TCG Mobile Reference Architecture

Specification version 1.0
Revision 1
12 June 2007

Contact: mobilewg@trustedcomputinggroup.org

TCG PUBLISHED

TCG

Revision History

1	First Revision of version 1.0

Table of Contents

1. Scope and Audience

The TCG specifications define a Trusted Platform Module (TPM) and its use in a PC client. The "TCG Mobile Trusted Module Specification" [5] is a specification that defines the necessary interface for implementing Mobile Trusted Modules. This specification defines a reference architecture that defines ways of instantiating and using Mobile Trusted Modules as defined in "TCG Mobile Trusted Module Specification".

This document is an industry specification that enables the building of trust in mobile phones using a standardized approach.

1.1 Key words

The key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in the chapters 2-7 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 94, March 2006

[2] Trusted Computing Group, *TPM Main Part 2 TPM Structures,* Specification Version 1.2 Revision 94, March 2006

[3] Trusted Computing Group, *TPM Main Part 3 Commands,* Specification Version 1.2 Revision 94, March 2006

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

[5] Trusted Computing Group, TCG Mobile Trusted Module Specification, Version 1.0 Revision 1, June 2007

2. Basic Definitions

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 a Mobile Trusted Module (MTM) to some other entity.

End of informative comment.

2.1 Glossary

Abbrevation	**Description**
AIK	Attestation Identity Key. A key used to sign remote attestations. Defined in [2] and [3].
Allocated Resources	A resource that is built from dedicated resources and/or allocated resources.
Communications Carrier	An actor that controls access to the cellular radio network through its engine.
CounterBootstrap	A monotonic counter in a Shielded Location that is used to provide version control for integrity credentials used to verify SW during pristine boot. This counter state is typically a global platform or engine variable and therefore bound to a particular class/model of platform or engine.
CounterRIMProtect	A monotonic counter in a Shielded Location that is used to provide version control for engine-specific credentials used to verify SW during standard boot. This counter is a local engine variable and therefore bound to a particular engine instance.
Dedicated Resources	Resources that automatically become available to the platform.
Device Manufacturer (DM)	An actor that controls the DM Engine (a Mandatory Engine that controls communications between the Trusted Engines in a platform). This stakeholder is responsible for the IMEI, for example, and the core mandatory functions of the platform.
Device Owner	An actor that controls the authorized presence of Discretionary Engines in a platform, normally the legal owner of the device
Discretionary engine	A trusted engine that may be present in a platform under the authorization of the Device Owner.
Engine	A dedicated processor or run-time environment with access to trusted resources that is used to run trusted services and normal services. An engine may consist entirely of dedicated resources or may have parts of it instantiated as allocated resources.
integrityCheckRootData	Data that enables verification of the stored representation of the RVAI.
Mandatory Engine	A DM authorized trusted engine that must be present In a platform to maintain its trusted state.
Measurement Verification Agent (MVA)	The combination of a Measurement Agent and Verification Agent associated with at least one target object.
Measured Resources	Normal resources that have been measured by Trusted Services.

Normal Resources	Engine resources that are not supplied with an EK or an AIK.
Normal Services	Services instantiated by customizing normal resources.
Reference Integrity Metric (RIM)	A value used to validate the result of a measurement taken before software or hardware is loaded or initialized (for execution). Typically a digest of compiled software and configuration data which can affect the engine trust state.
RIM_Auth	An actor that signs the RIM_Certs and delegation RIM_Auth_Certs under its authority.
Primary RIM Auth	A RIM_Auth whose authority has been assigned directly by the RVAI.
RIM_Auth_Cert	A certificate used to validate the identity and authority of a RIM Auth, typically instantiated as a TPM VERIFICATION_KEY structure.
Internal RIM_Cert	A certificate containing a RIM value, generated internally in the platform using the MTM_INSTALL_RIM command.
External RIM_Cert	A means of securely authenticating RIM information for a given target object, provided by an authorized RIM_Auth.Typically this is a data structure that is signed by the RIM_Auth.
RIM Conversion Agent	An agent that converts the arbitrary formats of external RIM_Certs into the TCG-defined formats of internal RIM Certs.
RIM_run	A value used to validate the result of a measurement taken after software is loaded (after the software is executing). Typically a digest of the image of software executing in memory.
RIM_run_Cert	A certificate containing a RIM_run value.
Root-of-Trust for Enforcement (RTE)	An optional Root-of-Trust that instantiates other Roots-of-Trust in a trusted engine, if and when they are not available as dedicated resources.
Root-of-Trust for Verification (RTV)	A Root-of-Trust which is the first verification agent used when secure booting an engine.
RVAI	The root public key of a hierarchy of RIM_Auth public keys.
Supplier	The party that originally provided a Trusted Component for use within a platform, for instance a Root of Trust.
Target Integrity Metric (TIM)	Integrity Metric of a target object or component as measured by the measurement agent of that object. Typically this is a hash of a software image of the executable code of the object, along with its associated configuration data.
Target Object (TO)	A particular instantiation of a trusted component on a platform that is measured by a measurement agent (MA) to produce an integrity measurement called TIM.
Trusted Component (TC)	A conceptually singular, separately identifiable part of a platform or engine that is supplied by an authorized and authenticated provider. Trusted Components may be either hardware or software. They are active entities that are connected together through a variety of means to form the complete platform or engine. Trust in the component is a policy decision by the domain (mandatory or discretionary) owner, and must be supported a provider's RIM certificate. That is, they can be trusted to the extent that they are authenticated and that the

	provider (RIM_Auth) is trusted.
Trusted Resource	Engine Resources that are supplied with a unique EK or AIK
Trusted Services	Services instantiated by customizing trusted resources.
Verification Agent	A local agent of the engine stakeholder used when secure-booting an engine. It verifies that software about to be loaded (and executed) is described and authorized in a RIM_Cert
Verified Extend	A TPM_Extend that requires verification of data in a TCG-specified RIM_Cert, before extending that data into a PCR; the function is performed by MTM_VerifyRIMCertAndExtend.
Validity List	A credential containing a list of valid (RIM or RIM Auth) external certificates for a particular target object.

2.2 Representation of Information

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 2.3.1, 2.3.2 and 2.3.3 are reproduced here for the readers convenience. This document fully re-uses the type definitions from [2] as reproduced in Sections 2.3.1 and 2.3.2.

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. The first three of these type definitions are reproduced from [5].

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
TPM_TAG_RIM_AUTH_VALIDITY_LIST	0x0305	TPM_RIM_AUTH_VALIDITY_LIST
TPM_TAG_RIM_VALIDITY_LIST	0x0306	TPM_RIM_VALIDITY_LIST

2.3.4 MTM 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 Overview

This specification defines an abstract architecture of a trusted mobile platform. This specification does not define how the architecture must be implemented, but compliant implementations must have certain specified properties and functions.

The scope of the specification is:

- the definition of a trusted mobile platform as a collection of engines in mandatory and discretionary domains, and means for the engines to communicate with other and with generic resources.
- the definition of a generic engine
- the definition of a set of trusted resources required to instantiate a trusted engine

3. Introduction

Start of informative comment:

Cellular-radio enabled platforms conform to regulations that govern network protocols and require unrestricted access to proper network parameters. Otherwise, a cellular-radio enabled platform cannot operate (and should not operate, for fear of disabling the network). Regulations dictate that certain network parameters (such as the IMEI) must be unique to individual platforms. It follows that a trusted mobile platform requires guaranteed availability of specific functions with access to protected data. Conventional TCG-enabled platforms enforce the rights of a single platform Owner (who has exclusive control over the data protection mechanisms in the platform) and the rights of multiple data owners (who use the data protection mechanisms, with permission from the platform Owner). If a cellular-radio enabled platform was just a conventional TCG-enabled platform, it follows that an Owner or user who turned off his TPM would prevent the radio from operating. To maintain the right of an Owner or user to turn off his TPM, this specification generalizes the concept of a platform to mean a set of conventional TCG-enabled platforms, and calls them "engines" to differentiate them from the ensemble platform. Each trusted engine has a separate Owner, called a "stakeholder", who has exclusive control over the data protection mechanisms in their own engine, and can permit data owners to use those data protection mechanisms. Some engines are mandatory and some are discretionary.

This specification introduces functions for the secure-boot of a MTM-enabled engine. A conventional TCG-enabled engine provides data protection and platform attestation. Secure-boot additionally forces the engine to boot properly or not at all. Secure-boot therefore provides an engine's stakeholder with confidence that certain services were correctly instantiated when they are available, and is particularly valuable to entities whose engines are constrained by regulations. The extra facilities for secure-boot include an entity called the RVAI (Root Verification Authority Identifier), Reference Integrity Metric Certificates, and a Root-of-Trust-for-Verification.

