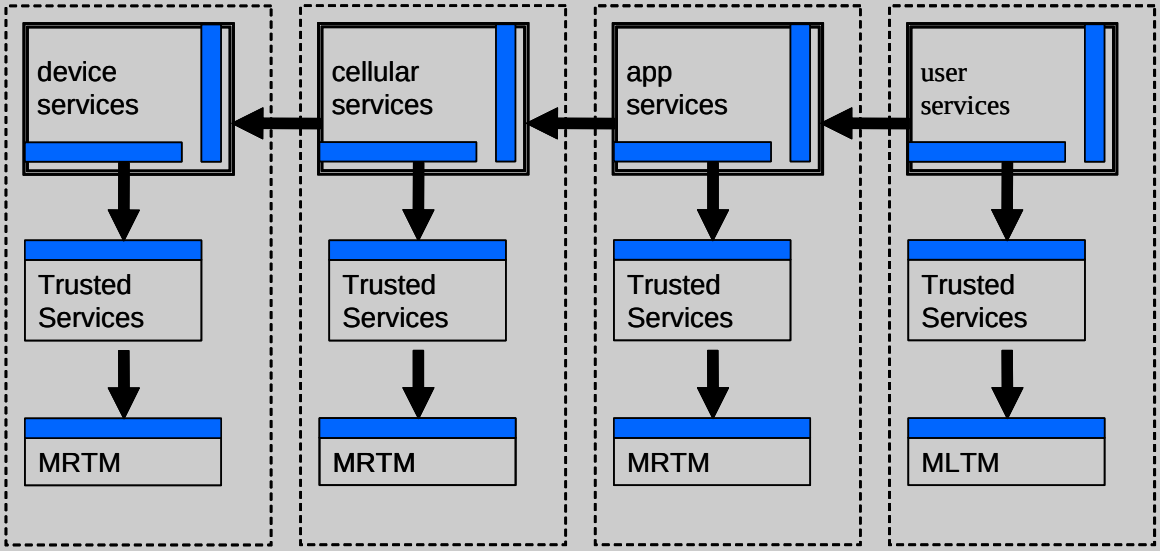


Figure 1. Example of a Generalized Mobile Platform

A generalized trusted mobile platform, shown in Figure 1, contains multiple abstract engines, each acting on behalf of a different stakeholder. The engines in Figure 1 provide services on behalf of the entities that provide the device, cellular access, an application, and user services. The solid rectangles indicate interfaces and the solid arrows indicate dependency (the arrow pointing away from the dependant entity).

In this example, the device engine provides basic platform resources, which include a user interface, debug connector, a radio transmitter and receiver, Random Number Generator, the IMEI, and a SIM interface. The device engine provides its services to an engine that provides cellular services. The cellular engine provides its services to an application engine, and the application engine provides its services to the user.

In each engine, conventional services have access to Trusted Services, which make measurements of the conventional services and store those measurements in a Mobile Trusted Module (MTM). The device, cellular, and application engines have a Mobile Remote-owner Trusted Module (MRTM), because those stakeholders do not have physical access to the phone and need a secure boot process to ensure that their engines do what is needed. The user engine has a Mobile Local-owner Trusted Module (MLTM), because the user does have physical access to the phone, and can load the software he wishes to execute. The MTMs can be trusted to report the current state of their engine, and provide evidence about the current state of the engine. The MRTM differs from the MLTM primarily in that the MRTM contains additional Protected Capabilities to support a secure boot process.

This specification defines the MRTM and the MLTM.

End of informative comment

4. Mobile Trusted Modules

Start of informative comment:

The "TCG Glossary" defines the Trusted Platform Module (TPM) as "an implementation of the functions defined in the TCG Trusted Platform Module Specification; the set of Roots of Trust with Shielded Locations and Protected Capabilities. Normally includes just the RTS and the RTR". The fundamental concept is that a TPM is the collection of all Protected Capabilities that require access to Shielded Locations (where sensitive information can be safely manipulated).

The TCG specifications "TPM Main Part 2 TPM Structures" [2] and "TPM Main Part 3 Commands" [3] describe the Protected Capabilities that require Shielded Locations. The TCG specification "TPM Main Part 1 Design Principles" [1] section #39 "Mandatory and Optional Functional Blocks" defines the TPM functional blocks that are mandatory in all types of platform, and the TPM functional blocks that are optional. Optional functional blocks may be declared as mandatory or forbidden in platform-specific TCG specifications.

This specification provides definitions of a Mobile Local-Owner Trusted Module (MLTM) and a Mobile Remote-Owner Trusted module (MRTM). These modules are defined in terms of the commands ("Protected Capabilities") they must implement. The majority of these commands are defined in the TCG TPM specifications [2] and [3]. A set of new commands and associated structures required for implementing some of the use cases in [4] are defined.

End of informative comment.

The two types of trusted modules defined by this specification are:

- Mobile Remote-Owner Trusted Module (MRTM)
- Mobile Local-Owner Trusted Module (MLTM)

The term Mobile Trusted Module (MTM) is used to refer to both MRTMs and MLTMs.

A MRTM MUST support a set of additional Mobile specific commands defined in this specification and a subset of the TPM v1.2 commands. A MLTM is NOT REQUIRED to support any additional commands defined in this specification, but MUST support a subset of TPM v1.2 commands.

This specification does NOT define or require any certain implementation method for instantiating a MRTM or a MLTM. Both trusted modules are from the viewpoint of this specification an entity exporting an interface consisting of a set of commands and associated data structures.

Start of informative comment:

The reason for having two separate types of trusted modules is their differing design objectives. A MRTM is designed to be used to implement local verification for use cases such as IMEI protection [4]. A MLTM is designed to be used to support remote verification for e.g. remote attestation. An MLTM may also be used for providing local verification under the direction of a local owner.

End of informative comment.

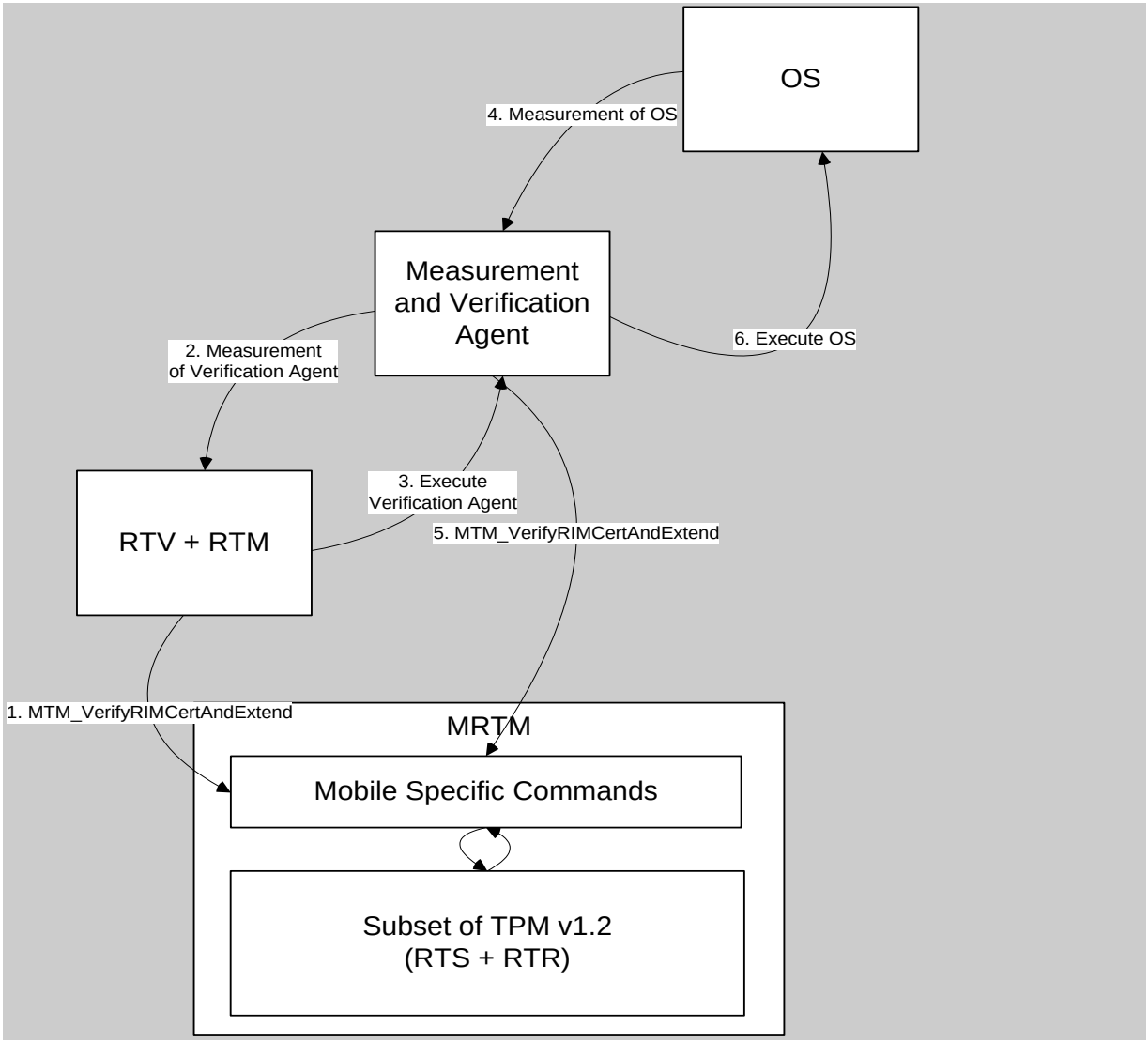


Figure 2. Overview of MRTM

Start of informative comment:

Figure 2 shows a simple example of how a MRTM could be used. The MRTM would itself consist of a subset of the TPM v1.2 plus a set of new Mobile-specific commands designed to support the requirements set by [4]. Additionally a Root-of-Trust-for-Verification (RTV) and Root-of-Trust-for-Measurement (RTM) module would be the first executable running in the runtime environment. The RTV+RTM module would first record a diagnostic measurement of its implementation. After the diagnostic extend the RTV+RTM module would measure and verify a measurement and verification agent executable using the MRTM before passing control to it. This measurement and verification agent then again measures and verifies the OS image before passing control to the OS.

This structure allows a simple implementation of secure boot. See the examples in section 10 for more detailed and concrete examples. Figure 2 is a functional diagram and shall not give any implications on which elements are implemented in hardware or software, nor depict all new functionalities of the MRTM.

End of informative comment.

5. Structures

5.1 Counter References

Start of informative comment:

A MTM implementing the commands in Section 7 MUST support 2 counters, the counterRIMProtect and the counterBootstrap. See Section 6 for descriptions of each. The MTM_PERMANENT_DATA->counterRimProtectId field contains a TPM_COUNT_ID for the counter labeled counterRIMProtect from which the counter value can be read using TPM_ReadCounter. The MTM_PERMANENT_DATA->counterBootstrap contains the actual counterBootstrap value. (Note that counterBootstrap is not exposed for external party update as a TPM Counter with its own TPM_COUNT_ID.)

The validity of objects in a MTM can be bound to defined reference counter values. This validity binding is done via a MTM_COUNTER_REFERENCE structure that is embedded into another host structure such as TPM_RIM_CERTIFICATE. The embedded MTM_COUNTER_REFERENCE structure describes the counter and reference value for the embedding object.

This structure omits the TPM_STRUCTURE_TAG field by design. This structure is intended to be embedded in the TPM_RIM_CERTIFICATE structures and TPM_VERIFICATION_KEY structures. These structures are never passed or manipulated in a stand-alone manner independent of the embedding structure.

End of informative comment.

Definition

```
#define MTM_COUNTER_SELECT_NONE 0
#define MTM_COUNTER_SELECT_BOOTSTRAP 1
#define MTM_COUNTER_SELECT_RIMPROTECT 2
#define MTM_COUNTER_SELECT_MAX 2
typedef struct MTM_COUNTER_REFERENCE_STRUCT {
    BYTE counterSelection;
    TPM_ACTUAL_COUNT counterValue;
} MTM_COUNTER_REFERENCE;
```

Parameters

Type	Name	Description
BYTE	counterSelection	IF counterSelection == MTM_COUNTER_SELECT_NONE THEN the embedding object is valid independent of any reference counter value. IF counterSelection == MTM_COUNTER_SELECT_BOOTSTRAP THEN the embedding object is valid ONLY if counterValue is greater or equal to MTM_PERMANENT_DATA->counterBootstrap. IF counterSelection == MTM_COUNTER_SELECT_RIMPROTECT THEN the embedding object is valid ONLY if counterValue is greater or equal to the counter instance identified by MTM_PERMANENT_DATA->counterRimProtectId.
TPM_ACTUAL_COUNT	counterValue	The reference value that the embedding object is bound to. This field only has relevance if *counterSelection* is NOT set to MTM_COUNTER_SELECT_NONE.

Descriptions

1. If *counterSelection* is set to MTM_COUNTER_SELECT_NONE THEN there is no reference counter and the validity of the embedding object is not bound to any single counter value.
2. IF the MTM_PERMANENT_DATA->*counterRimProtectId* counter does not exist or cannot be read THEN the embedding object is NOT VALID.
3. IF *counterSelection* is set to greater than 2 THEN the embedding object is NOT VALID.

5.2 TPM_RIM_CERTIFICATE

Start of informative comment:

A standard method is defined to provide *Reference Integrity Metrics* (RIMs) for use by the MTM. A RIM is a reference value to compare a measurement against. As an example, a RIM could be the SHA1 hash of a software image. A RIM Certificate ("RIM Cert") is an authenticated and integrity-protected structure containing a RIM and some auxiliary information. A RIM Cert can be a signed structure containing a SHA1 hash and a definition of a pre-requisite state. However, a RIM Cert is **not** a public key certificate.

There are two standardization requirements on RIM Certificates. One is a standard means for passing them to and from a MTM and allowing it to (optionally) record the use of a RIM Certificate. This must be done in such a manner that integrity is not compromised. A second standardization requirement concerns how RIM Certificates are authenticated, authorized and bound to individual MTMs. The parties creating authentic and authorized RIM Certificates are called RIM_Auths and the keys used for verifying these RIM Certificates are called TPM Verification Keys (described in the next section).

This specification describes "internal" and "external" RIM Certs. External RIM Certs are certificates which are provided to the device from outside the MTM (and presumably outside the device) and which are possibly valid in a variety of platforms. Internal RIM_Certs are certificates which could be generated on the platform itself. Both internal and external RIM Certificates are authenticated using digital signatures or message authentication codes. It is assumed (but not explicitly required) that in practice only internal RIM certificates would be authenticated using message authentication codes.

For each MTM one can set up a hierarchy of keys that can be used to authorize RIM certificates. Generally the key at the root of that hierarchy is called the RVAI (Root Verification Authority Identifier). The leaves of this hierarchy are the keys that authorize individual RIM Certs. These keys belong to RIM_Auths. A key may be both an RVAI and a RIM_Auth key. The party holding the RVAI private or secret key is ultimately responsible for providing RIM Certs to the MTM and thereby authorizing programs to run on the device embedding it. This can be done directly or via separate RIM_Auths.

The main benefit of using the MTM is that it allows to securely transform external RIM Certs of various forms into internal RIM Certs which are unique to a specific platform/engine. And then to use these transformed RIMs in an efficient way when verifying a normal boot process. These processes are described in detail in [5].

End of informative comment.

A RIM Certificate ("RIM Cert") is a structure authorizing a measurement value that is extended using MTM_VerifyRIMCertAndExtend [See Section 7] into a PCR defined in the RIM Cert. A RIM Cert consists of a set of standard information and a proprietary authentication field. The actual RIM certificate structure is defined using the notation and types in [2]. The type is named TPM_RIM_CERTIFICATE.

A RIM Cert can also be used to authorize measurements that do not result in "verify and extend" commands. In this case the command MTM_VerifyRIMCert [See Section 7] is just used to verify the integrity of the RIM Cert and the actual extend (if necessary) can be performed by an agent external to the MTM.

Definition

```
typedef struct TPM_RIM_CERTIFICATE_STRUCT {
    TPM_STRUCTURE_TAG tag;
    BYTE label[8];
    UINT32 rimVersion;
    MTM_COUNTER_REFERENCE referenceCounter;
    TPM_PCR_INFO_SHORT state;
    UINT32 measurementPcrIndex;
```

```
    TPM_PCRVALUE measurementValue;
    TPM_VERIFICATION_KEY_ID parentId;
    BYTE        extensionDigestSize;
    [size_is(extensionDigestSize)] BYTE extensionDigestData[];
    UINT32      integrityCheckSize;
    [size_is(integrityCheckSize)] BYTE integrityCheckData[];
} TPM_RIM_CERTIFICATE;
```

Parameters

Type	Name	Description
TPM_STRUCTURE_TAG	tag	This MUST be set to TPM_TAG_RIM_CERTIFICATE. It identifies the type of this structure.
BYTE	label[8]	This a proprietary label. There are no restrictions on the content of this array.
UINT32	rimVersion	This a proprietary version number for the RIM Certificate.
MTM_COUNTER_REFERENCE	referenceCounter	This field defines the validity of this structure in relation to a reference counter as described in Section 5.1.
TPM_PCR_INFO_SHORT	state	For MTM_VerifyRIMCertAndExtend to accept this certificate this field MUST contain the contents of the TPM_PERMANENT_DATA->pcrAttrib at the time of use.
UINT32	measurementPcrIndex	This field MUST contain the PCR index that is to be extended using *measurementValue* by MTM_VerifyRIMCertAndExtend.
TPM_PCRVALUE	measurementValue	This field MUST contain the measurement value to be extended into PCR index *measurementPcrIndex* by MTM_VerifyRIMCertAndExtend.
TPM_VERIFICATION_KEY_ID	parentId	This MUST be the key id of the TPM_VERIFICATION_KEY used for verifying this structure. These fields are described in section 5.3
BYTE	extensionDigestSize	This is the length in bytes of the embedded buffer *extensionDigest*. This MUST be less than or equal to 64.
BYTE[]	extensionDigest	This is a buffer containing a hash of proprietary extension data. See below for more information.
UINT32	integrityCheckSize	This MUST be the length of the buffer *integrityCheckData*.

BYTE[]	integrityCheckData	This field MUST contain an integrity check of the TPM_RIM_CERTIFICATE. This exact manner in which to verify this is defined in the object referenced by *parentId*.

Descriptions

1. RIM Certificates created by a single RIM_Auth SHOULD be uniquely identifiable by a <*label, rimVersion*>-pair. The *rimVersion* and *label* fields are not used by any of the commands defined in this specification, but are expected to be used by the agents calling those commands in lookup and management operations. The fields are integrity-protected by the integrityCheckData. As an example, the *label* field can either be used to lookup the correct TPM_RIM Certificate given a target object or identify the target object given a TPM_RIM_CERTIFICATE.

2. The TPM_PCR_INFO_SHORT *state* defines the state of the system as a set of values of the PCRs. These values are hashed into *state* as defined by [2] and [3]. The state is represented before the measurement, which the RIM certificate authorizes has taken place. Only the PCRs that must be matched exactly are to be included in *state*. TPM_PCR_INFO_SHORT is defined in [2]. The TPM_PCR_SELECTION in *state* can be empty. In this case the TPM_RIM_CERTIFICATE authorizes any measurement independently of the current PCR state.

3. The field *measurementPcrIndex* must denote the PCR index to be extended AND the *measurementValue* field must denote the actual event to be extended. MTM_VerifyRIMCertAndExtend will extend the value *measurementValue* into PCR index *measurementPcrIndex if the TPM_PERMANENT_DATA->pcrAttrib[]* matches TPM_RIM_CERTIFICATE->state and the RIM Certificate is authentic and authorized.

4. The fields *extensionDigestSize* and *extensionDigestData* define a digest of auxiliary proprietary extension data that is attached to this RIM Certificate. This field will be integrity-protected by the authorization data in IntegrityCheckData, but is not used in any other way. The intent is that this field will contain a cryptographic hash of the actual extension data and not the actual extension data itself. This is to keep the size of RIM Certificates manageable when they are cached by MTM. The *extensionDigestSize* MUST NOT be greater than 64 bytes.

5. The *integrityCheckSize* field defines the length of the *integrityCheckData* field in bytes. The *integrityCheckData* field protects the integrity and authenticity of the TPM_RIM_CERTIFICATE type. The algorithm and scheme of the integrity check is defined the structure referenced via *parentId*.

6. All compliant systems MUST be able to verify PKCS#1 v1.5 compliant signatures using SHA1 as the hash function with *integrityCheckAlgorithm* == *TPM_ALG_RSA* and *integrityCheckScheme* == *TPM_SS_RSASSAPKCS1v15_SHA1*. The integrityCheck is computed over the entire TPM_RIM_CERTIFICATE structure with the exception of the *integrityCheckData* field at the end. The TPM_RIM_CERTIFICATE structure MUST be considered as a bytestring while computing *integrityCheckData*, with no special consideration for the contents of any fields. This means that any '\0'-bytes in e.g. *label* MUST be included.

7. IF the integrityCheckData is NOT a PKCS#1 RSA signature with a 2048-bit key, it MUST have a cryptographic strength at least as strong as a 2048-bit RSA signature or a 3DES CBC-MAC in the case that the algorithm is keyed. IF an immutable cryptographic hash is used to bind a RIM Cert to a device (e.g. a hash of the RIM Cert is burned into ROM) THEN that hash algorithm must be *acceptable* as defined in Section 2.4.

8. The field *parentId* is used for looking up the correct key for verifying the *integrityCheckData* field. See Section 5.3 for the definition of the TPM_VERIFICATION_KEY_ID type. If the *parentId* field matches the *myId* field of TPM_VERIFICATION_KEY 'k1' then 'k1' is used to verify the RIM certificate. IF this field is set to TPM_VERIFICATION_KEY_ID_INTERNAL THEN that denotes that this RIM Certificate has no parent AND that it was created using MTM_InstallRIM using the key MTM_PERMANENT_DATA->internalVerificationKey.

5.3 TPM_VERIFICATION_KEY

Start of informative comment:

The TPM_VERIFICATION_KEY structure is the syntax for representing keys in the authorization hierarchy used to authorize RIM_Certs for a MTM. The TPM_VERIFICATION_KEYs can be used to represent RVAI or RIM_Auth keys. The TPM_VERIFICATION_KEY instances are used to verify TPM_RIM_CERTIFICATE structures or other TPM_VERIFICATION_KEYs. The specification also allows other purposes to be defined later by leaving unassigned bits in the *usageFlags* field.

A TPM_VERIFICATION_KEY can be authenticated and integrity-protected in the following ways:

- It is signed by an authentic and authorized TPM_VERIFICATION_KEY
- It has been loaded into the MTM before integrity checks were enabled
- A cryptographic hash (or equivalent) of the key is embedded into the MTM:

Typically a TPM_VERIFICATION_KEY structure contains only public key information, and can only be used by the MTM for verification purposes. The corresponding private key is held by the RIM_Auth and not loaded into the MTM. If the RIM_Auth uses its private key to sign a TPM_RIM_CERTIFICATE structure, the structure is therefore an instance of an external RIM_Cert, but constructed in a format that can be recognized and processed by an MTM. Such a special form of external RIM_Cert is needed in the case that the MTM has not created any internal RIM_Certs. This is discussed further in [5], Section 6.

End of informative comment.

Definition

```
// Type for containing identifier for TPM_VERIFICATION_KEY nodes
typedef UINT32 TPM_VERIFICATION_KEY_ID;

// Defined values for TPM_VERIFICATION_KEY_ID fields
#define TPM_VERIFICATION_KEY_ID_NONE 0xFFFFFFFF
#define TPM_VERIFICATION_KEY_ID_INTERNAL 0xFFFFFFFE

// These bits are reserved for the MTM
#define TPM_VERIFICATION_KEY_USAGE_MTM_MASK 0x00ff
#define TPM_VERIFICATION_KEY_USAGE_AGENT_MASK 0x0f00
// These bits are reserved for proprietary vendor extensions
#define TPM_VERIFICATION_KEY_USAGE_VENDOR_MASK 0xf000
// This bit denotes authorization to sign TPM_RIM_CERTIFICATE structures
#define TPM_VERIFICATION_KEY_USAGE_SIGN_RIMCERT 0x0001
// This bit denotes authorization to sign TPM_VERIFICATION_KEY structures
#define TPM_VERIFICATION_KEY_USAGE_SIGN_RIMAUTH 0x0002
// This bit denotes authorization to increment the bootstrap counter
#define TPM_VERIFICATION_KEY_USAGE_INCREMENT_BOOTSTRAP 0x0004

// Handle used to refer to TPM_VERIFICATION_KEY structures
typedef UINT32 TPM_VERIFICATION_KEY_HANDLE;
```

```
typedef struct TPM_VERIFICATION_KEY_STRUCT {
    TPM_STRUCTURE_TAG tag;
    UINT16      usageFlags;
    TPM_VERIFICATION_KEY_ID parentId;
    TPM_VERIFICATION_KEY_ID myId;
    MTM_COUNTER_REFERENCE referenceCounter;
    TPM_ALGORITHM_ID        keyAlgorithm;
    TPM_SIG_SCHEME  keyScheme;
    BYTE        extensionDigestSize;
    [size_is(extensionDigestSize)] BYTE extensionDigestData[];
    UINT32      keySize;
    [size_is(keySize)] BYTE keyData[];
    UINT32 integrityCheckSize;
    [size_is(integrityCheckSize)] BYTE integrityCheckData[];
} TPM_VERIFICATION_KEY;
```

Parameters

Type	Name	Description
TPM_STRUCTURE	tag	This field MUST contain the value TPM_TAG_VERIFICATION_KEY. It is used to identify the structure.
UINT16	usageFlags	This field defines the capabilities for the key contained in *keyData*. This field consists of 3 separate fields, one 8-bit field for the MRTM, one 4-bit field for Verification Agents and one 4-bit field for proprietary vendor extensions. This specification defines meaning for the 8 least significant bits in *usageFlags*. See below for definitions of the bits.
TPM_VERIFICATION_KEY_ID	parentId	This is an arbitrary identifier that is used to lookup the signing key. IF this field is set to TPM_VERIFICATION_KEY_ID_NONE THEN that denotes that this key has no parent. IF this field has the value TPM_VERIFICATION_KEY_ID_INTERNAL THEN this TPM_VERIFICATION_KEY is invalid. IF the *parentId* field of TPM_VERIFICATION_KEY key1 matches the *myId* field of a different TPM_VERIFICATION_KEY key2 THEN the key in key2->keyData is used to verify key1.
TPM_VERIFICATION_KEY_ID	myId	The *myId* is an arbitrary identifier that identifies this key structure. IF this field has the value TPM_KEY_ID_NONE THEN this key is INVALID. IF this field has the value

		TPM_KEY_ID_INTERNAL THEN this key is INVALID.
MTM_REFERENCE_COUNTER	referenceCounter	This field defines the validity of this structure in relation to a reference counter as described in Section 5.1.
TPM_ALGORITHM_ID	keyAlgorithm	This MUST be the algorithm identifier of the key in *keyData*.
TPM_SIG_SCHEME	keyScheme	This field MUST define exact manner in which to verify *integrityCheckData* fields using *keyData*.
BYTE	extensionDigestSize	This is the length in bytes of the embedded buffer *extensionDigest*. This MUST be less than or equal to 64.
BYTE[]	extensionDigest	This is a buffer containing a hash of proprietary extension data. See below for more information.
UINT32	keySize	This MUST be the length of the buffer *keyData*.
BYTE[]	keyData	This MUST contain a key for verifying *integrityCheckData* fields in the manner defined by *keyAlgorithm* and *keyScheme*.
UINT32	integrityCheckSize	This MUST be the length of the buffer *integrityCheckData*.
BYTE[]	integrityCheckData	This field MUST contain an integrity check of the TPM_VERIFICATION_KEY. This exact manner in which to verify this is defined in the object referenced by *parentId*

Descriptions

1. IF the *TPM_VERIFICATION_KEY_USAGE_SIGN_RIMAUTH* bit in *usageFlags* is set THEN the key is valid for signing other TPM_VERIFICATION_KEY structures.
2. IF the *TPM_VERIFICATION_KEY_USAGE_SIGN_RIMCERT* bit in *usageFlags* is set THEN the key is valid for signing TPM_RIM_CERTIFICATE structures.
3. IF the *TPM_VERIFICATION_KEY_USAGE_INCREMENT_BOOTSTRAP* bit is set THEN the RIM Certs signed by this key can increment the MTM_PERMANENT_DATA->counterBootstrap field.
4. A TPM_VERIFICATION_KEY structure is not valid for a purpose, unless the appropriate *usageFlags* bit is set.
5. IF the *parentId* is TPM_VERIFICATION_KEY_ID_NONE THEN this is considered to be a “root key”. These keys may however STILL be cryptographically authenticated and integrity-protected. However, the mechanism to do that is outside the scope of this specification. An acceptable solution would for example be to keep a cryptographic hash of a root key burned into a MTM and have the MTM ALWAYS accept a TPM_VERIFICATION_KEY with this hash.
6. The *extensionDigest* and *extensionDigestSize* fields are ignored except for integrity verification purposes. The intention is that *extensionDigest* can be used to store a cryptographic hash of some proprietary extension data of size *extensionDigestSize*. The *extensionDigest* could for example contain the hash of a unique device address that could be used to limit the applicability of the structure to certain identified platforms. The *extensionDigestSize* MUST NOT be greater than 64 bytes.

7. The keyData[] field MUST contain a cryptographic key for a cryptographic primitive of strength comparable to at least 3DES CBC-MAC. The format of this key is implementation-dependant. IF this key is a symmetric key THEN the confidentiality AND integrity of the structure MUST be protected. IF this key is a public key then only the integrity of this structure must be protected. This can be done either by storing the structure in a shielded location or binding this structure cryptographically to the MTM instance (e.g. by storing a hash of this structure in a shielded location). Any required confidentiality protection must be at least as strong as 3DES-CBC.

8. The *keyAlgorithm* and *keyScheme* define the exact manner in which to verify *integrityCheckData* fields of objects (e.g. TPM_VERIFICATION_KEY or TPM_RIM_CERTIFICATE structures) referring to this TPM_VERIFICATION_KEY via their *parentId* fields.

9. All compliant systems MUST be able to verify PKCS#1 v1.5 compliant signatures using SHA1 as the hash function with *integrityCheckAlgorithm* == *TPM_ALG_RSA* and *integrityCheckScheme* == *TPM_SS_RSASSAPKCS1v15_SHA1*. The integrityCheck is computed over the entire TPM_VERIFICATION_KEY structure with the exception of the *integrityCheckData* field at the end. The TPM_VERIFICATION_KEY structure MUST be considered as a bytestring while computing *integrityCheckData*, with no special consideration for the contents of any fields.

10. IF the integrityCheckData is NOT a PKCS#1 RSA signature with a 2048-bit key, it MUST have a cryptographic strength at least as strong as a 2048-bit RSA signature or a 3DES CBC-MAC in the case that the algorithm is keyed. IF an immutable cryptographic hash is used to bind a TPM_VERIFICATION_KEY to a device (e.g. a hash of the structure is burned into ROM) THEN that hash algorithm must be *acceptable* as defined in Section 2.4.

11. The TPM_VERIFICATION_KEY structures are referenced via TPM_VERIFICATION_KEY_HANDLE objects when they are loaded into a MTM.

5.4 MTM Permanent Structures

Start of informative comment:

The MTM_PERMANENT_DATA structure contains the permanent data associated with a MTM that are used by the commands defined in Section 7. This structure contains both immutable and mutable fields. There is no requirement to store this structure exactly as defined, but for convenience this specification places all these fields into a single structure.

End of informative comment.