This specification introduces mandatory engines, which are always resident in a platform and hence particularly useful for absent (remote) stakeholders. The term "mandatory engines" comes from the fact that they provide mandatory (critical and indispensable) services, including services subject to regulatory enforcement

This specification introduces discretionary engines, which may or may not be resident in a platform. The term "discretionary engines" comes from the fact that they provide discretionary (non-critical) services.

This specification also introduces the concept of the role of a Device Owner, which has exclusive control over the presence in the platform of discretionary engines and mandatory engines that provide critical and indispensable services but are not subject to regulatory enforcement.

Current phone implementations often split a baseband processor (which controls the radio unit, radio software, voice functions, interactions with SIM card etc.) from an applications processor (which has a fully fledged OS and does all user-visible interactions: menus, icons, camera, multimedia, music player, web browser, messaging and so on).

This specification introduces the concept of a secure-boot mandatory Device Manufacturer engine. This might do all baseband processor and applications processor functions in a simple platform. In a more complex platform, however, the DM engine might just control the basic hardware and interfaces. The Device Manufacturer stakeholder could be the Mobile Equipment manufacturer, for example.

This specification introduces the concept of a Communications Carrier engine that could eventually be both secure-boot and mandatory, and perform the functions currently executed by the baseband processor: it would be capable of being upgraded with new radio protocols, patches, enhancements and so on. The Communications Carrier engine must obviously be strongly isolated from most of the other applications on the platform to prevent undesirable results such as incompatible protocols, interference, radio hijacking, for example. The Communications Carrier stakeholder could be the cellular network provider, for example.

This specification introduces the concept of a Service Provider engine that could eventually perform some of the user-visible functions currently done by the applications processor. Discretionary Service Provider engines could do specific functions (music player, web browser, and messaging, for example) currently done by the applications processor.

Cellular-radio enabled platforms are also often embedded implementations. This specification therefore includes concepts intended to enable implementation of embedded mobile platforms. These concepts include dedicated and allocated resources, and a Root-of-Trust-for-Enforcement.

***End of informative comment*.**

4. Reference Architecture for Mobile Trusted Computing

4.1.1 Architecture Overview

Start of informative comment:

This specification abstracts a trusted mobile platform into 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 platform, and provide evidence about the current state of the platform. This abstraction enables designers to implement platforms using one or more processors, each processor supporting one or more engine.

End of informative comment

Internal engine operation MAY be independent of other engines, or a superior engine MAY provide resources for a subordinate engine. If a superior engine provides resources to a subordinate engine, and the superior engine is working normally, those resources MUST conform to their published properties. If a superior engine provides resources to implement Protected Capabilities and/or Shielded Locations in a subordinate engine, and the superior engine is working normally, those resources MUST be compatible with the properties of Protected Capabilities and Shielded Locations defined in the TPM Main Specification Part I section three "Protections".

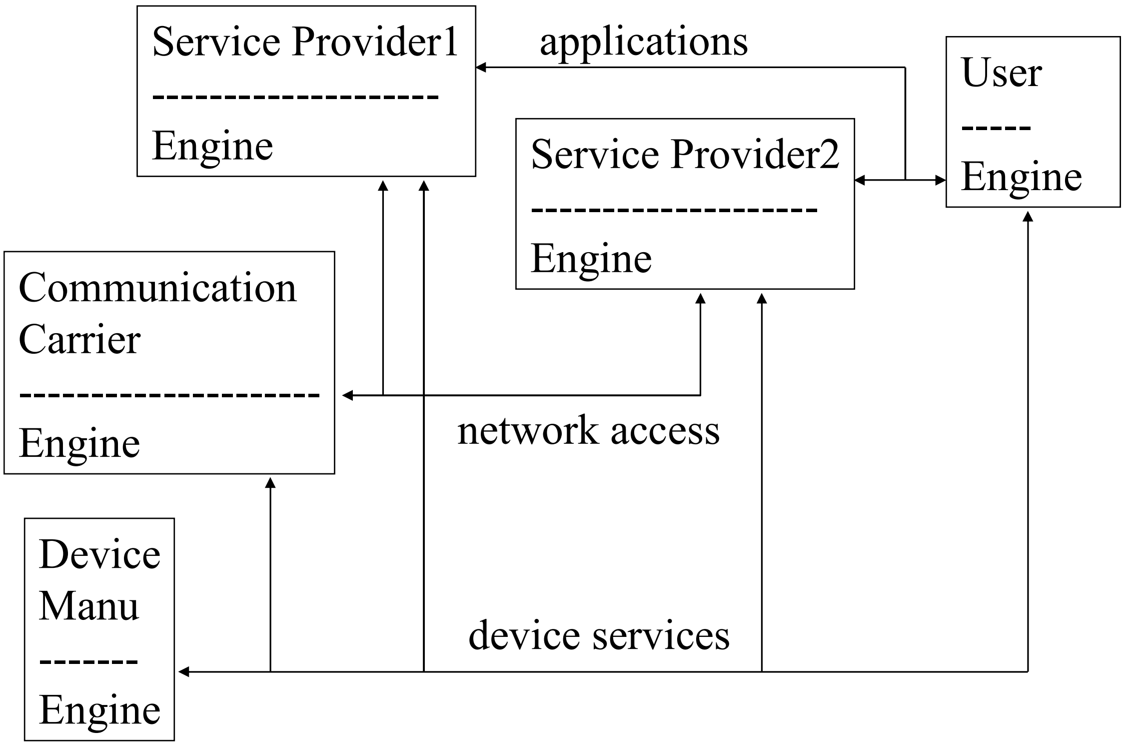

Figure 1. Generic Trusted Mobile Platform.

Start of informative comment:

A trusted mobile platform, shown in Figure 1 contains multiple abstract engines, each acting on behalf of a different stakeholder. Stakeholders are role-playing entities.

End of informative comment

An entity MAY perform one or more stakeholder roles.

Start of informative comment:

The principle stakeholders in a trusted mobile platform are:

- Users, who store their data in the platform. There may be multiple User stakeholders in a platform. An employee or a consumer may be a User stakeholder, for example.
- Service Providers, who provide services consumed in a platform. There may be multiple Service Provider stakeholders in a platform. Some examples of services are: corporate services for employees; content distribution services for consumers; an address book; a diary.
- Communications Carriers, who are specialist Service Providers providing cellular radio access for the platform. There may be multiple Communications Carrier stakeholders in a platform.
- The Device Manufacturer, who provides the internal communications within a platform and typically provides all the hardware resources within a platform. There is a single Device Manufacturer stakeholder in a platform.

An engine's purpose is to enable confidence in all the services exported and consumed by the engine. The engine enforces policies: the engine's resources constrain what the engine can do, and the engine's Owner (its stakeholder) constrains how the engine can be used. Each engine is isolated from other engines. The resources of these engines are provided by the platform or another engine. The isolation of these engines depends on the integrity of the platform and the integrity of the platform may depend on the isolation of these engines. This specification does not mandate a specific form or strength of isolation, which should depend on the purpose of the trusted mobile platform. Normal processes, threads and sandboxes may be sufficient. More sophisticated techniques such as virtual machines, hypervisors and the like may be required.

Each stakeholder has its own engine. Each engine provides platform services on behalf of its stakeholder. A Device Manufacturer's engine is responsible for the integrity and configuration of a device, including the presence of mandatory engines and specified discretionary engines in the platform, coordinating communications between engines in the platform, and controlling access to protected platform resources. In one example, in a general case, a Communications Carrier's engine is responsible for AAA (Authentication, Authorization, Access control) services related to network access and a User's engine consumes services provided by other engines and protects data on behalf of a user of a device.

A trusted mobile platform has both a mandatory domain and a discretionary domain.

The Device Manufacturer actor determines some of the engines in the mandatory domain. This specification also defines the role of a Device Owner, which determines the remaining engines in the mandatory domain and determines all engines in the discretionary domain. This role is NOT the same as a stakeholder, who owns an individual engine. However, the role will generally be performed by the legal owner of the Device, and that legal owner may in fact be a stakeholder for an engine.

Mandatory engines provide the indispensable functionality of a trusted mobile platform, especially those functions required to comply with regulations governing the operation of mobile platforms in cellular radio systems. Mandatory engines are required to be supported by a Mobile Remote owner Trusted Module (MRTM), which supports secure boot and does not permit a local Operator to remove the stakeholder from the engine. The intent is to ensure that the correct services are always present in a platform. Mandatory engines therefore require a stakeholder that will ensure that the Roots of Trust in a mandatory engine are always enabled and activated.

Discretionary engines provide services that must be capable of being added, removed, turned on and turned off without consent of any external service provider. Discretionary engines are required to be supported by a Mobile Local owner Trusted Module (MLTM), which is not required to support secure boot and does permit a local Operator to remove the stakeholder from the engine and become the stakeholder of the engine. The Device Owner is responsible for ensuring that all discretionary engines and their stakeholders conform to the Device Owner's privacy policy.

Both mandatory and discretionary engines are implicitly capable of protecting data, irrespective of stakeholder. This is because the fundamental architectural properties of TCG technology prevent any engine's stakeholder subverting an engine's Protected Storage facilities. Therefore it does not matter who is an engine's stakeholder if the only concern is data protection. Data could be protected in separate isolated engines in a platform, or in separate isolated compartments in the same engine in a platform. As a result, only one mandatory engine with the entity performing the Device Manufacturer role as stakeholder is strictly necessary to provide services subject to regulatory enforcement, only one mandatory engine with the entity performing the Device Owner role as stakeholder is strictly necessary to provide other indispensable functionality, and only one discretionary engine with the entity performing the Device Owner role as stakeholder is strictly necessary to provide discretionary functionality. It may be more convenient or more efficient, however, to provide multiple engines than to provide multiple compartments in a single engine. That is why this specification permits multiple mandatory and discretionary engines.

The Device Manufacturer and the Device Owner permit the presence of engines in the platform via three separate lists (DM_mandatoryEngineList, DO_mandatoryEngineList, DO_DiscretionaryEngineList) enforced by the Device Manufacturer's engine. Every engine in the platform must appear in exactly one list. Not all platforms need all lists.

An engine should be allocated to either a mandatory list or the discretionary list depending on the service provided by the engine. If a service is essential, it should be in a mandatory list. If a service is not essential, it should be in the discretionary list. A handset manufacturer needs at least one engine in the DM_mandatoryEngineList if the DM's engine does not provide all the services needed to satisfy regulations, for example. A corporation needs at least one engine in the DO_mandatoryEngineList if the corporation wishes to install services that are essential for employees, for example. An employee needs at least one engine in the DO_discretionaryEngineList to install non-critical services in a 'phone provided by his employer, for example. A parent needs at least one engine in the DO_mandatoryEngineList to install essential services in a child's 'phone, for example. A child needs at least one engine the DO_DiscretionaryEngineList to install games in a 'phone, for example.

End of informative comment

Engines in the DM_mandatoryEngineList MUST provide services subject to regulatory enforcement, MUST NOT provide indispensable services that are not subject to regulatory enforcement, and MAY provide non-critical services whose access to TCG functionality can be denied by the local Operator.

Engines in the DO_mandatoryEngineList MUST NOT provide services subject to regulatory enforcement, MAY provide indispensable services that are not subject to regulatory enforcement, and MAY provide non-critical services whose access to TCG functionality can be denied by the local Operator

Engines in the DM_mandatoryEngineList and DO_mandatoryEngineList MUST NOT facilitate interference by local operators with their service's access to TCG functionality. Each Engine in the DM_mandatoryEngineList and DO_mandatoryEngineList MUST have a Mobile Remote-owner Trusted Module (MRTM), whose Owner is the stakeholder of that engine. The MRTM of an engine can be built based on an MLTM, TPM v1.2 or even TPM v1.1.

Engines in the DO_discretionaryEngineList MUST NOT provide services subject to regulatory enforcement, MUST NOT provide non-regulatory indispensable services, and MAY provide non-critical services whose access to TCG functionality can be denied by the local Operator..

Each Engine in the DO_discretionaryEngineList MUST have a Mobile Local-owner Trusted Module (MLTM), whose Owner is the stakeholder of that engine. An Engine's MLTM can be built using a TPM v1.2 or even a TPM v1.1.

Start of informative comment:

The Device Manufacturer's engine enforces the lists and controls the presence of engines in both domains, and the interactions of those engines. The nature of the lists is manufacturer-specific. Lists could be simple lists or sophisticated control surfaces that enforce policies, for example. Lists might need to be modified post-manufacture. The DM_mandatoryEngineList might be modified if a DM is sold through a specific Communications Carrier or Service Provider, subject to contract, and then modified again if the Device changes Communications Carrier or Service Provider (e.g. because the contract has expired)."

End of informative comment

The Device Manufacturer MUST have ultimate control over the list of engines in the DM_mandatoryEngineList.

The Device Owner MUST have ultimate control over the list of engines in the DO_mandatoryEngineList. This control mechanism MUST be separate from that used to control the DO_DiscretionaryEngineList.

The Device Owner MUST have ultimate control over the list of engines in the DO_DiscretionaryEngineList. This control mechanism MUST be separate from that used to control the DO_ mandatoryEngineList.

Engines MUST appear in exactly one of the DM_mandatoryEngineList, the DO_mandatoryEngineList, or the DO_DiscretionaryEngineList. One engine on the DM_mandatoryEngineList MUST be the Device Manufacturer's (DM's) own engine. The Device Manufacturer's engine MUST permit engines in the DM_mandatoryEngineList, DO_mandatoryEngineList, and DO_discretionaryEngineList to communicate with other engines in the platform and access generic resources provided by the DM's engine.

Engines in the DM_mandatoryEngineList MUST be booted. Engines in the DO_mandatoryEngineList and DO_discretionaryEngineList should be booted, and failure to boot them MUST be treated as a serious error. In the event of such an error, the DM's engine MUST take appropriate manufacturer-specific action.

The DM's engine MAY use an integrity challenge to determine whether an engine is working properly

Start of informative comment:

Figure 2 illustrates that engines in the domains are able to communicate (shown by thick lines) with each other and access generic services provided by the Device Manufacturer's engine.

End of informative comment.

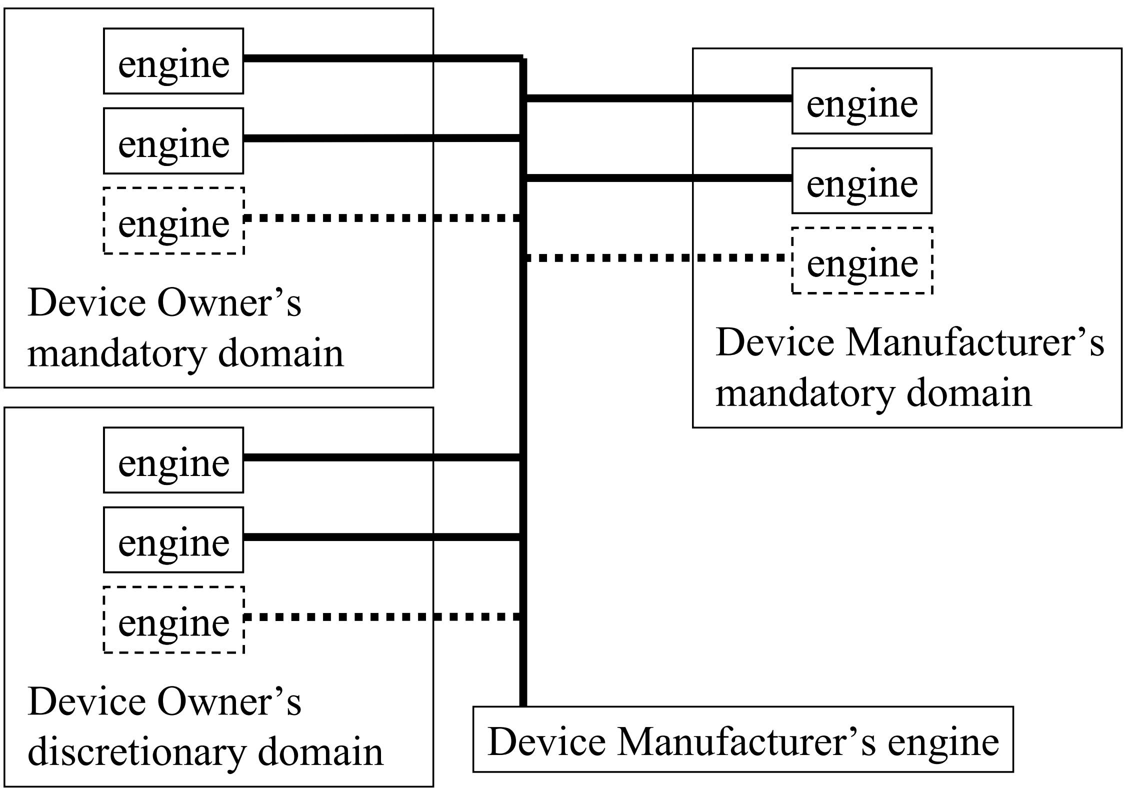

Figure 2. Domains in a Trusted Mobile Platform

Start of informative comment:

The architecture of an engine reflects the fact that there are only two ways by which a resource can be trusted.