Definition

```
// Type for indicating supported methods to load a TPM_VERIFICATION_KEY
typedef BYTE TPM_VERIFICATION_KEY_LOAD_METHODS;

typedef struct MTM_ PERMANENT_DATA_STRUCT {
    TPM_STRUCTURE_TAG tag;
    BYTE        specMajor;
    BYTE        specMinor;
    TPM_KEY aik;
    TPM_PCR_SELECTION verifiedPCRs;
    TPM_ACTUAL_COUNT counterBootstrap;
    TPM_COUNT_ID     counterRimProtectId;
    TPM_COUNT_ID     counterStorageProtectId;
    TPM_VERIFICATION_KEY_LOAD_METHODS  loadVerificationKeyMethods;
    UINT32      integrityCheckRootSize;
    [size_is(integrityCheckRootSize)] BYTE integrityCheckRootData[];
    TPM_SECRET internalVerificationKey[];
    TPM_SECRET verificationAuth;
} MTM_PERMANENT_DATA;

// The following bits are defined for the field loadVerificationKeyMethods.
#define TPM_VERIFICATION_KEY_ROOT_LOAD  0x01
#define TPM_VERIFICATION_KEY_INTEGRITY_CHECK_ROOT_DATA_LOAD 0x02
#define TPM_VERIFICATION_KEY_OWNER_AUTHORIZED_LOAD 0x04
#define TPM_VERIFICATION_KEY_CHAIN_AUTHORIZED_LOAD 0x08
// All remaining bits of this field are reserved.
```

Parameters

Type	Name	Description
TPM_STRUCTURE_TAG	Tag	This field MUST be MTM_TAG_PERMANENT_DATA.
BYTE	specMajor	Major version of the MTM spec. MUST be 0x01 for this spec version.

BYTE	specMinor	Minor version of the MTM spec. MUST be 0x00 for this spec version.
TPM_KEY	Aik	This MUST contain an identity key, in the case no endorsement key was provided.
TPM_PCR_SELECTION	verifiedPCRs	The field *verifiedPCRs* describes which PCRs MUST NOT be extended using TPM_Extend, but MUST be extended with MTM_VerifyRIMCertAndExtend. In addition, these PCRs MUST NOT be reset using TPM_PCR_Reset.
TPM_ACTUAL_COUNT	counterBootstrap	This is the value of an actual monotonic counter specific to the MTM. This counter is read and updated using TPM_GetCapability and the command MTM_IncrementBootstrapCounter defined in this specification.
TPM_COUNT_ID	counterRimProtectId	This is an id for a counter that is used to certify the validity of RIM certificates and verification keys. The counter referenced via this field is read by MTM_InstallRIM, MTM_VerifyRIMCert and MTM_VerifyRIMCertAndExtend.
BYTE	loadVerificationKeyMethods	This field contains a bit-map indicating what methods are supported by the MTM for loading TPM_VERIFICATION_KEY structures.
UINT32	integrityCheckRootSize	This field MUST contain the length of the *integrityCheckRootData* in bytes.
BYTE[]	integrityCheckRootData	This field is a proprietary field that can be used to represent an immutable cryptographic binding of a single TPM_VERIFICATION KEY or a set of TPM_VERIFICATION_KEYs to this MRTM instance. This field can also be undefined. This field MUST be undefined in a MLTM.
TPM_SECRET	internalVerificationKey	This field SHALL contain a secret unique to the MTM. The secret SHALL be used as an HMAC key by MTM_InstallRIM when creating new internal RIM Certs and used by MTM_VerifyRIMCertAndExtend when checking internal RIM Certs. The HMAC SHALL be based on an *acceptable* hash function.

TPM_SECRET	verificationAuth	This is used to authorize operations of the MTM_InstallRIM command and updates of the counter counterRIMProtect. In a MLTM this field MUST be a mirror of the TPM_PERMANENT_DATA->ownerAuth. In a MRTM, one of the following options MUST be used: - verificationAuth is unchangeable - verificationAuth is a mirror of the ownerAuth - verificationAuth is a delegate of the ownerAuth
TPM_COUNT_ID	counterStorageProtectId	This is the id of a counter for protecting storage by the user of a MTM. This field MUST be defined.

Descriptions

1. There is only a single instance of this structure in a MTM. A member *field* of this structure instance is referenced as MTM_PERMANENT_DATA->*field* in this specification.
2. The field *aik* contains an AIK in the case that the TPM_PERMANENT_DATA->endorsementKey is not valid. This key is a pre-enrolled attestation identity key that MUST be provided to the platform during manufacture IF an endorsementKey (TPM_KEY TPM_PERMANENT_DATA->endorsementKey) is NOT provided. This key must be stored in this *aik* field in this case. In such a case, it is not possible to take ownership, and the MTM MUST be an MRTM with a pre-installed owner and SRK (see [5], Section 6). The corresponding attestation identity certificate MUST also be provided, although this specification does not define HOW it must be provided. The aik key data MUST be protected with the SRK of this MTM instance.
3. The field *aik* MAY NOT be stored in the context of a MTM. Its public part MUST be however readable using the command TPM_GetCapability. The field *aik* is a part of the MTM_PERMANENT_DATA for convenience of reading and writing the specification.
4. The bits of *loadVerificationKeyMethods* MUST be set as follows. All remaining bits are reserved and MUST NOT be set. Set TPM_VERIFICATION_KEY_ROOT_LOAD if and only if TPM_Init initialises the flag *loadVerificationRootKeyEnabled* to TRUE at MTM power-up. Set TPM_VERIFICATION_KEY_INTEGRITY_CHECK_ROOT_DATA_LOAD if and only if the MTM's *integrityCheckRootData* is defined. Set TPM_VERIFICATION_KEY_OWNER_AUTHORIZED_LOAD if and only if the MTM supports the load of verification keys using owner Auth data. Set TPM_VERIFICATION_KEY_CHAIN_AUTHORIZED_LOAD if and only if the MTM can use an already loaded verification key to authorize the load of further verification keys.
5. If the *integrityCheckRootData* field is defined THEN it MUST store an immutable value that authorizes the loading of one or more TPM_VERIFICATION_KEY structures. If it is undefined for an MRTM then MTM_STANY_FLAGS->loadVerificationRootKeyEnabled MUST be TRUE until MTM_LoadVerificationRootKeyDisable has been called.
6. In a MLTM *integrityCheckRootData* MUST be undefined, e.g. it MUST NOT be used by MTM_LoadVerificationKey to accept any keys.
7. The format of *integrityCheckRootData* is proprietary. For example it could be a SHA1 hash of a known TPM_VERIFICATION_KEY structure. The strength of the cryptographic binding MUST be at least as strong as a 2048 bit RSA signature or a 3DES CBC-MAC in the case that the algorithm is keyed. IF a cryptographic hash is used to bind this structure to a device (e.g. a hash is burned into ROM) THEN that hash algorithm must be *acceptable* as defined in Section 2.4.

8. The *verificationAuth* is NOT REQUIRED to be changeable. This specification does not define any mechanism for changing this secret. This secret is expected to be created and embedded into a MRTM during creation of that MRTM.
9. This counter identified by *counterRimProtectId* is incremented using the TPM_IncrementCounter and the increment is authorized using *verificationAuth*.
10. The *verificationAuth* field in a MLTM is NOT REQUIRED if the commands using it are not implemented.
11. If the TPM_RIM_Certficate or TPM_Verification_Key handling commands are implemented in a MLTM THEN this field must be mirror of TPM_PERMANENT_DATA->ownerAuth, i.e. when *verificationAuth* is used in this specification then *ownerAuth* must be used instead. If *ownerAuth* is unset then this field must also be unset.
12. In a MLTM the *verificationAuth* field MUST NOT be directly changeable, rather all changes to this value should be done by changing TPM_PERMANENT_DATA->ownerAuth instead.
13. In a MRTM the *verificationAut*h field MUST be managed in one of the following ways:
    i) *verificationAuth* is created at manufacture of the MRTM and is not changeable (see 10)
    ii) *verificationAuth* is not directly changeable, but is a mirror of the *ownerAuth* (see 12)
    iii) *verificationAuth* is a delegate of the *ownerAuth*, managed through delegation commands

    If option ii) is used, then TPM_TakeOwnership and TPM_ChangeAuthOwner MUST be supported. If option iii) is used, then all the delegation commands in Section 9.17 MUST be supported.
14. The integrity of the MTM_PERMANENT_DATA structure MUST be protected. If the integrity protection of this structure is cryptographic THEN it MUST have a cryptographic strength at least as strong as a 2048-bit RSA signature (with SHA1) or a 3DES CBC-MAC in the case that the algorithm is keyed. IF a cryptographic hash is used to bind this structure to a device (e.g. a hash is burned into ROM) THEN that hash algorithm must be *acceptable* as defined in Section 2.4.

5.5 TPM_PERMANENT_DATA in a MTM

Start of informative comment:

This section describes authorization and authentication related secrets and keys in compliant implementations of an MTM that are used by the commands defined in [3]. The definition of the TPM_KEY, TPM_PERMANENT_DATA, TPM_SECRET and TPM_PUBKEY types used in this specification is taken from the "TPM Structures" specification v1.2 revision 85 [2].

End of informative comment.

5.5.1 Secrets and Keys from TPM v1.2

This specification does not require that a TPM v1.2 TPM_PERMANENT_DATA is used as defined in [2]. Not all fields are required and in some cases some may even be excluded or undefined. This section defines the fields in TPM_PERMANENT_DATA that this specification requires and how they are expected to be used. If not otherwise stated in this section, a field is required and used as defined in [2].

Note that this specification does NOT REQUIRE a MTM to be able to provide secure non-volatile storage that is secure against replay attacks. Secure NV storage MUST be supported. Security against replay attacks is RECOMMENDED, but may in some implementations be uneconomical. This has the implication that if a default usageAuth or ownerAuth is placed in a TPM_SECRET and the user is expected to change this default THEN an adversary may be able to replay this well-known default passphrase. As such, public defaults for TPM_SECRETs are NOT RECOMMENDED.

TPM_SECRET TPM_PERMANENT_DATA->ownerAuth;

This is the TPM owner authorization data. This specification does NOT require the ability to use the TPM owner data during runtime and as such this value may be unreadable or not present. This TPM_SECRET field MAY be immutable and hence it MAY NOT be possible to set or change it.

IF the commands requiring ownerAuth are not implemented OR these commands have delegation setup at build time THEN this ownerAuth secret does not need to be defined.

TPM_SECRET TPM_PERMANENT_DATA->adminAuth;

This field is present in revision 62 of the TPM v1.2 specification, but not in the later 85 revision. As such this field can be considered deprecated.

TPM_PUBKEY TPM_PERMANENT_DATA->manuMaintPub;

This field is required for the maintenance functionality, as specified in TPM specifications [2][3]. These commands are OPTIONAL, and if they are not implemented, then this field is also OPTIONAL. The keys used as manuMaintPub MAY be a key that is also contained in one of the TPM_VERIFICATION_KEY structures that is loaded using MTM_LoadVerificationKey.

TPM_KEY TPM_PERMANENT_DATA->endorsementKey;

This key is the endorsement key. This key MUST be defined for a MLTM. This key is OPTIONAL for a MRTM. IF this key is NOT defined or present in a compliant implementation THEN there must exist a MTM_PERMANENT_DATA->aik.

TPM_KEY TPM_PERMANENT_DATA->srk

This is the storage root key. This field MUST be present on a compliant implementation. It is RECOMMENDED that the usageAuth field is a public constant. The TPM_SECRET usageAuth field of the SRK MAY be immutable and hence it MAY NOT be possible to set or change it. IF the TPM_SECRET usageAuth field is a public-well known constant AND it is immutable THEN the usageAuth secret MUST be a 160-bit integer with the value 0.

This field MAY be pre-installed on a MRTM and MAY be immutable. The TPM_TakeOwnership command is NOT REQUIRED to be present on a MRTM.

Start of informative comment:

IF the SRK usageAuth TPM_SECRET is NOT a public value then the secure storage hierarchy cannot generally be used without the knowledge of this secret. This would imply that in a general purpose

implementation the secret would have to be widely available and evaluating its confidentiality could be difficult. This would also not add any security benefits, as the same security benefits could be achieved by creating a child key under the SRK that has a secret usageAuth, and using that as the parent key for all confidential data. As a possible alternative to a public value, the SRK's authDataUsage value could be set to TPM_AUTH_NEVER.

End of informative comment.

TPM_KEY TPM_PERMANENT_DATA->contextKey

This key is used by the TPM_SaveContext and TPM_LoadContext commands [3]. These commands are OPTIONAL and hence this field is also OPTIONAL.

TPM_KEY TPM_PERMANENT_DATA->delegateKey

This key is used to store the delegation tables in insecure storage by a TPM. This field is required ONLY if support for MANAGEMENT of delegation is needed. IF delegation tables have been instantiated during manufacture THEN this field is NOT REQUIRED. In this case however the delegation tables cannot be changed during run-time.

TPM_SECRET TPM_PERMANENT_DATA->operatorAuth;

This is an authorization secret that is required for the use of the TPM_SetTempDeactivated command [3]. This command MUST NOT be supported in a compliant MRTM implementation. This field is therefore also NOT required. In a MRTM this command MUST NOT be usable if it is not intended that the MTM can be shutdown during runtime. This is the case for example if the MTM is used to enable verification of loaded software.

The TPM_SetTempDeactivated command is required in a MLTM and therefore this field is also required in a MLTM.

5.5.2 TPM_PERMANENT_DATA in a MTM Summary

This section provides a table summarizing the OPTIONAL/REQUIRED requirements for fields in the TPM_PERMANENT_DATA structure from the one defined in [2]. This table is derived from section 4.4.1 and the REQUIRED/OPTIONAL requirements in this specification for implementing TPM commands in a MTM. If this table and the REQUIRED/OPTIONAL requirements for a command imply inconsistent REQUIRED/OPTIONAL requirements for a TPM_PERMANENT_DATA field then the requirements implied by the REQUIRED/OPTIONAL classification of commands have precedence.

Type	Name	Description
TPM_STRUCTURE_TAG	Tag	REQUIRED for MRTM and MLTM.
BYTE	revMajor	REQUIRED for MRTM and MLTM. (Major version of theTPM Main or Library Spec)
BYTE	revMinor	REQUIRED for MRTM and MLTM. (Minor version of theTPM Main or Library Spec)
TPM_NONCE	tpmProof	REQUIRED for MRTM and MLTM.
TPM_SECRET	ownerAuth	The ownerAuth MAY be immutable in a MRTM. The ownerAuth MAY be undefined or unreadable in a MRTM. REQUIRED for a MLTM.
TPM_SECRET	operatorAuth	OPTIONAL for a MRTM and REQUIRED for MLTM.
TPM_PUBKEY	manuMaintPub	OPTIONAL for compliant implementations. MAY be the same as a key used in TPM_VERIFICATION_KEY.
TPM_KEY	endorsementKey	OPTIONAL for MRTM IF a MTM_PERMANENT_DATA->aik is defined. REQUIRED for MLTM.
TPM_KEY	Srk	REQUIRED for MRTMs and MLTMs.
TPM_KEY	delegateKey	Delegation is OPTIONAL for a MTM. Therefore this field is OPTIONAL.
TPM_KEY	contextKey	OPTIONAL for MRTMs and MLTMs.
TPM_COUNTER_VALUE	auditMonotonicCounter	OPTIONAL for MRTMs and MLTMs.
TPM_COUNTER_VALUE	monotonicCounter	OPTIONAL for MRTMs and MLTMs. The counterBootstrap may be implenented independent of this field, if necessary.

TPM_PCR_ATTRIBUTES	pcrAttrib	REQUIRED for all MTMs. This specification requires that pcrAttrib[] array at least has size 16 (has 16 entries).
Byte	ordinalAuditStatus	OPTIONAL for MRTMs and MLTMs.
TPM_DIRVALUE	authDIR	OPTIONAL for MRTMs and MLTMs.
BYTE*	rngState	This field however is REQUIRED to be internally present because a RNG is required to be present. There is however no requirement for this field to be readable or writeable by any commands.
TPM_FAMILY_TABLE	familyTable	OPTIONAL for MRTMs and MLTMs.
TPM_DELEGATE_TABLE	delegateTable	OPTIONAL for MRTMs and MLTMs.
TPM_NONCE	ekReset	OPTIONAL for MRTMs and MLTMs.
UINT32	maxNVBufSize	OPTIONAL for MRTMs and MLTMs.
UINT32	lastFamilyID	OPTIONAL for MRTMs and MLTMs.
UINT32	noOwnerNVWrite	OPTIONAL for MRTMs and MLTMs.
TPM_CMK_DELEGATE	restrictDelegate	OPTIONAL for MRTMs and MLTMs.
TPM_DAA_TPM_SEED	tpmDAASeed	OPTIONAL for MRTMs and MLTMs.

1 **Table 1. TPM_PERMANENT_DATA changes for a MTM**

5.6 MTM_STANY_FLAGS

Start of informative comment:

The MTM_STANY_FLAGS structure houses additional flags that are initialized by TPM_Init when the MTM boots.