- Trusted Resources: The first way is when an entity vouches for a specific instantiation of a resource that contains a specific Endorsement Key and / or Attestation Identity Key. This requires the resource to keep that EK and / or AIK secret. TCG calls such resources "Roots-of-Trust".
- Measured Resources: The second way is when one reliable entity measures an instantiation of a resource and provides that measurement, and (another) reliable entity vouches for the resource by providing a reference measurement. TCG uses RoTs to make the first measurements and provide measurements, while other entities provide reference measurements. Measurement enables the use of resources that are unable to be provided in a way that can guarantee protection of an EK or AIK, or resources whose suppliers do not wish the overhead of providing protection for an EK or AIK. Measurement therefore enables a generic instantiation of a resource to be trusted.

Any part of a platform that is invariant (or can be changed only by the proper entity) could be pre-installed with an EK or AIK and associated credentials to prove that it is genuine. That part need not make or supply any measurements to prove that it is genuine. It simply needs to use its AIKs to sign information. A platform with an invariant Device Manufacturer's engine, for example, could use this method to prove that the Device Manufacturer's engine is genuine. A platform with invariant arbitrary services, for example, could use this method to prove that those services are genuine. The disadvantage of this approach is that a larger part of a platform needs to undergo a security assessment. This TCG architecture, on the other hand, defines just the minimum set of generic Trusted Services that are necessary to measure other resources and services. This TCG approach is simpler because: (1) the same generic Trusted Services can be used no matter what normal service is provided; (2) while the generic Trusted Services still require a security assessment, measured resources and services require just a trust assessment.

End of informative comment.

Engines MAY be instantiated using any combination of dedicated or allocated resources.

Start of informative comment:

Dedicated resources automatically become available to the platform after power is applied to the platform, albeit often after a short delay. A dedicated resource enters a usable state of its own accord. The internal operations of dedicated resources are implicitly isolated from other dedicated resources and from virtual resources. Dedicated resources are inherently parallel, in the sense that they all exist at the same time. Some examples of dedicated resources are hardware / firmware devices such as analogue circuits, solid-state memories, hardware processors, and programmable engines that automatically load firmware to implement particular (fixed) functions.

Any stand-alone trusted mobile platform always needs at least some dedicated resources where data can be protected from attack, but the instantiation of those resources is vendor specific.

Dedicated resources, including Roots of Trust, may be fixed (immutable) or may be capable of being changed.

End of informative comment

If a dedicated RoT can be changed, it MUST verify evidence of sufficient privilege to perform an alteration (before permitting the alteration).

Start of informative comment:

This implies that mutable RoTs must include means to protect and maintain the information used to recognise evidence of privilege.

Dedicated resources may include a monotonic counter that can be regularly incremented, yet never roll-over during the intended life of the platform. A monotonic counter appears essential for verifying that the current state of an engine is the most recent state of an engine.

Allocated resources are functions that do not exist in the platform in a particular boot cycle unless dedicated resources and / or other allocated resources take specific action to create and maintain the resource after power is applied to the platform. Obviously, there must be enough dedicated resources to instantiate allocated resources.

End of informative comment

If some resources are allocated, at least one dedicated resource MUST be responsible for ensuring that mandatory allocated resources are instantiated.

Start of informative comment:

The degree of isolation of internal operations of allocated resources depends on the construction of the resource. Allocated resources may be serial, in the sense that not all allocated resources might exist at the same time. Some examples of allocated resources are the boot software environment created by an Operating System, a virtual machine, and a function implemented in a software application.

One advantage of "dedicated / allocated" terminology is that it enables concise description of all possible implementations of Roots-of-Trusts:

- An allocated RoT must be described in terms of measurements made by a trusted building entity.
- An allocated RoT and its trusted builder can together be described as a dedicated RoT.
- A dedicated RoT is one that is not described in terms of measurements, and therefore must have an Endorsement Key and / or Attestation Identity Key and associated credentials.

The EK and the AIK are defined in the TCG specifications "TPM Main Part 1 Design Principles", "TPM Main Part 2 TPM Structures", and "TPM Main Part 3 Commands". The credentials are defined in the TCG specification "Credential Profiles" (*to be published*).

End of informative comment

Every stand-alone set of dedicated RoTs MUST have either:

- an Endorsement Key, Endorsement Credential, Platform Credential, and Conformance Credential

or

- an Attestation Identity Key and AIK Credential

as evidence that they are genuine.

Start of informative comment:

Trusted Resources need an EK and associated credentials if they will obtain AIK credentials from entities outside the platform. Dedicated resources that build allocated Trusted Resources need an EK or AIK and associated credentials for use by those allocated Trusted Resources. If Trusted Resources in one engine are exported services from another engine, these EK-associated-credentials can be signed by an AIK belonging to the exporting engine. If imported Trusted Resources are exported Trusted Resources from another engine, and only ever obtain AIK credentials from the exporting engine, they don't need an EK. This is because the exporting engine does not need to be convinced that the resource is genuine. An engine that exports a Trusted Resource is implicitly able to check that an AIK is a genuine AIK and simply signs the requested AIK credential using one of its own AIKs.

Engines may or may not need to undergo a boot sequence before being operational. Engines could be instantiated as complete (finished) engines, or as incomplete engines that must finish booting before they are operational. No matter whether resources are dedicated or allocated, or whether the engine is instantiated complete or incomplete, defined mandatory functions must always be present and proper and correct in a completed engine.

Each engine in a trusted mobile platform is provided with the resources necessary to do its job, consumes the services it is designed to consume, and exports the services that it is designed to export. Each engine includes some selection of functions for: (1) reporting evidence (credentials) for the engine's trustworthiness; (2) reporting the evidence (measurements: PCR values and event log) about the engine's current state; (3) obtaining and using Attestation Identity Keys; (4) providing Protected Storage for use by the engine; and (5) other TCG trusted platform functions (time stamping, delegation, etc.) that may be required by that particular engine.

Each engine has many of the same characteristics of a conventional stand-alone TCG platform.

- Each engine has some Protected Capabilities and Shielded Locations, used to implement Trusted Service functions that must not be subverted.
- Each engine could clean-up and continue to persist after processing sensitive data (a static engine), or could shut-down after processing sensitive information (a dynamic engine). Static engines must be designed so that the previous history of the engine does not affect confidence in future processing by the engine. In a dynamic engine, the previous history of the engine is irrelevant but there is a cost due to dynamic creation and destruction of engines.
- Each engine can use attestation identities to prove that information originated in a trusted platform. Each engine's stakeholder controls the use of any EK and any AIKs belonging to that engine. This does not disenfranchise the Device Owner, who either becomes the stakeholder of engines or delegates that privilege to entities that will uphold the Device Owner's privacy policies. The DO always retains the ultimate sanction of expelling any misbehaving non-regulatory engine from the platform, via the DO_mandatoryEngineList and the DO_DiscretionaryEngineList
- Each engine has access to a set of Roots-of-Trust. It can use them to make measurements on "Normal Services" (thereby creating "Measured Services"), and reliably report those measurements to third parties which can then decide whether to trust the service.
- Each engine has access to a Protected Storage facility, with a Storage Root Key and subsequent hierarchy. Each engine may provide its Measured Services with access to that Protected Storage. Then arbitrary data may be encrypted within the engine and the keys stored in Protected Storage. Arbitrary keys may be stored in Protected Storage and used for signing without leaving Protected Storage.
- Each engine may have the ability to delegate Owner privilege within the engine. This is essential if the engine is to be able to prove Owner privilege in the absence of the Owner.
- Each engine may implement other conventional TCG functions, such as time stamping, access to long-term protected non-volatile data storage, and the ability to create transport sessions. The presence or absence of extra functions depends on the intended functionality of the engine's Measured Services.

Some features peculiar to engines in a mobile trusted platform are:

- The Device Manufacturer's engine coordinates communications between the other engines, including the discovery and announcement of all engines within the platform.
- The Device Manufacturer's engine could operate as an internal Privacy-CA. AIK credentials are obtained from a Privacy-CA (as usual). Allocated engines manufactured by the Device Manufacturer's engine could use the Device Manufacturer's engine as an internal Privacy-CA, instead of contacting an external Privacy-CA.
- Identity privacy may not always be necessary. Cellular networks unambiguously authenticate both subscribers (using e.g. USIM) and platforms (using e.g. IMEI-code). If a trusted mobile platform is designed to identify itself only to a cellular network, there is no privacy advantage from EK-based AIK enrollment, and such a platform can be issued with preassigned AIKs (instead of an EK). It follows that a DM engine may be pre-installed with an EK and associated credentials, or may be pre-installed with an AIK (or AIKs) and associated credential.