End of informative comment.

Definition

```
typedef struct MTM_STANY_FLAGS_STRUCT {
    TPM_TAG   tag;
    BOOL loadVerificationRootKeyEnabled;
} MTM_STANY_FLAGS;
```

Parameters

Type	Name	Description
TPM_TAG	Tag	This MUST be set to MTM_STANY_FLAGS.
BOOL	loadVerificationRootKeyEnabled	If set to FALSE then all loaded TPM_VERIFICATION_KEYs must be verified against an integrity check. If set to TRUE then TPM_VERIFICATION_KEYs can be loaded without integrity checks being performed.

Descriptions

1. There is only a single instance of this structure in a MTM. A member *field* of this structure instance is referenced as MTM_STANY_FLAGS->*field* in this specification.
2. The loadVerificationRootKeyEnabled flag is set by TPM_Init.
3. The *loadVerificationRootKeyEnabled* flag can be set after TPM_Init by using MTM_LoadVerificationRootKeyDisable. This flag is used by MTM_LoadVerificationKey and MTM_SetVerifiedPCRs.

6. Monotonic Counters

Start of informative comment:

This section describes the use of monotonic counters in the mobile phone platform.

End of informative comment.

The term monotonic counter or counter in this section refers to monotonic counters as defined in the TCG Main Specifications [2][3] and to the special counterBootstrap counter which is defined in this specification.

The following counters MUST be implemented if the MTM_VerifyRIMCert, MTM_InstallRIM, MTM_LoadVerificationKey, MTM_IncrementBootStrapCounter or MTM_VerifyRIMCertAndExtend are implemented:

- counterRIMProtect
- counterBootstrap

The MRTMs MUST therefore support the above counters. The MLTM must support them only if it implements any of the commands that use them.

The **counterRIMProtect** is required for protecting internal RIM certificates against reflash attacks.

The MRTM must additionally provide a second counter for reflash protection of the firmware image that is the initial image that is executed during the bootstrap process. This counter is called **counterBootstrap**.

TPM_RIM_CERTIFICATE and TPM_VERIFICATION_KEY structures contain a field *counterReference* that can be used to bind the RIM Certificate to a counter on the MTM. This counter may either be compared against **counterRIMProtect** or **counterBootstrap**.

A third counter, counterStorageProtect, is defined to allow use by the user for protecting secure storage. This counter is not used by any of the commands defined in this specification, but is required to be present both on a MRTM and MLTM.

In this specification an MTM is **not** REQUIRED to implement the minimum amount of four monotonic counters defined in [2] and [3] in addition to the counters defined in this section. This is because there is no use case currently seen to require additional counters and this specification intends to optimize for a constrained embedded systems style environment.

6.1 CounterRIMProtect

The counterRIMProtect is a counter that is implemented using regular TCG TPM counters and is used to protect internal RIM certificates against reflash attacks.

The counterRIMProtect is not an explicitly reserved counter in the MTM. However, the TPM_COUNT_ID of the counterRIMProtect is stored in MTM_PERMANENT_DATA->counterRimProtectId. In a MRTM this field MUST be defined and it MUST reference a valid and existing counter. This counter SHOULD NOT be releasable via TPM_ReleaseCounter or TPM_ReleaseCounterOwner.

The TPM_SECRET used to control access to increase the counterRIMProtect counter MUST be equal to the TPM_SECRET used to authorize MTM_InstallRIM operations. This is currently defined to be *verificationAuth*.

This specification does NOT require that the counter counterRIMProtect be able to run up to 2^32-1 (as is required by the [2][3] for counters). Due to feasibility concerns this counter MAY have a maximum value of 4095 (2^12 –1). This implies that the counter SHOULD NOT be incremented more often than once per day and SHOULD NOT be incremented more than once per boot cycle.

In case some RIM certificates shall be revoked, the counter counterRIMProtect needs to be increased after the new RIM certificate(s) are installed. This should be done after replacement RIM certificate(s) are installed. All existing RIM certificates that will be required in the future must be re-installed using MTM_InstallRIM.

6.2 CounterBootstrap

This counter is intended for verifying the validity of the first executable image. This counter is synchronized directly to an external RIM Certificate. This way the counter can directly be synchronized to a firmware image. In case a new firmware image is installed, this image would also contain a RIM Certificate with an increased counter value (reflecting an increase in the version number).

This counter is incremented via a special new command MTM_IncrementBootstrapCounter and read via TPM_GetCapability. TPM_GetCapability does not require any authorization. MTM_IncrementBootstrapCounter takes as input a RIM Certificate. The TPM_VERIFICATION_KEY used to verify the RIM Certificate in MTM_IncrementBootstrapCounter MUST have the *incrementBootstrapCounter* bit set to TRUE. IF the RIM Certificate is VALID and the TPM_VERIFICATION_KEY has said bit set THEN the MTM_PERMANENT_DATA->counterBootstrap value has its value incremented to the value in the TPM_RIM_CERTIFICATE->counterReference->counterValue. If TPM_RIM_CERTIFICATE->counterReference->counterValue is less or equal to MTM_PERMANENT_DATA->counterBootstrap OR the RIM certificate is not valid OR the verification key does not have the incrementBootstrapCounter bit set THEN the MTM_IncrementBootstrapCounter does nothing.

The MTM_PERMANENT_DATA->counterBootstrap counter MUST be able to increase to a value of at least 31 (2^5 – 1). It is NOT REQUIRED that this field is able to have greater values, although it is RECOMMENDED. Due to the low maximal value of this counter it should ONLY be increased when absolutely necessary to prevent the ability to run or install outdated firmware images. The low maximum value of the counter enables implementation using unary integers. This implies the ability to implement using one-time-programmable bits in hardware.

6.3 counterStorageProtect

This specification defines an additional counter called counterStorageProtect that is REQUIRED for both a MLTM and a MRTM. The usageAuth of this counter MUST be MTM_PERMANENT_DATA->verificationAuth (which would equal ownerAuth for an MLTM).

This specification does NOT require that the counter counterStorageProtect be able to run up to 2^32-1 (as is required by the [2][3] for counters). Due to feasibility concerns this counter MAY have a maximum value of 4095 (2^12 –1). This implies that the counter SHOULD NOT be incremented more often than once per day and SHOULD NOT be incremented more than once per boot cycle.

This counter is not used by any commands defined in this specification.

6.4 Strength-of-Function of Monotonic Counters

Start of informative comment:

The TPM specifications [1][2][3] require that the counters (used via TPM_IncrementCounter, TPM_ReadCounter, TPM_CreateCounter and TPM_ReleaseCounter) be stored in shielded locations and that the only feasible way of manipulating the counter values is via the use of the TPM_IncrementCounter or TPM_ReleaseCounter command.

End of informative comment.

This specification relaxes this requirement. After careful consideration of the currently available implementation options, this specification currently does NOT REQUIRE compliant implementations to store counter values in locations that are non-volatile and physically shielded (stored in tamper-resistant or tamper-evident hardware). The counter values SHOULD be stored in physically shielded locations. The counter-values MUST be stored in non-volatile storage. The counter-values SHOULD be stored in locations that are shielded from software executing outside the context of the MTM.

The above requirements also holds for the counterBootstrap and counterStorageProtect counter as defined in this specification.

The TCG MPWG intends to tighten this requirement to the level of the TPM specifications immediately when it becomes feasible to implement such counters in mobile phones.

7. MTM Commands for Local Verification

7.1 Overview

The commands defined in this specification are:

- MTM_InstallRIM
- MTM_LoadVerificationKey
- MTM_LoadVerificationRootKeyDisable
- MTM_VerifyRIMCert
- MTM_VerifyRIMCertAndExtend
- MTM_IncrementBootstrapCounter
- MTM_SetVerifiedPCRSelection

All of them are REQUIRED for a MRTM.

All of them are OPTIONAL for a MLTM.

This specification considers only the case that ALL of these commands are implemented or NONE of them are implemented. This specification has not considered cases where only a subset of the above commands is implemented.

7.2 MTM_InstallRIM

Start of informative comment:

This command generates internal RIM certificates. The assumption is that the common use case would be to generate an internal RIM certificate from an external RIM certificate. The RIM certificates are expected to be verified using MTM_VerifyRIMCert or MTM_VerifyRIMCertAndExtend.

When running MTM_InstallRIM, there is no requirement for the MTM to itself verify the integrityData on the input rimCert (i.e. the integrity data accompanying the "external" RIM certificate). If the command parameters verify successfully, the MTM can assume that the relevant RIM has already been validated and authorized by the party that owns the verificationAuth data.

The internal RIM certificate can be thought of as a "ticket" i.e. a structure created by the MTM, which the same MTM can identify later and use as evidence that the RIM was authorized. This behaviour is comparable to the command TPM_CMK_CreateTicket used in certified migration [3].

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	Tag	TPM_TAG_RQU_Auth1_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_InstallRIM
4	4	2S	4	UINT32	rimCertSize	Size of rimCert data
5	<>	3S	<>	TPM_RIM_CERTIFICATE	rimCertIn	Data to be used for internal RIM certificate
6	4			TPM_AUTHHANDLE	authHandle	The authorization session handle used for authorization
		2H1	20	TPM_NONCE	authLastNonceEven	Even nonce previously generated by MTM to cover inputs
7	20	3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
8	1	4H1	1	BOOL	continueAuthSession	The continue use flage for the authorization session handle.
9	20			TPM_AUTHDATA	authData	The authorization session digest for inputs. HMAC key: verificationAuth

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	Tag	TPM_TAG_RSP_Auth1_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_InstallRIM
4	4	2S	4	UINT32	rimCertSize	Size of rimCert data
5	<>	4S	<>	TPM_RIM_CERTIFICATE	rimCertOut	An internal RIM certificate
6	20	2H1	20	TPM_NONCE	nonceEven	Even nonce newly generated by MTM to cover outputs
		3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
7	1	4H1	1	BOOL	continueAuthSession	Continue use flag, TRUE if handle is still active
8	20			TPM_AUTHDATA	resAuth	The authorization session digest for the returned parameters. HMAC key: verificatioAuth

Action

The MTM SHALL perform the following steps:

1. Check that the input parameter rimCertIn is syntactically correct. On error return TPM_BAD_PARAMETER.
2. If the command parameters are not authenticated correctly using verificationAuth return TPM_AUTHFAIL
3. Set TPM_ACTUAL_COUNT cntProtect to the result of TPM_ReadCounter for the counter id MTM_PERMANENT_DATA->counterRimProtectId.
4. If the TPM_ReadCounter fails (e.g. counter does not exist) then return TPM_BAD_COUNTER
5. Set rimCertOut to rimCertIn
6. If rimCertIn->counterReference->counterSelection != MTM_COUNTER_SELECT_NONE
    a. Set rimCertOut->counterReference->counterValue to the value cntProtect + 1.
    b. Set rimCertOut->counterReference->counterSelection = MTM_COUNTER_SELECT_RIMPROTECT

Start of informative comment:

Note there is no verification of rimCertIn before starting to create rimCertOut.

The reason for setting the counterValue in the rimCertOut to cntProtect + 1 is to facilitate the use-case of upgrading a complete set of RIM Certificates to a new counter value. For example, suppose the current counter value is 2, but needs to be upgraded to 3. The MTM_InstallRIM command can be used to re-create internal RIM certificates using the new counter value (i.e. 3). Then, only when a full set are created is the counter incremented from 2 to 3 (using either TPM_IncrementCounter or MTM_IncrementBootstrapCounter) thereby invalidating the old set of RIM Certificates.

It would be very risky to increment the counter **before** creating a new set of RIM Certificates. If the creation process aborted for any reason (such as mobile battery running out of power), the device might find it has no valid RIMs on next power-up, and so is unable to boot.

End of informative comment.

7. Else
    a. Set rimCertOut->counterReference->counterValue to 0
8. Set rimCertOut->parentId = TPM_VERIFICATION_KEY_ID_INTERNAL
9. Set rimCertOut->integrityCheckSize to 0
10. Generate the integrityCheckData for rimCertOut using the HMAC key MTM_PERMANENT_DATA->internalVerificationKey.
11. Set rimCertOut->integrityCheckData to the generated integrityCheckData
12. Set rimCertOut->integrityCheckSize to the size of the generated integrityCheckData
13. Return TPM_SUCCESS

7.3 MTM_LoadVerificationKey

Start of informative comment:

This command is used to load one Verification Key into the MTM. The command is complicated because the load of a TPM_VERIFICATION_KEY structure can be authorized in any of the following four ways:

- The key is loaded into the MTM before integrity checks have been enabled
- A cryptographic hash (or equivalent) of the key is embedded into the MTM
- The key loading is directly authorized by the MTM Owner
- The key to be loaded is signed by an authentic, authorized and already loaded TPM_VERIFICATION_KEY

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RQU_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_LoadVerificationtKey
4	4	2S	4	TPM_VERIFICATION_KEY_HANDLE	parentKey	Parent key used to verify this key.
5	4	3S	4	UINT32	verificationKeySize	Size of the verificationKey parameter in bytes.
6	<>	4S	<>	TPM_ VERIFICATION_KEY	verificationKey	The Verification Key to be loaded.
7	4			TPM_AUTHHANDLE	authHandle	The authorization session handle used for authorization
		2H1	20	TPM_NONCE	authLastNonceEven	Even nonce previously generated by MTM to cover inputs
8	20	3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
9	1	4H1	1	BOOL	continueAuthSession	The continue use flage for the authorization session handle.
10	20			TPM_AUTHDATA	authData	OPTINALLY the authorization session digest for inputs. HMAC key: ownerAuth

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RSP_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_LoadVerificationKey
4	4	3S	4	TPM_VERIFICATION_KEY_HANDLE	verificationKeyHandle	Handle for the key that was loaded.
5	1	4S	1	BYTE	loadMethod	If the return code is success, shows which method was used to load this verification key
5	20	2H1	20	TPM_NONCE	nonceEven	Even nonce newly generated by MTM to cover outputs
		3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
6	1	4H1	1	BOOL	continueAuthSession	Continue use flag, TRUE if handle is still active
7	20			TPM_AUTHDATA	resAuth	OPTIONALLY The authorization session digest for the returned parameters. HMAC key: ownerAuth

The TPM owner authentication passed to this command is OPTIONAL and it is only possible to provide it if there is a valid owner authentication set. If it is the case that such a secret has been set then it MUST be possible to authorize the loading of TPM_VERIFICATION_KEY objects with that secret.

Action

The MTM SHALL perform the following steps:

1. Check that the input parameter verificationKey is syntactically correct. If verificationKey is not syntactically correct then return TPM_BAD_PARAMETER.
2. Check that there is enough space to store verificationKey inside the MTM. If there is not enough space then return TPM_NOSPACE.
3. If MTM_STANY_FLAGS -> loadVerificationRootKeyEnabled == TRUE
    a. Associate a key handle K1 with the incoming key and store the key so that it can be used by the MTM_VerifyRIMCertAndExtend command. MTM_VerifyRIMCertAndExtend MUST be able to find and access this key until it is unloaded by TPM_FlushSpecific. A key MUST NOT be usable by above-mentioned commands before it has been loaded successfully with this command.
    b. OPTIONAL: If the MTM_PERMANENT_DATA->integrityCheckRootData in undefined, then set it now so that it will verify the integrity of the incoming key (verificationKey) if this key is ever loaded again.

Start of informative comment:

The use-case here is to allow a verification root key to be loaded once without integrity checks but **only** when the MTM is first manufactured or first customized for a particular engine. After this first load, the flag loadVerificationRootKeyEnabled can be set to FALSE, and when the same root key is loaded in future boot cycles, it will be verified using the integrityCheckRootData instead.

End of informative comment.

    c. Set verificationKeyHandle to K1
    d. Set loadMethod to TPM_VERIFICATION_KEY_ROOT_LOAD
    e. Return TPM_SUCCESS
4. Else if MTM_PERMANENT_DATA->integrityCheckRootData is defined AND verifies the integrity of verificationKey
    a. Associate a key handle K1 with the incoming key and store the key so that it can be used by the MTM_VerifyRIMCertAndExtend command. MTM_VerifyRIMCertAndExtend MUST be able to find and access this key until it is unloaded by TPM_FlushSpecific. A key MUST NOT be usable by above-mentioned commands before it has been loaded successfully with this command.
    b. Set verificationKeyHandle to K1
    c. Set loadMethod to TPM_VERIFICATION_KEY_INTEGRITY_CHECK_ROOT_DATA_LOAD
    d. Return TPM_ SUCCESS
5. Else attempt to validate the command and the parameters using TPM Owner Authentication. If the command was owner authenticated
    a. Associate a key handle K1 with the incoming key and store the key so that it can be used by the MTM_VerifyRIMCertAndExtend command. MTM_VerifyRIMCertAndExtend MUST be able to find and access this key until it is unloaded by TPM_FlushSpecific. A key MUST NOT be usable by above-mentioned commands before it has been loaded successfully with this command.
    b. Set verificationKeyHandle to K1
    c. Set loadMethod to TPM_VERIFICATION_KEY_OWNER_AUTHORIZED_LOAD

d. return TPM_ SUCCESS

6. Else if parentKey is NOT defined OR the parentKey is not loaded into the MTM
   a. Return TPM_KEYNOTFOUND
7. Else // A Verifying Key is Found and Defined
   a. Check that TPM_VERIFICATION_KEY_USAGE_SIGN_RIMAUTH is set in parentKey->usageFlags. If not then return TPM_INVALID_KEYUSAGE.
   b. Check that IF TPM_VERIFICATION_KEY_USAGE_INCREMENT_BOOTSTRAP is set in verificationKey->usageFlags then TPM_VERIFICATION_KEY_USAGE_INCREMENT_BOOTSTRAP is also set in parentKey->usageFlags AND if not then return TPM_INVALID_KEYUSAGE.
   c. Check that parentKey->myId == verificationKey->parentId. If not then return TPM_AUTHFAIL.
   d. Check that parentKey does verify verificationKey->integrityCheckData. If not then return TPM_AUTHFAIL.
   e. Check that verificationKey->counterReference->counterSelection <= MTM_COUNTER_SELECT_MAX. If not then return TPM_BAD_COUNTER.
   f. If verificationKey->counterReference->counterSelection == MTM_COUNTER_SELECT_BOOTSTRAP
      i. Check that verificationKey->counterReference->counterValue >= MTM_PERMANENT_DATA->counterBootstrap. If not then return TPM_BAD_COUNTER.
   g. If verificationKey->counterReference->counterSelection == MTM_COUNTER_SELECT_RIMPROTECT
      i. Set cntVal *to* the result of TPM_ReadCounter of counter MTM_PERMANENT_DATA->counterRimProtectId. If TPM_ReadCounter did not return TPM_SUCCESS then return TPM_BAD_COUNTER.
      ii. Check that verificationKey->counterReference->counterValue >= cntVal. If not then return TPM_BAD_COUNTER.
   h. Associate a key handle K1 with the incoming key and store the key so that it can be used by the MTM_VerifyRIMCertAndExtend command. MTM_VerifyRIMCertAndExtend MUST be able to find and access this key until it is unloaded by TPM_FlushSpecific. A key MUST NOT be usable by above-mentioned commands before it has been loaded successfully with this command.
   i. Set verificationKeyHandle to K1
   j. Set loadMethod to TPM_VERIFICATION_KEY_CHAIN_AUTHORIZED_LOAD
   k. Return TPM_SUCCESS.

7.4 MTM_LoadVerificationRootKeyDisable

Start of informative comment:

This command disables the functionality to load Verification Root Keys. After it has been called, a MTM must validate integrity checks, parent key usage flags and so on for all TPM_VERIFICATION_KEYs that are passed to MTM_LoadVerificationKey.

In all MLTMs, the flag *loadVerificationRootKeyEnabled* will always be set to FALSE at power-up, and this command will have no effect.

For an MRTM this command is typically only needed at manufacture, or else when the MRTM is first customized for a particular Engine. In both these cases, the safest course is to then set the *loadVerificationRootKeyEnabled* flag to FALSE during all subsequent power-up cycles, thereby ensuring that verification root keys cannot be loaded by unauthorized parties.

***** SECURITY WARNING *****

Setting the flag *loadVerificationRootKeyEnabled* to TRUE on power-up can have a major impact on the boot properties of the Engine in which the MTM is placed. A decision to use that setting, and then use this command to set the flag to FALSE, must be made with great care.

An MRTM *may* be designed so that the *loadVerificationRootKeyEnabled* flag is set to TRUE on each power-up, with the assumption that the RTV will load the necessary verification root key (or keys) before calling this command. In that case, there is a security risk that some entity other than the RTV is able to access the MTM immediately after power-up, in which case the whole boot process could be subverted.

If by design only the RTV code is running at this early stage, and it is trusted, the attack would generally need to involve some form of physical manipulation e.g. hijacking a bus between the RTV and the MTM. Such a physical attack can be resisted in a number of ways:

- The RTV and MTM are implemented in a common unit, with no path between them that can be attacked physically
- The path between the RTV and MTM is physically protected to make insertion/manipulation of traffic extremely difficult, or else expensive (it results in the phone being destroyed)
- The path between the RTV and MTM is cryptographically protected.

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RQU_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4			TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_LoadVerificationRootKeyDisable

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RSP_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4			TPM_RESULT	returnCode	The return code of the operation.

Action

The MTM SHALL perform the following steps:

1. Set MTM_STANY_FLAGS -> loadVerificationRootKeyEnabled to FALSE
2. OPTIONAL: Set the TPM_VERIFICATION_KEY_ROOT_LOAD flag in MTM_PERMANENT_DATA -> loadVerificationKeyMethods to zero, and ensure that loadVerificationRootKeyEnabled is set to FALSE on all subsequent power-up cycles
3. Return TPM_SUCCESS.

7.5 MTM_VerifyRIMCert

Start of informative comment:

This command is used to verify an internal or external RIM certificate. Note that this command does NOT check that the TPM_PERMANENT_DATA->pcrAttrib[] array is in a required state as defined by TPM_RIM_CERTIFICATE->state. This function is intended for diagnostic and management purposes, especially for internal RIM certificates. In these cases the caller can perform the check of the required state if desired e.g. by using TPM_PCRRead.

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RQU_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4			TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_VerifyRIMCert
4	4			UINT32	rimCertSize	The size of rimCert parameter in bytes.
5	<>			TPM_RIM_CERTIFICATE	rimCert	An internal or external RIM certificate.
6	4			TPM_VERIFICATION_KEY_HANDLE	rimKey	Key Handle for the verification key to be used. Use NULL if the verification key for internal RIM Certs is to be used.

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RSP_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4			TPM_RESULT	returnCode	The return code of the operation.

Action

The MTM SHALL perform the following steps:

1. Check that the input parameter rimCert is syntactically correct. If not then return TPM_BAD_PARAMETER.
2. Check that rimCert->parentId does not equal TPM_VERIFICATION_KEY_ID_NONE. If it does then return TPM_AUTHFAIL.
3. If rimCert->parentId = TPM_VERIFICATION_KEY_ID_INTERNAL
    a. Check the rimCert->integrityCheckData using the HMAC key in MTM_PERMANENT_DATA->internalVerificationKey
    b. If this check fails return TPM_AUTHFAIL
4. Else
    a. Check that rimKey is defined. If not then return TPM_KEYNOTFOUND.
    b. Check that TPM_VERIFICATION_KEY_USAGE_SIGN_RIMCERT is set in rimKey->usageFlags. If not then return TPM_INVALID_KEYUSAGE.
    c. Check that rimCert->parentId == rimKey->myId. If not then return TPM_AUTHFAIL.
    d. Check that rimCert->integrityCheckData verifies using rimKey->keyData. If not then return TPM_AUTHFAIL.

5. Check that rimCert->counterReference->counterSelection <= MTM_COUNTER_SELECT_MAX. If not return TPM_BAD_COUNTER.
6. If rimCert->counterReference->counterSelection == MTM_COUNTER_SELECT_BOOTSTRAP
    a. Check that rimCert->counterReference->counterValue >= MTM_PERMANENT_DATA->counterBootstrap. If not return TPM_BAD_COUNTER.
7. If rimCert->counterReference->counterSelection == MTM_COUNTER_SELECT_RIMPROTECT
    a. Set cntVal to the result of TPM_ReadCounter of counter MTM_PERMANENT_DATA->counterRimProtectId. If TPM_ReadCounter DID NOT return TPM_SUCCESS then return TPM_BAD_COUNTER.
    b. Check that rimCert->counterReference->counterValue >= cntVal. If not then return TPM_BAD_COUNTER.
8. Return TPM_SUCCESS.

7.6 MTM_VerifyRIMCertAndExtend

Start of informative comment:

This command is used to verify and to extend the RIM given in the RIM certificate in to a PCR given in the RIM certificate. The command definition is optimized to avoid three separate commands being called in close sequence: loading a RIM_Cert into a MTM; performing an extend while comparing the extended value against the loaded RIM_Cert; flushing the loaded RIM_Cert.

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RQU_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4			TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_VerifyRIMCertAndExtend
4	4			UINT32	rimCertSize	The size of the rimCert parameter in bytes.
5	<>			TPM_RIM_CERTIFICATE	rimCert	RIM certificate to be verified.
6	4			TPM_VERIFICATION_KEY_HANDLE	rimKey	Key Handle for the verification key to be used. Use NULL if the verifivation key for internal RIM Certs is to be used.

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RSP_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4			TPM_RESULT	returnCode	The return code of the operation.
4	20			TPM_PCRVALUE	outDigest	THE PCR value after the execution of the command.

Action

The MTM SHALL perform the following steps:

1. Check that the input parameter rimCert is syntactically correct. If not then return TPM_BAD_PARAMETER.
2. Check that rimCert->parentId does not equal TPM_VERIFICATION_KEY_ID_NONE. If not then return TPM_AUTHFAIL
3. If rimCert->parentId = TPM_VERIFICATION_KEY_ID_INTERNAL
    a. Check the rimCert->integrityCheckData using the HMAC key in MTM_PERMANENT_DATA->internalVerificationKey. If not then return TPM_AUTHFAIL.
4. Else
    a. Check that rimKey is defined. If not then return TPM_KEYNOTFOUND.
    b. Check that TPM_VERIFICATION_KEY_USAGE_SIGN_RIMCERT is set in rimKey->usageFlags. If not then return TPM_INVALID_KEYUSAGE.
    c. Check that rimCert->parentId = rimKey->myId. If not then return TPM_AUTHFAIL.
    d. Check that rimCert->integrityCheckData verifies using rimKey->keyData. If not then return TPM_AUTHFAIL.
5. Check that rimCert->counterReference->counterSelection <= MTM_COUNTER_SELECT_MAX. If not then return TPM_BAD_COUNTER.

6. If rimCert->counterReference->counterSelection == MTM_COUNTER_SELECT_BOOTSTRAP
    a. Check that rimCert->counterReference->counterValue >= MTM_PERMANENT_DATA->counterBootstrap. If not then return TPM_BAD_COUNTER.
7. If rimCert->counterReference->counterSelection == MTM_COUNTER_SELECT_RIMPROTECT
    a. Set cntVal to the result of TPM_ReadCounter of counter MTM_PERMANENT_DATA->counterRimProtectId. If TPM_ReadCounter did not return TPM_SUCCESS then return TPM_BAD_COUNTER.
    b. Check that If rimCert->counterReference->counterValue >= cntVal. If not then return TPM_BAD_COUNTER.
8. If rimCert->state->pcrSelection has at least one bit set
    a. Let comp1 be the TPM_PCR_INFO_SHORT corresponding to the current PCR state in TPM_PERMANENT_DATA->pcrAttrib[] for the PCRs selected in rimCert->state->pcrSelection
    b. Check that comp1 == rimCert->state. If not then return TPM_WRONGPCRVAL.
9. Extend PCR rimCert->measurementPcrIndex with the measurement rimCert->measurementValue as defined by the TPM_Extend command [3].
10. Set outDigest to the value of PCR index rimCert->measurementPcrIndex
11. Return TPM_ SUCCESS

7.7 MTM_IncrementBootstrapCounter

Start of informative comment:

This command is used to increment the MTM_PERMANENT_DATA->counterBootstrap.

The counterBootstrap counter is distinguished from other monotonic counters, and can only be incremented using the signed authorization of a privileged RIM_Auth. The RIM_Auth provides this authorization by issuing a RIM_Cert containing the incremented counter value. If the command is called using an irrelevant RIM_Cert (e.g. one containing a different counterSelection from counterBootstrap, or a selection of counterBootstrap but with the existing counter value) then the command terminates with no effect.

Once incremented in an MTM, RIM_Certs (and RIM_Auth TPM_Verification_Keys) signed with a lower value of the bootstrap counter will no longer be accepted as valid by the MTM. Thus this counter provides a means of revoking RIM_Certs that might be used when a device boots after a reflash.

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RQU_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4			TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_IncrementBootstrapCounter
4	4			UINT32	rimCertSize	The size of the rimCert parameter in bytes.
5	<>			TPM_RIM_CERTIFICATE	rimCert	A RIM certificate.
6	4			TPM_VERIFICATION_KEY_HANDLE	rimKey	Key Handle for the verification key to be used.

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RSP_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4			TPM_RESULT	returnCode	The return code of the operation.

Action

The MTM SHALL perform the following steps:

1. Check that the input parameter rimCert is syntactically correct. If not then return TPM_BAD_PARAMETER.
2. Check that rimKey is a handle for a valid and defined TPM_VERIFICATION_KEY. If not then return TPM_KEYNOTFOUND.
3. Check TPM_VERIFICATION_KEY_USAGE_SIGN_RIMCERT is set in rimKey->usageFlags AND TPM_VERIFICATION_KEY_USAGE_INCREMENT_BOOTSTRAP is set in rimKey->usageFlags. If not then return TPM_INVALID_KEYUSAGE.
4. Check that rimKey->myId == rimCert->parentId. If not then return TPM_AUTHFAIL.
5. Check that rimCert->integrityCheckData verifies using rimKey->keyData. If not then return TPM_AUTHFAIL.
6. Check that rimCert->counterReference->counterSelection <= MTM_COUNTER_SELECT_MAX. If not then return TPM_BAD_COUNTER.
7. If rimCert->counterReference->counterSelection == MTM_COUNTER_SELECT_BOOTSTRAP

1 a. Check that rimCert->counterReference->counterValue >= MTM_PERMANENT_DATA-
2 >counterBootstrap. If not then return TPM_BAD_COUNTER.
3 b. If rimCert->counterReference->counterValue > MTM_PERMANENT_DATA-
4 >counterBootstrap
5 i. Set MTM_PERMANENT_DATA->counterBootstrap to rimCert->counterReference-
6 >counterValue
7 8. Return TPM_SUCCESS
8

7.8 MTM_SetVerifiedPCRSelection

Start of informative comment:

This command is used to set the MTM_PERMANENT_DATA->verifiedPCRs, either with owner authorization, or in the case that the MTM_STANY_FLAGS.loadVerificationRootKeyEnabled is TRUE.

***** SECURITY WARNING *****

This command has a major impact on the boot properties of the Engine in which the MTM is placed, and it must be used with great care.

For an MLTM, the command is typically called after a local user has first taken ownership; it cannot be called again without the owner's authorization. For an MRTM the command is typically called at manufacture, or else when the MRTM is first customized for a particular Engine. In both cases, the safest course is to then set the *loadVerificationRootKeyEnabled* flag permanently to FALSE, ensuring that the command cannot be called again by unauthorized parties.

However, an MRTM *may* be designed so that the loadVerificationRootKeyEnabled flag is set to TRUE on power-up, with the assumption that the RTV sets the selection of *verifiedPCRs* before then calling the command MTM_LoadVerificationRootKeyDisable. **The informative comment for MTM_LoadVerificationRootKeyDisable describes some important security concerns that can arise with this model, and the additional steps that must be taken to resolve them.**

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RQU_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_SetVerifiedPCRSelection
4	<>	2S	<>	TPM_PCR_SELECTION	VerifiedSelection	Set of PCRs that can only be extended with MTM_VerifyRIMCertAndExtend
5	4			TPM_AUTHHANDLE	authHandle	The authorization session handle used for authorization
		2H1	20	TPM_NONCE	authLastNonceEven	Even nonce previously generated by MTM to cover inputs
6	20	3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
7	1	4H1	1	BOOL	continueAuthSession	The continue use flage for the authorization session handle.
8	20			TPM_AUTHDATA	authData	The authorization session digest for inputs. HMAC key: ownerAuth

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RSP_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: MTM_ORD_SetVerifiedPCRSelection
4	20	2H1	20	TPM_NONCE	nonceEven	Even nonce newly generated by MTM to cover outputs
		3H1	20	TPM_NONCE	nonceOdd	Nonce generated by system associated with authHandle
5	1	4H1	1	BOOL	continueAuthSession	Continue use flag, TRUE if handle is still active
6	20			TPM_AUTHDATA	resAuth	OPTIONALLY The authorization session digest for the returned parameters. HMAC key: ownerAuth

The TPM owner authentication passed to this command is OPTIONAL and it is only possible to provide it if there is a valid owner authentication set. If it is the case that such a secret has been set then it MUST be possible to authorize the setting of MTM_PERMANENT_DATA->verifiedPCRs with that secret.