Both EK and AIKs are still necessary if engines require privacy when signing information. This is the case, for example, if an engine is designed to communicate with multiple entities and there is no legal requirement to unambiguously identify the engine to all those entities.

End of informative comment

If a trusted engine boots, it MUST always use an authenticated boot mechanism, so integrity metrics are available once the platform has booted.

Start of informative comment:

If a platform uses secure boot, there are two fundamental phases. The first phase starts with engine initialization and ends when an engine is able to protect itself against subversion. During the first phase, software must always be verified before it is executed. Otherwise the engine may be subverted. Typically a fully-linked list of software is required. The second phase starts when an engine is able to protect itself against subversion and continues until the platform is reinitialised. During the second phase, it is unnecessary to check all software before execution. The software might not be checked, or the platform state before execution of the most recent software, plus the most recent software, could both be verified. Alternatively, the platform state after execution of the most recent software could be verified.

This specification does not mandate any particular secure boot mechanism. A particular secure boot mechanism may, however, incorporate the following basic states:

- *Engine Reset*. This is the state from which an engine starts. No services are running in the engine (e.g. no software is running). All volatile memory is reset. The engine transitions from the Engine Reset state to the Engine RoT Initialisation state.
- *Engine RoT Initialisation*, where the engine ensures that the RoTs are operational. If a RoT is built from dedicated resources, it must indicate that it has passed self test. If a RoT is built from allocated resources, it must be properly built by a trusted resource and may (in some circumstances) be measured. This topic is discussed in more detail in section 4.1.3 "The Roots of Trust". When the RoTs are operational, the engine transitions to the *Engine-Loading* state. Otherwise the engine must transition to the *Engine Failed* state.
- *Engine-Loading*. In this state the engine loads programs (such as components of an OS or hypervisor) and verifies each program before loading it. Verification is necessary because the platform is not yet in a state where it can isolate existing process from new programs. If the engine fails to verify a program being loaded then the engine transitions to the *Engine Failed state*. When the engine is in a state where it can isolate processes, the engine transitions to the *Engine Verified* state.
- *Engine-Verified*. In this state the engine is operating according to policy. If the engine loads a program that affects the isolation abilities of the engine, the engine must first transition to the *Engine-Loading* state. Otherwise, if the engine loads a program that does not affect the isolation abilities of the engine, the engine can remain in the *Engine-Verified* state.
- *Engine Failed*. In this state all trusted resources must be operational and configured, but the engine's capabilities may be deliberately restricted. Some remedial action, such as a reboot, may be necessary before the engine is once again operating according to policy.

The requirement necessary to support the *Engine-Loading* state is that all software executed on an engine before the engine can isolate existing processes from new processes, and vice versa, must be verified before execution. The companion specification "TCG Mobile Trusted Module Specification" describes primitives that may be used to record verification of a fully-linked list of software.

End of informative comment.

All software executed on a secure-boot engine before the engine can protect existing processes from new processes, and vice versa, MUST be authenticated before execution.

Start of informative comment:

A simple credential called a Reference Integrity Metric (RIM) certificate is used to describe an approved state of a platform and / or approved integrity metrics. A RIM certificate can be used to: (1) verify software; (2)

verify the state of a platform and verify software before executing the software; (3) verify the state of a platform.

Until mobile trusted platforms are the norm, it is anticipated that RIMs will be delivered to platforms in "external" certificates, via proprietary or extant protocols, and with proprietary or extant certificate formats. When a platform receives an external certificate, a RIM conversion agent verifies the external certificate and converts it to an internal certificate. The conversion from external to internal involves checks that defend against a variety of other attacks. Converting to an internal certificate with a TCG-defined format has two advantages: (1) verifying an internal certificate is simpler than verifying an external certificate because less attacks can be mounted on internal certificates and hence less defenses are required; this reduces the total time to verify frequently used RIM certificates; (2) eventually, when mobile trusted platforms are the norm, a standardized infrastructure protocol could deliver (external) RIM certificates with internal certificate formats.

External certificates require a platform to store a global monotonic counter value as a defense against off-line replay attacks.

Internal RIM certificates are customised for a particular platform by replacing all external RIM certificate protective data with internal RIM protective data. (This involves a current monotonic counter value and a signature value using the engine's internal verification key, stored in the engine's RTS.)

End of informative comment

External certificates MUST be used when corresponding internal certificates are unavailable: when a secure-boot engine boots for the first time and has not yet reached the stage when it can execute a RIM conversion agent, for example.

Each RTS in a secure-boot engine MUST have access to at least one persistent key that is the root of a key hierarchy for verifying a RIM_Auth certificate hierarchy. Each key in the key hierarchy MUST be used to verify other keys or a RIM certificate. The root key MUST be either immutable or MUST be used to verify its replacements. The RTS MUST store either the root key or a means to recognize the root key (its digest, for example).

Start of informative comment:

The root key or its digest can be embedded in an RTS (during manufacture, for example) if the RTS has protected non-volatile memory. Alternatively, the root key or its digest can be loaded into the RTS during platform boot, before rogue software can execute on the platform. This enables the root key or its digest to be stored in protected non-volatile memory outside the RTS, and loaded into protected volatile memory inside the RTS every boot cycle.

The RTS has a record of dedicated PCRs that can be extended only with a field from a verified RIM certificate. The list of these dedicated PCRs can be embedded in an RTS (during manufacture, for example) if the RTS has protected non-volatile memory. Alternatively, the list can be loaded into the RTS during platform boot, before rogue software can execute on the platform. This enables the list to be stored in protected non-volatile memory outside the RTS, and loaded into protected volatile memory inside the RTS every boot cycle.

Certain dedicated PCRs can be extended only with a field in a verified RIM certificate. Specifically, a RIM certificate can contain a digest, a PCR index, a PCR_COMPOSITE value, and a signature value. If the signature value is verified using an internal RIM verification key or a key from the RIM certificate hierarchy, AND the current MTM PCR values match the stated PCR_COMPOSITE value, THEN the RTS will extend the digest value into the PCR with the PCR index. This primitive has two desirable properties.

- It simplifies recognition of an engine that has (or had) a particular software state: a RIM certificate can contain a digest of a signed statement (instead of a digest of an actual measurement) and a PCR_COMPOSITE value indicating a particular boot process. Any PCR that was extended via that RIM certificate is therefore a reliable indication that the engine had a particular software state, such as completion of a particular boot process, and can be verified by checking the signature on the statement that was extended into the PCR. Data can be sealed to that single PCR, and the value of that single PCR can be reported in an integrity challenge. If the boot process is modified, the PCR_COMPOSITE value in the authorizing RIM certificate must change but sealed data and remote attestation does not change. Note that this property is independent of whether an engine is secure-boot.

- It is a locality mechanism that prevents Denial-of-Service attacks on selected PCRs during the second phase of a secure boot process: those selected PCRs cannot be extended unless the integrity metric is approved and the engine is in the approved state.

End of informative comment

4.1.2 Reference Engine

Start of informative comment:

Each fully booted engine in a trusted mobile platform is an adaptation of a single generic engine, shown in Figure 3.