Action

The MTM SHALL perform the following steps:

1. Validate the command parameters using TPM Owner secret. If there is no valid ownerAuth then assume this authorization has failed.
2. If MTM_STANY_FLAGS->loadVerificationRootKeyEnabled == FALSE AND the owner authorization has failed
    a. RETURN TPM_FAIL
3. Else if any of the PCRs in VerifiedSelection have a localityModifier set
    a. Return TPM_FAIL
4. Else
    a. Set MTM_PERMANENT_DATA->verifiedPCRs to VerifiedSelection
    b. Return TPM_SUCCESS

7.9 MTM-specific Ordinals

Start of informative comment

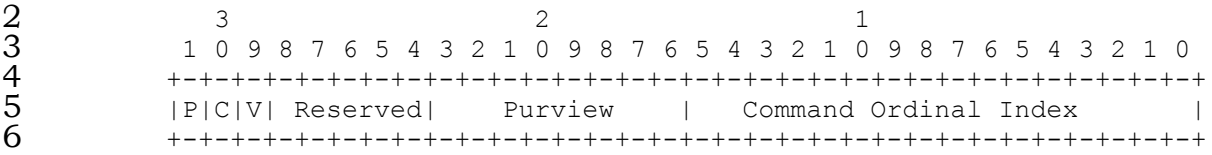
The command ordinals provide the index value for each command. The following list contains the index value and other information relative to the ordinal.

TPM commands are divided into three classes: Protected/Unprotected, Non-Connection/Connection related, and TPM/Vendor.

This section contains only the MTM specific ordinals.

End of informative comment

Ordinals are 32 bit values. The upper byte contains values that serve as flag indicators, the next byte contains values indicating what committee designated the ordinal, and the final two bytes contain the Command Ordinal Index.



Where:

P is Protected/Unprotected command. When 0 the command is a Protected command, when 1 the command is an Unprotected command.

C is Non-Connection/Connection related command. When 0 this command passes through to either the protected (TPM) or unprotected (TSS) components.

V is TPM/Vendor command. When 0 the command is TPM defined, when 1 the command is vendor defined.

All reserved area bits are set to 0.

The following masks are created to allow for the quick definition of the commands

Value	Event Name	Comments
0x00000000	TPM_PROTECTED_COMMAND	TPM protected command, specified in main specification
0x80000000	TPM_UNPROTECTED_COMMAND	TSS command, specified in the TSS specification
0x40000000	TPM_CONNECTION_COMMAND	TSC command, protected connection commands are specified in the main
0x20000000	TPM_VENDOR_COMMAND	Command that is vendor specific for a given TPM or TSS.

If a command is tagged from the audit column the default state is that use of that command SHALL be audited. Otherwise, the default state is that use of that command SHALL NOT be audited.

Column	Column Values	Comments and valid column entries
AUTH2	x	Does the command support two authorization entities, normally two keys
AUTH1	x	Does the commands support an single authorization session
RQU	x	Does the command execute without any authorization
Optional MRTM	x	Is the command optional for MRTM
Optional MLTM	x	Is the command optional for MLTM
No Owner	x	Is the command executable when no owner is present
PCR Use Enforced	x	Does the command enforce PCR restrictions when executed
Physical presence	P, P*, O, T, T*, A*	P = The command requires physical presence P* = The command may require physical presence O = The command requires physical presence or operator authorization T = The command requires physical presence or TPM owner authorization T* = The NV space may be configured to require physical presence in addition to TPM owner authorization A* = The NV space may be configured to require physical presence in addition to other entity authorization
Audit	X, N	Is the default for auditing enabled N = Never the ordinal is never audited X = Auditing is enabled by default
Duration	S, M, L	What is the expected duration of the command, S = Short implies no asymmetric cryptography M = Medium implies an asymmetric operation L = Long implies asymmetric key generation

The following table is normative, and is the overriding authority in case of discrepancies in other parts of this specification.

	TPM_PROTECTED_ORDINAL +	Complete ordinal	AUTH2	AUTH1	RQU	Optional MRTM	Optional MLTM		No Owner	Physical Presence	PCR Use enforced	Audit	Duration
MTM_ORD_InstallRIM	66	0x00000042		X			X					X	S or M
MTM_ORD_LoadVerificationKey	67	0x00000043		X	X		X					X	S or M
MTM_ORD_LoadVerificationRootKeyDisable	68	0x00000044			X		X					X	S
MTM_ORD_VerifyRIMCert	69	0x00000045			X		X					X	S or M
MTM_ORD_VerifyRIMCertAndExtend	72	0x00000048			X		X					X	S or M
MTM_ORD_IncrementBootstrapCounter	73	0x00000049			X		X					X	S or M
MTM_ORD_SetVerifiedPCRSelection	74	0x0000004A		X	X		X					X	S
RESERVED	75	0x0000004B											
RESERVED	76	0x0000004C											
RESERVED	77	0x0000004D											
RESERVED	78	0x0000004E											
RESERVED	79	0x0000004F											

8. Differences to a TPM V1.2

Start of informative comment:

This section lists differences or extensions to commands or capabilities defined in [3] that are required to be implemented by a MTM.

End of informative comment.

8.1 TPM_GetCapability

This specification requires an extension to TPM_GetCapability for MTMs. The following additional TPM_CAPABILITY_AREA is defined:

Value	Capability Name	Sub cap	Comments
0x0000000A	TPM_CAP_MTM_PERMANENT_DATA	*dataSelect* parameter	Returns public-readable portions of the MTM_PERMANENT_DATA. Legitimate values of the dataSelect parameter are defined below.

Action

If the capArea is TPM_CAP_MTM_PERMANENT_DATA, the MTM SHALL perform the following steps:

1. Check dataSelect to be a valid parameter (see Table below). If not then return TPM_BAD_PARAMETER.
2. If dataSelect is 0x0000 0001 AND MTM_PERMANENT_DATA->aik is undefined
    a. Return TPM_FAIL
3. Set resp to the appropriate value
4. Set respSize to the size of resp
5. Return TPM_SUCCESS

Otherwise the MTM SHALL execute TPM_GetCapability, as defined in [3]. In the case that a capability area or subCap value is requested which is not supported by the MTM, the MTM SHALL return the error code TPM_BAD_PARAMETER.

Defined values for dataSelect

Value of dataSelect parameter	Data returned in resp parameter
0x0000 0001	MTM_PERMANENT_DATA->aik (public part)
0x0000 0002	MTM_PERMANENT_DATA->verifiedPCRs
0x0000 0003	MTM_PERMANENT_DATA->counterBootstrap
0x0000 0004	MTM_PERMANENT_DATA->counterRimProtectId
0x0000 0005	MTM_PERMANENT_DATA->counterStorageProtectId
0x0000 0006	MTM_PERMANENT_DATA->specMajor
0x0000 0007	MTM_PERMANENT_DATA->specMinor
0x0000 0008	MTM_PERMANENT_DATA->loadVerificationKeyMethods

8.2 TPM_Extend

This specification requires a change to TPM_Extend for MTMs implementing MTM_VerifyRIMCertAndExtend. The following pre-amble action MUST be executed before each TPM_Extend.

ACTION

1. If the *pcrNum* parameter is set in MTM_PERMANENT_DATA->verifiedPCRs THEN
    a. Return TPM_BAD_LOCALITY
2. Else
    a. Execute TPM_Extend as defined in [3].

8.3 TPM_Init

This specification requires that TPM_Init MUST also initialize the structure MTM_STANY_FLAGS. The initialization of this structure is according to policy, but the policy MUST be reflected in the flag TPM_VERIFICATION_KEY_ROOT_LOAD of MTM_PERMANENT_DATA->loadVerificationKeyMethods.

If the MTM is a MLTM with ownerAuth set, then MTM_STANY_FLAGS->loadVerificationRootKeyEnabled MUST be set to FALSE by TPM_Init.

8.4 TPM_PCR_Reset

This specification requires a change to TPM_PCR_Reset. The following pre-amble action MUST be executed before each TPM_PCR_Reset.

ACTION

1. If the *pcrNum* parameter is set in MTM_PERMANENT_DATA->verifiedPCRs THEN
    a. Return TPM_FAIL
2. Else
    a. Execute TPM_PCR_Reset as defined in [3].

8.5 TPM_ResetLockValue

The TPM_ResetLockValue command is OPTIONAL for the MRTM because this specification does not assume that owner authorization data for the MRTM is present on the platform. Nevertheless, there MUST be a mechanism to mitigate dictionary attacks AND a mechanism to reset this mitigation mechanism. TPM_ResetLockValue is an acceptable mechanism for a MRTM, but implementations of a MRTM compliant with this specification may also use vendor specific mechanisms instead of TPM_ResetLockValue. These proprietary mechanisms can include time outs, reboots, and periodical resets of the mitigation mechanism.

TPM_ResetLockValue is REQUIRED for a MLTM.

8.6 Physical Presence

The MRTM is designed for a scenario where the owner (as defined by TPM_TakeOwnership) is a remote party. In this case physical presence authorization is contrary to what is desired and therefore physical presence authorization MUST NOT be supported in the MRTM.

Physical presence authorization MUST be supported in a MLTM. An assertion of physical presence for a MLTM MAY be provided via trusted software that has been verified against a configuration secured using a MRTM.

8.7 Localities

Proofs of locality as specified in the TPM Main Specification [1] MAY be supported in MTMs; if localities are not supported the field TPM_STANY_FLAGS->localityModifier MUST always be zero. Nevertheless, localities MUST NOT be used whenever verified PCRs are involved. A PCR may be a verified PCR (have its index bit set in the MTM_PERMANENT_DATA->verifiedPCRs) OR that PCR may have a localityModifier set. A PCR can of course also have no locality bits set and not be a verified PCR. However, a PCR MUST NOT be a verified PCR AND have a localityModifier set. Especially, the TPM_PCR_Reset command MUST NOT work for verified PCRs.

Verified Extends allow one to conclusively check from a verified PCR whether an event has been recorded into the same or another verified PCR or not. Allowing TPM_PCR_Reset for verified PCRs would prohibit this.

8.8 Random Number Generation Requirements

Start of informative comment:

The TPM Design Principles specification [1] requires that the random number generation also save state in a non-volatile shielded location over a power-down.

The criteria for the PRNG state register in [1] are as follows:

A) The state register MUST be non-volatile

B) The update function to the state register is a TPM protected-capability

C) The primary input to the update function SHOULD be the entropy collector.

End of informative comment.

This specification aims to avoid requiring non-volatile shielded storage that can be written to at run-time and therefore the requirement "A" above is not applicable. Implementations compliant with this specification may use the following three criteria instead of the above three for the PRNG state register.

a. The state register MUST be non-volatile OR the state register MUST be initialized at power-on by the entropy collector to contain at least 128 random bits.
b. The update function to the state register is a TPM protected-capability.
c. The primary input to the update function SHOULD be the entropy collector.

8.9 MakeIdentity and ActivateIdentity

A MRTM Implementation MUST support using the MTM_PERMANENT_DATA->verificationAuth as the authorization secret in place of the owner secret. Concretely, this implies that the HMAC secret used to create the input "*ownerAuth*" to both MakeIdentity and ActivateIdentity can be the *verificationAuth* secret. The HMAC secret used to create the authentication of the outputs TPM_MakeIdentity and TPM_ActivateIdentity commands *(*the *resAuth field)* MUST be the same that was used to authorize the inputs.

A MRTM is NOT REQUIRED to accept the owner secret for authorizing TPM_MakeIdentity nor TPM_ActivateIdentity commands in the case that it accepts *verificationAuth*. For example the owner secret might be undefined in the MRTM.

In the case where a MTM could accept both secrets, then the following rules SHALL be used to distinguish which secret is to be used to verify the authorization on the command.

- If TPM_ActivateIdentity is called using an OIAP session, then the ownerAuth secret MUST be used.
- If TPM_MakeIdentity or TPM_ActivateIdentity is called using an OSAP session, then the secret is defined via the *entityType* input to the TPM_OSAP command. The following entity-type field shall be used to define use of the *verificationAuth* secret:

Value	Entity Name	Key Handle	Comments
0x000D	TPM_ET_VERIFICATION_AUTH		The entity is the MTM_PERMANENT_DATA->verificationAuth

8.10 TPM_FlushSpecific

The TPM_FlushSpecific command MUST be able to flush TPM_VERIFICATION_KEY structures from a MRTM that have been loaded with MTM_LoadVerificationKey. The TPM_RESOURCE_TYPE value to use when unloading TPM_VERIFICATION_KEY structures from a MTM is TPM_RT_KEY.

A MTM must be able to distinguish TPM_VERIFICATION_KEY_HANDLE-type handles from TPM_KEY_HANDLE-type handles internally. This implies that an implementation of TPM_FlushSpecific (when given a TPM_RT_KEY resourceType and a TPM_VERIFICATION_KEY_HANDLE OR a TPM_KEY_HANDLE) MUST be able to recognize which type of key to flush based on just the handle value.

8.11 Timing Ticks and Transport Sessions

In the case where an MTM does **not** support Timing Ticks commands (TPM_GetTicks, TPM_TickStampBlob) then the timing tick values that are used in Transport Session commands (TPM_EstablishTransport, TPM_ExecuteTransport, and – optionally – TPM_ReleaseTransportSigned) SHALL all be set to zero.

More formally, wherever a command attempts to use TPM_STANY_DATA -> currentTicks, it SHALL instead use a TPM_CURRENT_TICKS structure in which all sub-components apart from the tag (respectively the number of ticks, the tick rate and the tick nonce) are set to the value 0.

8.12 Ownership in a MLTM

Start of informative comment:

This specification considers the case of implementing the commands in Section 7 in a MLTM with a local owner. This section describes practices for taking and clearing ownership in this case. The intention is that IF ownership is set in this case then the ownerAuth must be usable for controlling which TPM_VERIFICATION_KEY are loadable, which internal RIM Certificates are generated and which PCRs require MTM_VerifyRIMCertAndExtend for extends.

The following practices are recommended in this scenario:

- IF TPM_TakeOwnership is called THEN the TPM counterRIMProtect (monotonic counter identified by MTM_PERMANENT_DATA->counterRimProtectid) SHOULD be incremented to invalidate any previously installed internal RIM certificates.
- IF TPM_TakeOwnership is called THEN MTM_LoadVerificatioRootKeyDisable SHOULD be called.
- IF TPM_OwnerClear is called or any TPM command that executes the steps in TPM_OwnerClear is called THEN the TPM counterRIMProtect (monotonic counter identified by MTM_PERMANENT_DATA->counterRimProtectid) SHOULD be incremented to invalidate any previously installed internal RIM certificates.
- IF the TPM_PERMANENT_DATA->ownerAuth is not valid (defined) and local verification is required THEN there should be external RIM certificates that allow for a “pristine boot” into a state where ownership can be taken.

End of informative comment.

9. Subset of TPM V1.2 Commands Required for a MTM

Start of informative comment:

This section defines the requirements for implementing commands from a TPM v1.2. Commands are either classified as REQUIRED (the MTM MUST implement), OPTIONAL (the MTM MAY implement) or EXCLUDED (the MTM MUST NOT implement).

This specification defines two types of trusted modules. They are respectively referred to as a MRTM and a MLTM.
The first type is intended to be used in a setting with a remote owner and mandatory security on the host platform. The second type is intended for use in a setting with a local owner and either discretionary security or mandatory security under control of the local owner on the host platform.

For this first type, the MTM must have an owner installed, and cannot be de-activated, disabled, or have its owner removed.

The second type is more like a PC TPM: it is intended to be used in a setting that aims to provide opt-in security functionality for the Device. For this second type, the MTM may have an owner or may not. Like a PC TPM, it can be activated, de-activated, enabled or disabled and have an owner installed, removed etc.

The assignment of Required and Optional commands to each type of MTM has been based on the above intended usage. The assignment assumes the MRTM is a valid component for building mandatory security, and the MLTM is a valid component for building discretionary or mandatory security under control of a local owner. This assignment may be revised in future versions of the specification: for instance to make finer distinctions between different classes of MTMs.

End of informative comment.

9.1 Admin Startup and State

The Admin Startup and State command group of groups defines the initialization and startup of the MTM. Note, that a deactivated start must not be allowed for the MRTM since the MRTM is a mandatory security function.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_Init	Required	Required	This is not an actual command. This is just a name for the action performed at power-on
TPM_Startup	Required	Required	'Deactivated' start must not be allowed for a MRTM . 'Save' start may be optionally supported. One possible implementation would be to call TPM_Startup at power-on.
TPM_SaveState	Optional	Optional	Tyoical mobile phones don't have state-saving modes like "suspend" or "hibernate". But some have "sleep modes" to save power and these might pose a risk to volatile data in the MTM. **If** the back-up of such volatile data is expected to be required, but not expected to occur automatically, **then** TPM_SaveState should be supported and used.

9.2 Admin Testing

The Admin Testing command group is required for all MTMs.

As stated in the TPM Main Specification Part 1 [1], a TPM MUST perform a limited self-test after initialization, i.e., it checks a selected subset of TPM commands (TPM_SHA1xxx, TPM_Extend, TPM_Startup, TPM_ContinueSelfTest, TPM_SelfTestFull and TPM_GetCapability). Note, that the Main Specification Part 1 states, that a platform specific specification MUST define the maximum startup self-test time.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_SelfTestFull	Required	Required	After initialisation, the TPM performs a limited self-test, anyhow. This command triggers a full self test
TPM_ContinueSelfTest	Required	Required	This command triggers the completion of the self test, which is started automatically at power-on. The command needs to be performed before most other TPM commands can be executed.
TPM_GetTestResult	Required	Required	The TPM_ContinueSelfTest does not return the result of the self-test, so in order to get the result, this command must be used.

9.3 Admin Opt-in

The Admin Opt-in command group manages the different states of a MTM. It is assumed that a MRTM will always be in the enabled and activated state and that it will not support all the different states given in TPM main spec v1.2. After manufacturing it will be in state S5 (Enabled - Active - Unowned). By using TPM_TakeOwnership the MTM will become owned by the Device Manufacturer. It will then be in state S1 (Enabled - Active - Owned). No further state transitions will then be possible. Therefore, these commands must not be offered by the MRTM.

The MLTM must offer this set of commands.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_SetOwnerInstall	Excluded	Required	Is only applicable if the TPM is unowned.
TPM_OwnerSetDisable	Excluded	Required	This command requires owner authorization and can be used to transition in either enabled or disabled states.
TPM_PhysicalEnable	Excluded, since Physical Presence must not be implemented	Required	
TPM_PhysicalDisable	Excluded	Required	
TPM_PhysicalSetDeactivated	Excluded, since Physical Presence must not be implemented	Required	
TPM_SetTempDeactivated	Excluded	Required	
TPM_SetOperatorAuth	Excluded	Optional	

9.4 Admin Ownership

The Admin Ownership command group manages the ownership of a TPM. It is assumed that the owner of the Device Manufacturer's MTM is never changed during its lifetime. This is not only a reasonable assumption but it is also necessary for security reasons. Hence, the Disable-commands are OPTIONAL or excluded for the MTM of the Device Manufacturer.

For MLTMs the owner must be changeable. Hence, the administrative commands for ownership are required.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_TakeOwnership	Optional	Required	Command might not be necessary if an AIK and SRK are pre-installed.
TPM_OwnerClear	Excluded	Required	See text on "Admin Opt-in" section 9.3: no state transitions are allowed after manufacture.
TPM_ForceClear	Excluded	Required	
TPM_DisableOwnerClear	Optional	Required	
TPM_DisableForceClear	Excluded	Required	
TSC_PhysicalPresence	Excluded, since Physical Presence must not be implemented	Optional	
TSC_ResetEstablishmentBit	Optional	Optional	

9.5 The GetCapability Commands

The GetCapability command group manages the capabilities of a TPM. Every MTM must be able to present details about its capabilities and may optionally offer the feature of setting them.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_GetCapability	Required	Required	E.g. for locally verified boot, it makes sense to check whether the TPM has enough non-volatile memory. Note the additional capability and dataSelect parameters defined in Section 8.1
TPM_SetCapability	Optional	Optional	

9.6 Auditing

Audit, in general, is optional for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_GetAuditDigest	Optional	Optional	
TPM_GetAuditDigestSigned	Optional	Optional	
TPM_SetOrdinalAuditStatus	Optional	Optional	

9.7 Administrative Functions - Management

The administrative management functions are optional for all MTMs. The TPM_ResetLockValue command MAY be replaced with a proprietary mechanism in a MRTM.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_FieldUpgrade	Optional	Optional	Note, that without knowledge of the owner authorization data no field upgrade of the MTM is possible.
TPM_SetRedirection	Optional	Optional	
TPM_ResetLockValue	Optional	Required	See section 8.5
TPM_GetCapabilityOwner	Excluded	Excluded	

9.8 Storage functions

The Storage functions of a TPM are part of the core functionality. Though not necessarily needed for locally verified boot, they are considered to be required for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_Seal	Required	Required	Necessary for the secure storage use case
TPM_Unseal	Required	Required	Necessary for the secure storage use case
TPM_UnBind	Required	Required	Necessary for the secure storage use case
TPM_CreateWrapKey	Required	Required	Necessary for the secure storage use case – for storage key generation
TPM_LoadKey2	Required	Required	Necessary for the secure storage use case – for storage key management
TPM_GetPubKey	Required	Required	Necessary for the secure storage use case – especially for binding data.
TPM_Sealx	Required	Required	Necessary for the secure storage use case

9.9 Migration

Migration, in general, is optional for MRTMs and required for MLTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_CreateMigrationBlob	Optional	Required	
TPM_ConvertMigrationBlob	Optional	Required	
TPM_AuthorizeMigrationKey	Optional	Required	
TPM_MigrateKey	Optional	Excluded	
TPM_CMK_SetRestrictions	Optional	Optional	
TPM_CMK_ApproveMA	Optional	Optional	
TPM_CMK_CreateKey	Optional	Optional	
TPM_CMK_CreateTicket	Optional	Optional	
TPM_CMK_CreateBlob	Optional	Optional	
TPM_CMK_ConvertMigration	Optional	Optional	

9.10 Maintenance

Maintenance functions are optional (also optional in TPM Main Spec 1.2).

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_CreateMaintenanceArchive	Optional	Optional	
TPM_LoadMaintenanceArchive	Optional	Optional	
TPM_KillMaintenanceFeature	Optional	Optional	
TPM_LoadManuMaintPub	Optional	Optional	
TPM_ReadManuMaintPub	Optional	Optional	

9.11 Cryptographic Functions

The cryptographic functions of a TPM offer a range of cryptographic functionality. The SHA-1 algorithm has to be present within every MTM anyhow since every MTM must be able to compute HMAC for authentication purposes. The exposure of this interface is optional as key certification is, while signing and the provision of random numbers is required for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_SHA1Start	Optional	Optional	The SHA1 when used without a key is not a security sensitive operation.
TPM_SHA1Update	Optional	Optional	
TPM_SHA1Complete	Optional	Optional	
TPM_SHA1CompleteExtend	Optional	Optional	
TPM_Sign	Required	Required	The user may want to strongly protect his signing keys.
TPM_GetRandom	Required	Required	
TPM_StirRandom	Required	Required	
TPM_CertifyKey	Required	Required	
TPM_CertifyKey2	Optional	Optional	

9.12 Endorsement Key Handling

The Endorsement Key Handling command group manages the endorsement key. Since the endorsement key might be generated outside the MTM, the creation command is optional. The same holds for the revocation commands.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_CreateEndorsementKeyPair	Optional	Required	
TPM_CreateRevocableEK	Optional	Optional	
TPM_RevokeTrust	Optional	Optional	
TPM_ReadPubek	If the MRTM has an EK THEN this command MUST be available on the host platform, however it MAY NOT be implemented by a MTM. It could for example just be a simple stub function in a software library. If a MRTM does NOT have an EK THEN this function is unnecessary and hence Optional.	Required	EK may not be available, hence this command might not be applicable at all.
TPM_OwnerReadInternalPub	Optional	Required	EK may not be available and the SRK can be symmetric, hence this command might not be applicable at all.

9.13 Identity Creation and Activation

These two commands are Optional for the MRTM. The MRTM may accept both the MTM_PERMANENT_DATA->verificationAuth as an authorization secret in addition to the owner secret for authorizing TPM_MakeIdentity and TPM_ActivateIdentity commands.

If neither field is defined, desirable or available for identity enrollment in a MRTM then the MTM must support delegation. The minimum support is a static delegation table built into the MTM at manufacture that allows the use of these commands. In this case TPM_DSAP needs to be supported during operation of these commands.

Note that the TPM_ReadPubEK is OPTIONAL for the MRTM. This does not directly impact these commands, but if a MRTM must be able to interoperate with a Privacy CA for identity enrollment, then access to the public part of the EK is necessary.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_MakeIdentity	If MRTM has an EK and MTM_PERMANENT_DATA->aik is undefined: Required If MRTM has MTM_PERMANENT_DATA->aik defined: Optional. If the MRTM has no EK THEN MTM_PERMANENT_DATA->aik MUST be defined.	Required	
TPM_ActivateIdentity	If MRTM has an EK and MTM_PERMANENT_DATA->aik is undefined: Required If MRTM has MTM_PERMANENT_DATA->aik defined: Optional. If the MRTM has no EK THEN MTM_PERMANENT_DATA->aik MUST be defined.	Required	

9.14 Integrity Collection and Reporting

The Integrity Collection and Reporting command group is required for all MTMs, but with some exceptional treatment for so-called verified PCRs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_Extend	Required However, a TPM_Extend implementation MUST exclude extending *verified PCRs.*	Required However, a TPM_Extend implementation MUST exclude extending *verified PCRs.*	See section 8.2
TPM_PCRRead	Required	Required	
TPM_Quote	Required	Required	
TPM_PCR_Reset	Optional. However, a TPM_PCR_Reset implementation MUST exclude resetting *verified PCRs.*	Optional However, a TPM_PCR_Reset implementation MUST exclude resetting *verified PCRs.*	See section 8.4
TPM_Quote2	Optional	Required	See section 8.7 for a description of how the locality modifier is handled

9.15 Changing AuthData

The Changing AuthData command group is required for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_ChangeAuth	Required	Required	TPM_ChangeAuth MAY fail for the SRK usageSecret. Data stored under the SRK in a MRTM might be owned by somebody else who should be able to change his auth data
TPM_ChangeAuthOwner	Optional	Required	TPM_ChangeAuthOwner may fail for a MRTM. At least, the user should be able to change authorization passwords, for a MRTM this is not required.

9.16 Authorization Sessions

Among the authorization sessions only the OIAP and the OSAP commands are required for all MTMs in order to allow for authorized access to protected objects like cryptographic keys.

The DSAP and the SetOwnerPointer command is optional for all MTMs since delegation itself is optional.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_OIAP	Required	Required	
TPM_OSAP	Required	Required	
TPM_DSAP	Optional	Optional	Delegation is optional.
TPM_SetOwnerPointer	Optional	Optional	

9.17 Delegation

Delegation, in general, is optional for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_Delegate_Manage	Optional	Optional	
TPM_Delegate_CreateKeyDelegation	Optional	Optional	
TPM_Delegate_CreateOwnerDelegation	Optional	Optional	
TPM_Delegate_LoadOwnerDelegation	Optional	Optional	
TPM_Delegate_ReadTable	Optional	Optional	
TPM_Delegate_UpdateVerification	Optional	Optional	
TPM_Delegate_VerifyDelegation	Optional	Optional	

9.18 Non-volatile Memory

The command group managing the direct access to non-volatile memory is optional for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_NV_DefineSpace	Optional	Optional	
TPM_NV_WriteValue	Optional	Optional	
TPM_NV_WriteValueAuth	Optional	Optional	
TPM_NV_ReadValue	Optional	Optional	
TPM_NV_ReadValueAuth	Optional	Optional	

9.19 Session Management

The Session Management command group is optional for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_KeyControlOwner	Optional	Optional	
TPM_SaveContext	Optional	Optional	
TPM_LoadContext	Optional	Optional	

9.20 Eviction

The FlushSpecific command is essential for resource and key management within the TPM, hence it is required for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_FlushSpecific	Required	Required	

9.21 Timing Ticks

The Timing Ticks command group is optional for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_GetTicks	Optional	Optional	
TPM_TickStampBlob	Optional	Optional	

9.22 Transport Sessions

The establishment of transport sessions is essential for confidential message exchange, hence the commands of this group are required for all MTMs, except for the ReleaseTransportSigned command which is optional. The ability to implement a secure channel between a MTM and another entity is required to protect the integrity for non-OSAP/OIAP/DSAP commands (e.g. MTM_VerifiedRIMCertAndExtend does provide integrity protection for the RIM certificate, but not for the choice of RIM Certificate) and also the confidentiality for e.g. TPM_Unseal results. See also Section 8.11.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_EstablishTransport	If there is no inherent confidential channel between the MTM and its RTM or RTV (or verification agents) THEN: Required Else: Optional	If there is no inherent confidential channel between the MTM and its RTM or RTV (or verification agents) THEN: Required Else: Optional	
TPM_ExecuteTransport	If there is no inherent confidential channel between the MTM and its RTM or RTV (or verification agents) THEN: Required Else: Optional	If there is no inherent confidential channel between the MTM and its RTM or RTV (or verification agents) THEN: Required Else: Optional	
TPM_ReleaseTransportSigned	Optional	Optional	

9.23 Monotonic Counter

The Monotonic Counter command group is optional for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_CreateCounter	Optional	Optional	
TPM_IncrementCounter	Required for the counters that are mandatory as defined in Section 6. Optional for other counters.	Required for the counters that are mandatory as defined in Section 6l.Required for counterStorageProtect. Required for counterRIMProtect if verification commands supported by MLTM. Optional for other counters.	
TPM_ReadCounter	Required for the counters that are mandatory as defined in Section 6 Optional for other counters.	Required for the counters that are mandatory as defined in Section 6l.Required for counterStorageProtect. Required for counterRIMProtect if verification commands supported by MLTM. Optional for other counters.	
TPM_ReleaseCounter	Optional	Optional	
TPM_ReleaseCounterOwner	Optional	Optional	

9.24 Direct Anonymous Attestation

The Direct Anonymous Attestation command group is optional for all MTMs.

Command – TPM Main Spec	MRTM	MLTM	Remark
TPM_DAA_Join	Optional	Optional	
TPM_DAA_Sign	Optional	Optional	

1

10. Example

10.1 Overview

Start of informative comment:

This section describes an example that intends to demonstrate how the MTM_VerifyRIMCertAndExtend and the associated methods may be used.

The following notation is used in these examples:

- Let *state* = [(*i*, *v1*), (*j*, *v2*),...] denote a set of PCRs such that PCR w/ index *i* holds the value *v1* and PCR w/ index *j* holds the value *v2* and so on.
- Let *RIM_Cert{K}(state, index, event)* denote a TPM_RIM_CERTIFICATE instance signed by key *K* authorizing an extend of *event* into PCR *index* when the PCRs already contain the values represented by *state*.
- An *event* can be for example the loading of a software image and that event can be represented by a SHA1 hash of that image.
- Let *Verification_Key{K}(V, usage)* denote a TPM_VERIFICATION_KEY instance signed by key *K* and authorizing the *public key V* with the usageFlags field *usage*
- Denote by *img_OS* an operating system image.
- Denote by SHA1(*x*) the SHA1 hash over the byte-string *x*, *e.g.* SHA1(*img_OS*) is the SHA1 hash of the operating system *img_OS*.
- Denote by TPM_Extend(*state*, *index*, *x*) the result of extending *state* (a set of PCRs as described above) with the event *x* into PCR *index* as defined by TPM_Extend[3].
- The 160-bit string of all zeros is denoted by 00..00.