End of informative comment

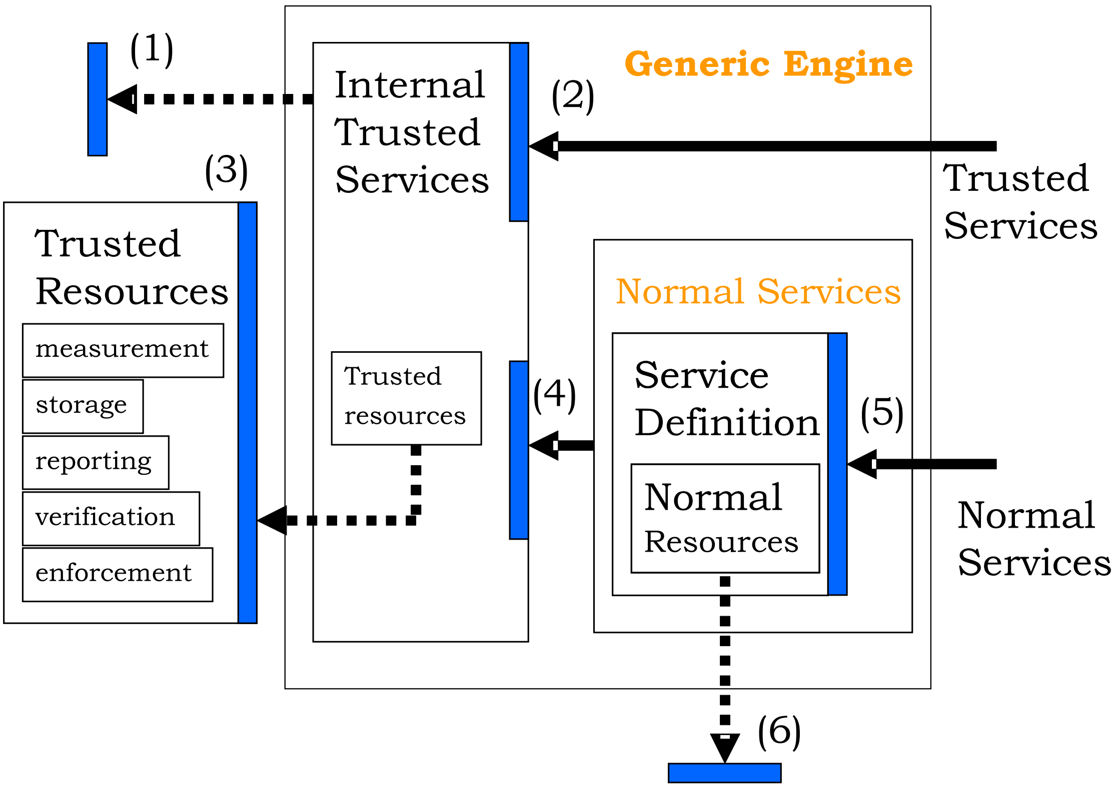

Figure 3. Generic Engine.

Start of informative comment:

In Figure 3, the solid rectangles indicate interfaces, the solid heavy arrows indicate dependency, and the dotted heavy arrows indicate optional dependency (in both cases the arrow pointing away from the dependant entity). Services are based on Resources or other Services. Resources are either provided within an engine or provided as a service by another engine.

The Trusted Resources are:

- A Root-of-Trust-for-Measurement (RTM), which performs the measurement functionality as defined in the TPM Main Specification.

- A Root-of-Trust-for-Storage (RTS), which provides PCRs and Protected Storage for an engine. The RTS stores the measurements made by the Root-of-Trust-for-Measurement (or its subsequent measurement agents), cryptographic keys, and security sensitive data.
- A Root-of-Trust-for-Reporting (RTR) , which reports the measurements stored by the Root-of-Trust-for-Storage by signing them with cryptographic signature keys stored by the Root-of-Trust-For-Storage.
- A Root-of-Trust-for-Verification (RTV), which checks measurements against reference integrity metrics before they are extended into the RTS. The RTV may verify the measurements of the current state of other engines. The RTV can reliably verify a measured integrity metric against a RIM, and extend the integrity metric into a Platform Configuration Register (PCR)
- A Root-of-Trust-for-Enforcement (RTE), which is responsible for building all the Roots-of-Trust in its own engine which are based on allocated resources. The RTE is described in more detail in section 4.1.3 “The Roots of Trust” of this specification.

End of informative comment

RoTs MUST be dedicated RoTs or allocated RoTs or services (interface #3) exported by another engine. At least one RoT somewhere in a mobile trusted platform MUST be a dedicated instantiation with a preassigned EK or AIK because there will be nothing on the platform to measure it. These RoTs MUST be instantiated and supplied in a way that guarantees that their EK or AIK remains secret.

Start of informative comment:

Not all engines necessarily have all Roots-of-Trust. An engine would need an RTE to guarantee that allocated RoTs will be booted, but not otherwise. An engine would need an RTV to guarantee that mandatory allocated resources are running properly, for example, but not necessarily otherwise. An engine always, however, needs an RTM, RTS and RTR because they are required for the measurement and reporting of integrity metrics.

Internal Trusted Services measure (interface #4) Normal Services and provide other services (such as Protected Storage, Monotonic Counters, Delegation, etc.) to the Normal Services. The Internal Trusted Services report (interface #2) measurements (signed by AIKs) to external entities, and use those measurements internally (for sealed data). Internal Trusted Services have internal access (interface #3) to a set of Trusted Resources.

End of informative comment

The Internal Trusted Services MUST be instantiated either by resources belonging to the engine or by Trusted Services from (interface #1) another engine. The Internal Trusted Services MAY export (interface #2) Trusted Services that will be the Internal Trusted Services for another engine.

Start of informative comment:

Exported Trusted Services need not be a duplicate of Internal Trusted Services.

Normal services (interface #5) are instantiated by customising Normal Resources with a Service Definition.

End of informative comment

The Normal Resources MUST be instantiated either by resources belonging to the engine or by Normal Services exported (interface #6) by another engine and supplied with measurements and /or certificates of their trustworthiness.

Start of informative comment:

Normal Services are measured by Internal Trusted Services via interface #4.

End of informative comment

Internal Trusted Services MUST be isolated from Normal Services.

Start of informative comment:

Distinct services within Normal Services might need to be isolated from each other.

The stakeholder of an engine dictates what services are provided by the engine, and how the engine boots. The stakeholder does this by customising the RTE, which determines how the engine boots if the engine is supplied in an unfinished state, and / or by customising the RTV, which determines whether a completed engine is permitted to operate.

Each engine can vouch for its measured services. It measures arbitrary services in normal TCG fashion (using an RTM and /or Measurement Agent), signs those PCR values using an AIK, and attaches the signed PCR value to its service. The AIK is used (either directly or indirectly via a session key) to sign all arbitrary data exported by a service. Any trusted service (or the RTV) in a receiving engine verifies the signed PCR values that describe the service, and verifies the signature on all data imported by the service.

Service verification may be simplified if a service executes in an engine on a highly integrated platform. If the Device Manufacturer's engine instantiated a complete service in an allocated engine, the DM's engine can sign credentials describing the PCR values that the service should produce, and send those credentials to the receiving engine. Further simplification may be possible if the receiving engine was also instantiated by the DM's engine. If the DM's engine "knows" that the measured engine is operating properly, the DM's engine could instantiate a protected communication path between the measured engine and the receiving engine, and the receiving engine can implicitly trust that information received via that path was sent by a properly executing service.

There are clearly many design options, including.

- Trusted resources may be provisioned with EKs and / or with AIKs
- AIKs belonging to an exported Trusted Resource might be AIKs belonging to an imported Trusted Resource.
- Exported Trusted Services could be a complete separate software instantiation of Internal Trusted Services, or equally well be a software wrapper (shim) that calls Internal Trusted Services.
- The PCRs in exported Trusted Services could be distinct registers or derived from PCRs in an engine's imported Trusted Services.
- Internal Trusted Services and exported Trusted Services might have different interfaces. Internal Trusted Services in an engine could be low-level (device-level) commands and exported Trusted Services in that engine could be TSS-level commands, for example. In one implementation of a highly integrated platform, the Internal Trusted Services in the Device Manufacturer's engine could have a low-level interface and the exported Trusted Services have a high-level interface. This permits the DM's engine to get its Trusted Resources from a dedicated resource (chip) but provide exported Trusted Resources for application level software.

End of informative comment.

4.1.3 The Roots of Trust

Start of informative comment:

Resources cannot be trusted unless they are supplied with an EK / AIK and associated credentials, or are measured by a trusted resource. But some of the RoTs are necessary to make and store measurements. Therefore there arises the question of when a RoT may be measured, and (indeed) "what qualifies as a RoT?".

The fundamental axioms governing the RoTs in a trusted mobile platform may be summarised as follows:

If any entity is mutable other than by the proper authorised entity, it cannot be a RoT. This does not mean that RoTs must be immutable. A RoT can be mutable provided the rest of the engine never needs to check the construction of that RoT. Note that this does not preclude trusted resources in an engine checking components before using them to build an allocated RoT.

RoTs composed of allocated resources can't be used to test those allocated resources.

RoTs composed of dedicated resources must perform a self-test and must shut-down if the test fails. Accordingly, if the RTE builds any other RoT, the RTE must pass a self-test before starting normal operation. If any RoT is not built by the RTE, it must pass a self-test before starting normal operation.

End of informative comment

For uniformity of reporting, measurements of the RTE, RTM, RTV, RTS and RTR MUST always be stored in the RTS. In cases where explicit measurement would be circular and have no security value (such as a measurement of the RTM, or of an RTE used to build the RTM or RTS), then the measurement values MAY be supplied with a RoT by the RoT's supplier and passed to the RTM on request. For example, such a pre-supplied RoT “measurement” may just consist of a component label (like a MTM Manufacturer Name and Part Name), and a component version number. Alternatively, such measurements MAY consist of actual values obtained while an allocated RoT was built. When possible, these measurements SHOULD be compared against a signed value provided by each RoT's supplier.