End of informative comment.

10.2 Secure Boot

Start of informative comment:

This sub-section shows an overview on how to implement a managed secure boot mechanism using the primitives in the specification. Let us assume the boot sequence consists of two software executables that must be loaded and executed in a defined order, before the *img_OS* can be loaded and executed. Denote these images *img*1 and *img2*. These are provided by independent developers who do NOT have access to each others images. As an additional requirement we require that an update to *img1* must not require any additional actions by the supplier of *img2*.

The system has at least the following states:

- state_0 = [(0, *diag_hw*), (1, *diag_rots*), (2, 0), (3, 0), ...]
- state_1 = TPM_Extend(state_0, 2, SHA1("RTV done"))
- state_2 = TPM_Extend(state_1, 3, SHA1(*img1*))
- state_3 = TPM_Extend(state_2, 2, SHA1("img1 loaded"))
- state_4 = TPM_Extend(state_3, 4, SHA1(*img2*))
- state_5 = TPM_Extend(state_4, 2, SHA1("img2 loaded"))
- state_6 = TPM_Extend(state_5, 7, SHA1(*img_OS*))
- state_7 = TPM_Extend(state_6, 2, SHA1("OS ready"))

The state state_0 represents the initialization of all PCRs from 2 upwards to zero, while PCRs 0 and 1 contain diagnostic information about the Hardware Platform and Roots of Trust themselves.

For simplicity this example has only one verification key Root = Verification_Key(.)(K, TPM_VERIFICATION_KEY_USAGE_SIGN_RIMCERT | TPM_VERIFICATION_KEY_USAGE_SIGN_RIMAUTH | TPM_VERIFICATION_KEY_USAGE_INCREMENT_BOOTSTRAP). For each of the states i > 0, we have a corresponding RIM_Cert_i = RIM_Cert{Root}(state_i, ...) that authorizes the extend into state_i from the preceding state. It is also assumed that PCRs 0 to 7 are verified PCRs, i.e. they can only be extended using MTM_VerifyRIMCertAndExtend. To reduce the number of RIM_Certs required, the verifiedPCR selection is set after state_0 has been reached.

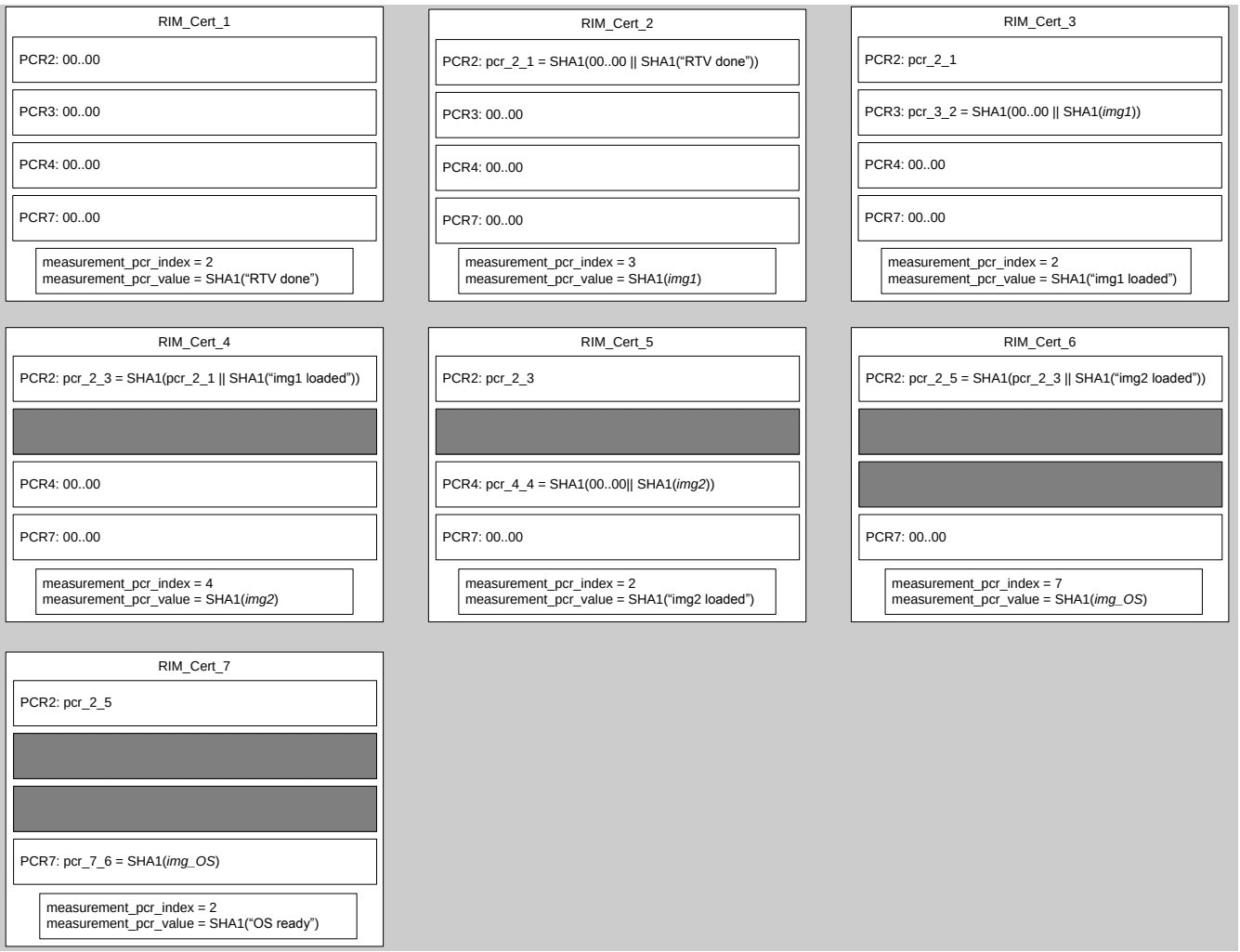


Figure 3. RIM Certificates for the boot process

Figure 3 shows the actual RIM Certificates generated for the boot process. The grayed out PCRs represent "Don't Care" in the PCR selection inside the TPM_RIM_CERTIFICATE.

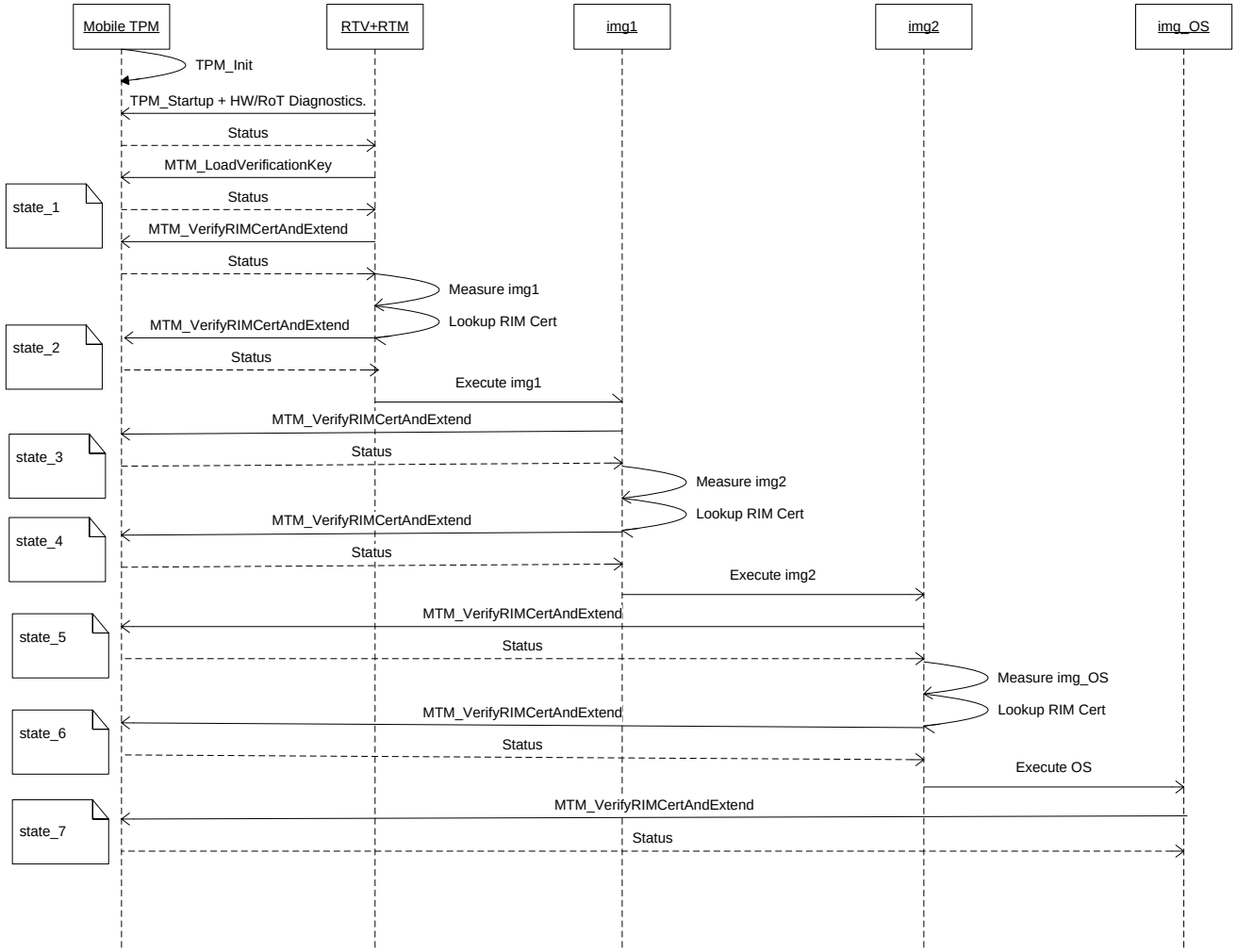


Figure 4. Example sequence during boot

Figure 4 shows the sequence how the boot would proceed through the above-mentioned states. The example is not the only way to create a secure boot mechanism using the primitives defined in this specification, rather it is constructed to show the utility of the MTM_VerifyRIMCertAndExtend and associated functionality.

The example boot is as follows:

- The MRTM starts up by having TPM_Init and TPM_Startup being called.
- All PCRs are initialized with the value 00..00.
- The RTV records diagnostic information about the Hardware Platform and Roots Of Trust into PCRs 0 and 1. The RTV calls MTM_SetVerifiedPCRSelection to set PCRs 0 to 7 as verifiedPCRs, and calls MTM_LoadVerificationRootKeyDisable.
- The RTV records into PCR 2 a SHA1 hash of the string ("RTV done") using MTM_VerifyRIMCertAndExtend and RIM_Cert_1.
- Next the RTV measures *img1* and looks up a RIM Cert for it. It should find RIM_Cert_2 for it.
- The RTV calls MTM_VerifyRIMCertAndExtend for RIM_Cert_2.
- Control is then passed to *img1*.

- *img1* extends into PCR 0 a SHA1 hash of the string ("img1 loaded") using RIM_Cert_3.
- *img1* then measures *img2* and looks up RIM_Cert_4.
- *img1* calls MTM_VerifyRIMCertAndExtend for RIM_Cert_4
- Control is then passed to *img2*.
- *img2* repeats the same steps as *img1* for RIM_Cert_5 and RIM_Cert 6.
- Control is then passed to the OS
- OS extends finally a SHA1 hash of the string ("OS ready") into PCR 2

If any of the MTM_VerifyRIMCertAndExtend calls return an error or the appropriate MTM_VerifyRIMCertAndExtend certificate is not found then the boot is aborted.

The above example provides the following advantages:

- The secure boot configuration is protected against tampering.
- Any component (*img1* or *img2*) of the secure boot chain can be updated, without updating the RIM certificates of the following components. This is due to the ability of using PCR 2 as a pre-requisite in the MTM_VerifyRIMCertAndExtend calls.
- Multiple execution paths of the secure boot are possible.
- RIM certificates for *img1*, *img2* and the OS can be produced independently of each other, as long as the platform integrator has fixed and published the strings being extended into PCR 2.
- Boot configuration can be managed remotely, by adding new RIM certificates.
- Entire boot configuration (with the exception of the recognition of the Verification Key *Root*) can be loaded onto volatile storage while the device is offline.

End of informative comment.

10.3 Remote Attestation and a Resource-Constrained Verifier

Start of informative comment:

The above mechanism also allows performing remote attestation to a resource-constrained verifier. This can be relevant for example in a case where a mobile phone is attempting to provide remote attestation to a smart card.

In addition to the above, the AIK credentials are associated either implicitly or explicitly with an RVAI. This RVAI is the root verification key loaded using MTM_LoadVerificationKey that is used to (directly or indirectly) authorize all TPM_RIM_Certificate instances accepted by MTM_VerifyRIMCertAndExtend extending PCR 2. If a remote verifier is provided with the public part of the RVAI key then the remote verifier can merely check the AIK signature, the AIK credentials, whether it trusts the key RVAI and then in this example case, the contents of PCR 2. The reason why PCR 2 is the only PCR necessary to check is that all extends to PCR 2 have been authorized by the RVAI and the events recorded into PCR 2 translate the events (e.g. extensions of the SHA1(*img1) etc..)* in the other PCRs into well-known bit-strings (e.g. SHA1("img1 loaded").

The remote verifier would NOT need to be aware of all the multitudes of configurations that are legitimate, it can instead trust a list of verification keys that are used to authorize MTM_VerifyRIMCertAndExtend operations.

Note that this will work well even if the bootchain is not required to be verified and any software image could be loaded as the initial image. In this case OS image (and other boot sequence components) would be measured using TPM_Extend. MTM_VerifyRIMCertAndExtend would only be used to record a statement of the validity (based on the RIM_Certs signer) about the boot-sequence which has been measured into e.g. PCR 2.

End of informative comment.

10.4 Re-sealing

Start of informative comment:

Relying on just TPM_Extend for restricting access to sealed data to identified programs causes the problem that if some program in the preceding state needs to be updated, all sealed data must be decrypted and re-encrypted (this is often referred to as re-sealing).

This specification attempts to minimize the amount of redundant management functionality and has chosen a mechanism that allows the re-sealing problem to be minimized. The ability to use PCRs for recording statements about the system state (as reflected in the PCRs) allows one to seal to these *verified PCRs*. In the above example, sealing for example to PCR0 would not require any re-sealing, if the SHA1 digest of *img1, img2 or img_OS* changed, as long as *RIM_Cert_3, RIM_Cert_5 and/or RIM_Cert_7* is updated.

This follows from the fact that sealing is merely done to a value in PCR 2 which consists of the extensions of digests of "RTV done", "img1 loaded", "img2 loaded" and "OS ready". These digests can only have been extended, if the corresponding RIM_Certs (3, 5, 7) authorized the extend. The pre-requisite for authorizing the extend was that *img1*, *img2* or the OS had certain digests defined (as contained in RIM_Certs 2, 4 and 6) by the RIM_Cert signer.

End of informative comment.

10.5 Reactive Run-Time Responses

Start of informative comment:

Finally the ability to use MTM_VerifyRIMCertAndExtend allows to hold configuration data for a watchdog that attempts to detect during run-time whether e.g. a Trusted Computing Base (TCB) has been corrupted. A set of PCRs can be allocated for the watchdog and configured as *verified*.

Assume now that the TCB image has been extended into PCR 7 and that PCRs 8, 9 and 10 have been allocated for the watchdog. Assume that the TCB image that resides in memory is different from the image that resides on disk (due to dynamic linking etc...). Assume further that for performance reasons we wish to cycle through the TCB image in memory in three steps. Partition the TCB address-space into three parts, and allocate the first part to PCR 8, second part to PCR 9 and the third part to PCR 10. Generate three RIM_Certs that have as a pre-requisite state the value in PCR 7 and each authorize the extension of the expected digest of the memory-range corresponding to its PCR.

The watchdog can now compute a digest of a partition '*x*' of the TCB , compute SHA1(00..00 || SHA1(x)) and compare it to the value in a PCR. If the watchdog is running in a separate context from the TCB this may provide some additional capability towards attacks that attempt to modify the TCB in-memory.

End of informative comment.

[End of document]