Start of informative comment:

At least the RTM, RTV and RTS in an engine must exist before that engine can make measurements.

End of informative comment

If any of the RTM, RTV and RTS in an engine are to be built from allocated resources, they MUST be built by trusted resources.

Start of informative comment:

It follows that a special trusted dedicated entity is required to build any of the RTM, RTV and RTS using allocated resources. That entity is the RTE. IF the RTE fails to build the allocated Roots-of-Trust in its engine THEN that engine is in a state that is outside the scope of this specification; an attack or failure has occurred that is beyond the capabilities of the mechanisms implicit in this specification.

An RTE is not required if the Roots of Trust of the Engine are provided as dedicated resources, or the Engine starts execution fully built (e.g. built by the Device Manufacturer’s Engine).

End of informative comment

If an Engine builds any Roots of Trust from allocated resources, the Engine MUST have an RTE. The RTE SHALL build all Roots-of-Trust of its engine that are based on allocated resources.

At least part of an RTE MUST consist of dedicated resources. An RTE therefore MUST be supplied with an EK and / or AIK plus relevant credentials. The RTE MAY be immutable or may be mutable, but its integrity MUST always be intact: the RTEs integrity and authenticity MUST be maintained during the lifecycle of the platform. The RTE’s supplier MUST ensure that the RTE’s authenticity and integrity are preserved when the RTE is supplied and / or changed. The methods of supplying and changing the RTE are outside the scope of this specification but replacement or modification MUST be performed only by an agent and method approved by the RTE supplier. This requires a manufacturer to implement an upgrade method that is compatible with the security properties of the Platform's Protection Profile.. **If** the RTE changes during the lifecycle **then** the supplier MUST make sure that no rollback attacks can occur after an RTE has been upgraded.

The RTE MAY be customized by the engine’s stakeholder in order to dictate the trusted services and resources that have to be present. In this case, the RTE MUST contain a list of these services and resources. Recustomisation of the RTE is outside the scope of this specification but the RTE’s supplier MUST provide means that preserve the control of the RTE by the engine’s stakeholder.

An engine’s RTS and RTR MUST comply with the requirements of the specification “TPM Main Part 1 Design Principles” Section 3 “Protection”. This describes Protected Capabilities and Shielded Locations.

A RTM MUST accurately measure the first software that is executed on the platform and MUST reliably record the result in the RTS.

A Device Manufacturer's Engine MUST incorporate a RTM, RTS, RTV, RTR and MAY incorporate an RTE. Other Engines MUST support at minimum the RTM, RTR and RTS. For each Engine (with the exceptional case of an engine built entirely from dedicated resources), there MUST be one or more measurement events, and where the RTV exists, verification events.

4.1.4 Physical Presence in a mobile trusted platform.

Start of informative comment:

"Physical Presence" in trusted platforms has three purposes. The first is to provide a safety net, to regain control of a trusted platform when an Owner's authorisation value is unavailable. The second is to authorise a TPM command in a way that cannot be subverted by rogue software. The third is to enable a human User to temporarily deactivate a TPM. All of these cases reduce to the authorisation of TPM commands by a (human) Operator via physical access to a trusted platform.

Physical Presence must not be supported in a mandatory engine. There are several reasons:

- The entity that determines the security properties of a mandatory engine is assumed to be an absent (physically remote) entity.
- A human Operator of a trusted mobile 'phone is not permitted to control the mandatory engine or to deactivate the MRTM in a mandatory engine. Such control is incompatible with regulations and contractual agreements.
- Owner authorised commands may not be implemented in a mandatory engine (the stakeholder may have taken Ownership before the platform was shipped, for example).
- Owner authorised commands may always be submitted to a mandatory engine by entities who prove privilege via TCG's delegation mechanisms.
- A mandatory engine's Owner authorisation value may be robustly backed-up, and will never be lost.

Physical Presence is required in a discretionary engine because:

- The entity that determines the security properties of the discretionary engine is not necessarily physically remote.
- A human Operator of a trusted mobile 'phone is permitted to control the discretionary engine and deactivate the MTM in the discretionary engine, for reasons of privacy which are compatible with regulations and contractual agreements.
- Owner authorised commands must be implemented in a discretionary engine to permit TPM_takeOwnership and the acquisition of AIKs.

The Owner authorisation value could be lost by the engine's Owner.

End of informative comment

A trusted mobile 'phone MUST NOT implement means of controlling a mandatory engine via Physical Presence.

A trusted mobile 'phone MUST implement means of controlling a discretionary engine via Physical Presence.

A manufacturer MAY implement Physical Presence in any way but all indications of Physical Presence MUST accurately represent detection of appropriate physical interactions with the platform. It is RECOMMENDED that the Device Manufacturer's engine detects Physical Presence and provides appropriate indications to other engines and to the engine's Owner.

4.2 Overview of Measurement

4.2.1 Measurement Components

Start of informative comment:

"Measurement" in this specification is defined in the same way as the term "integrity metric measurement" in the TPM Main specification. "Verification" is defined as comparing the actual result of any such measurement – termed a Target Integrity Metric, or "TIM" – with an expected value of that measurement – termed a Reference Integrity Metric, or "RIM".

There are essentially two components involved in the measurement process.

1. One or more measurement agents, each of which is securely connected to all components of the platform that need to be measured by that agent.

The components to be measured include basic hardware components, which the measurement agent can interrogate to receive status reports. In that case "measurement" simply consists in recording the hardware name and version number, and the results of self-tests performed by the hardware.

The components to be measured also include storage media like memory cards (or in some mobile devices hard disks), non-volatile memory (ROM, EEPROM), and volatile memory (RAM). The measurement agent must have sufficient read access to these storage media and memory units to determine the contents at any specified area of the storage media/memory that it needs to measure.

The measurement agent runs executable code which implicitly or explicitly gives a list of things to measure – these are the Target Objects of measurement. The measurement agent's stored list of measurements is called its "measurement configuration data". Both the executable code and list of Target Objects should be protected from tampering (i.e. held in tamper-evident storage), or at minimum the measurement agent should be able to detect tampering and flag an error i.e. some form of checksum or signature verification key of the expected code and Target Objects is held in tamper-evident storage. (Alternatively, the configuration data may have been measured and verified at an earlier stage of boot. This is a feature of the transitive boot process discussed in Section 5.4.)

2. A place to store the measurements. This functionality is provided by the set of Platform Configuration Registers (PCRs) of a Mobile Trusted Module. The RTM + other measurement agents use the RTS to store measurements, so the measurement agent will therefore need to have write access to the MTM's PCRs. As each PCR is expected to only receive a small amount of input data, the value of each object measured is hashed before extending into a PCR.

In the discussion which follows, a general measurement agent is distinguished from the very first measurement agent running within any Engine on the platform. This first measurement agent is the Root of Trust for Measurement of that Engine.

How tightly connected the measurement agent is to the objects of measurement will depend somewhat on when that agent is intended to execute. A measurement agent which executes soon after power-up (e.g. RTM) will be expected to have physical interfaces to the measured components, as the full set of software interfaces (device drivers, OS) has not been loaded yet. Later running measurement agents can assume that software connections to the objects of measurement already exist and are already known to be trustworthy, and thus the measurement agent is a custom piece of software on the OS running under an account with appropriate read privileges.

Similar considerations apply to the connection to a MTM. An early-running measurement agent might be able to send a direct MTM command over a physical bus; otherwise it might need to interact with a TPM through some part of the TSS. For example, the measurement agent may be able to use the TCG Device Driver

Library, or the TCG Core Services Interface (Tcsip_Extend, Tcsi_LogPcrEvent). Or if it is even higher level, the TCG Service Provider Interface (Tspi_TPM_PcrExtend).

As the platform is able to support several Engines, there may be dependencies between measurement agents in different Engines. For instance, Engines may be *chained* : one Engine executes, then terminates, but starts up another Engine etc. In that case, the RTM of a later Engine in the chain may be first measured by a separate measurement agent running in an earlier Engine of the chain.

Alternatively, Engines may be *nested*, so that all later Engines run as applications within the Device Manufacturer's Engine (which also continues running other functionality). In that case, whole nested Engines could be measured at boot-time by a measurement agent running in the Device Manufacturer's Engine, and the RTM of the nested Engine may not have to measure anything itself at boot-time (just make measurements during run-time).

***End of informative comment*.**

4.2.2 Interfaces to the Roots of Trust in a Secure Boot Engine

Start of informative comment:

For any given Engine, the RTS and RTR are Roots of Trust as defined in the TCG specifications "TPM Main Part 1 Design Principles", "TPM Main Part 2 TPM Structures", and "TPM Main Part 3 Commands". The RTM is a platform-specific Root of Trust that is defined in this specification (Section 4.1 and section 4.2.1 above). The RTV and RTE are new RoTs that are defined in this specification (Sections 4.1, 4.2.1 above and Section 5.1 below).

Figure 4 below illustrates the interfaces to the Roots of Trust during measurement and verification.

***End of informative comment*.**

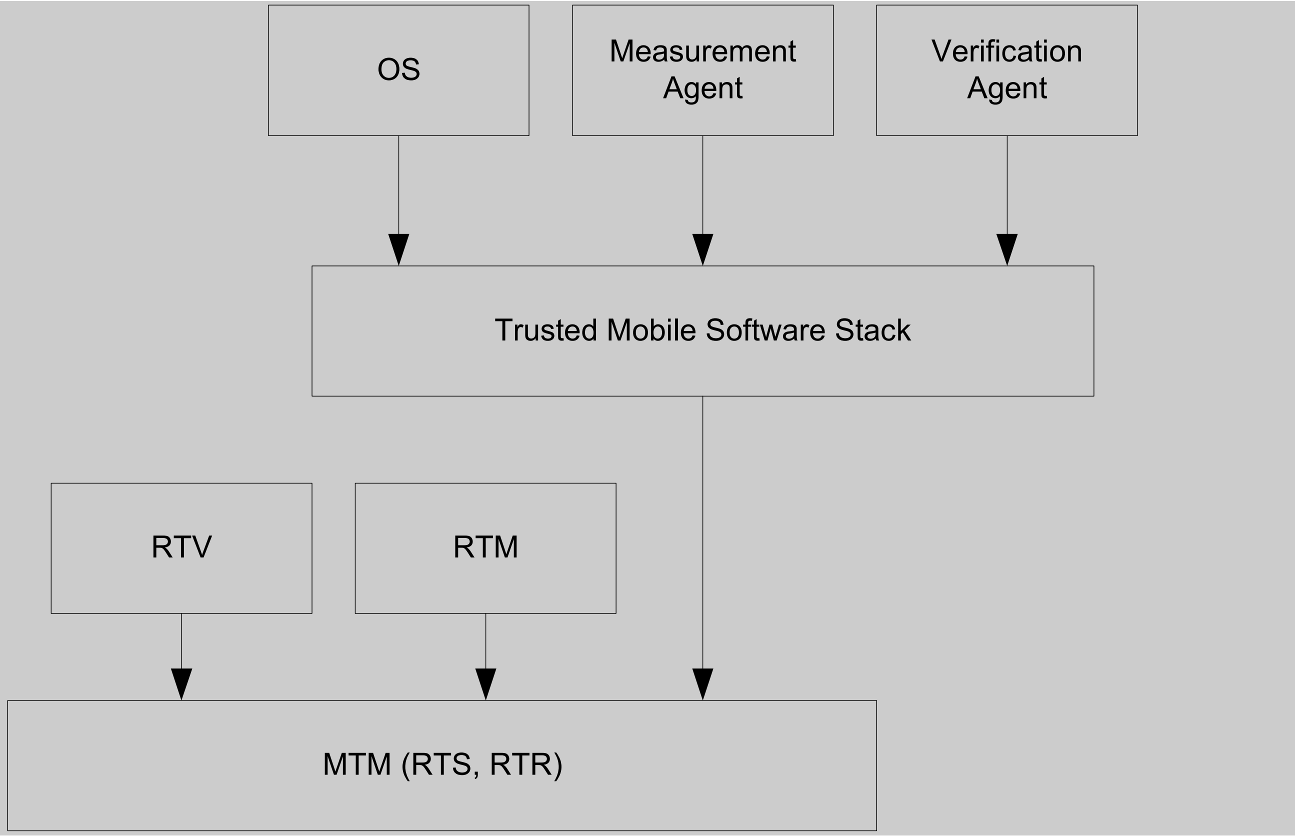

Figure 4. Interfaces

Start of informative comment:

At the bottom is the Mobile Trusted Module. This block contains the RTS and the RTR, and is much like a PC style Version 1.2 TPM. However, it provides some additional commands as listed in "TCG Mobile Trusted Module Specification", and some PC Version 1.2 TPM commands may be absent.

On top there is a point of entry called the Trusted Mobile Software Stack (TMSS). This stack is not explicitly defined in this specification, but is at about the same level as the TSS Core Services for PC platform. On top of this the 'users' of the trusted services are shown. These are the OS, Measurement Agents, and Verification Agents. Note, that all of those agents might actually be part of the OS, and there will be interactions between the OS and these agents, which are not depicted above. It is also important to note that in the early stages of the boot process, entities like an RTV and RTM may directly interact with the MTM without going through a TMSS.

End of informative comment.

4.3 One possible platform instantiation

Start of informative comment:

A commercial trusted mobile platform should be cost-effective. In this example, most functions that can be a software construct are a software construct; hence it could be the basis for a commercial trusted mobile platform. This design behaves the same as a more expensive platform built with extensive hardware support, albeit with lower strength-of-function, because hardware isolation is stronger than software isolation. There are of course many other platform instantiations than in this example, and this example is simply designed to illustrate the usage of an MTM.

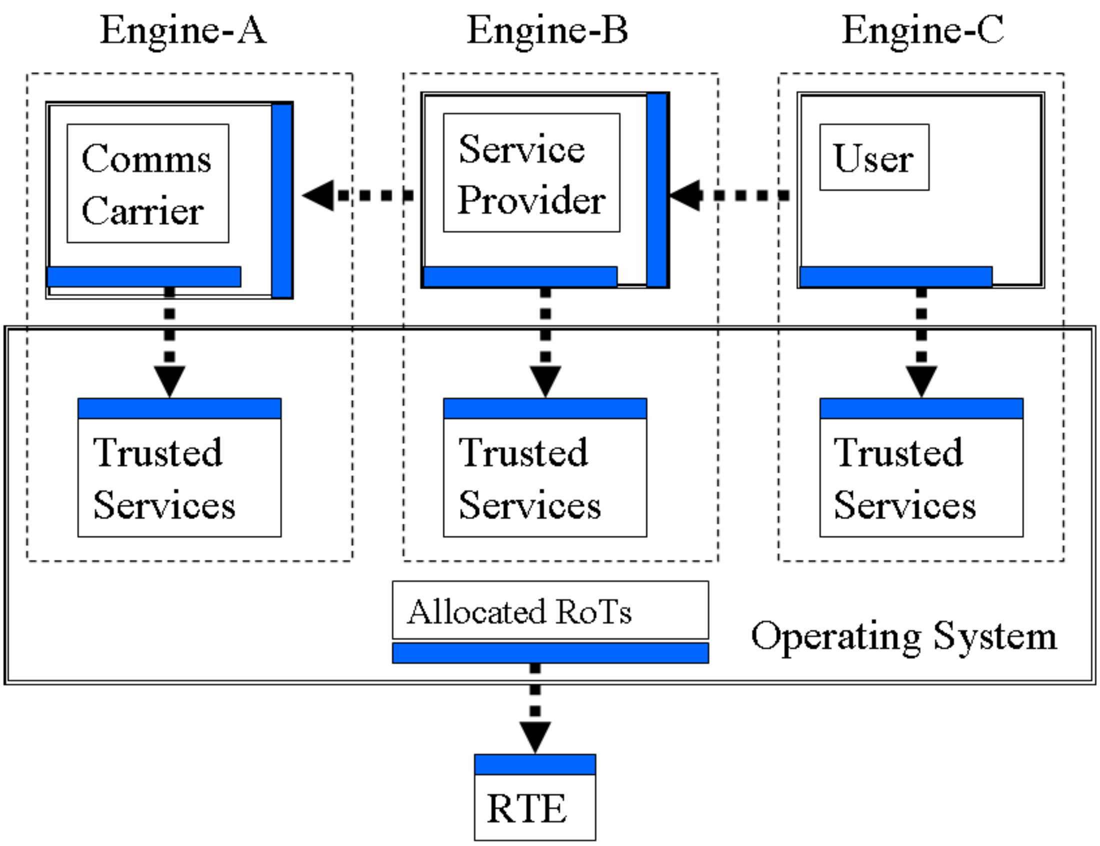

Figure 5. Example Platform

Figure 5 illustrates a platform composed of a DM's engine and trusted engines A, B and C. The platform boots from a dedicated RTE that creates an allocated RTM, RTS, RTR and RTV for an allocated Device Manufacturer's engine (an Operating System). The OS provides conventional OS services and also provides software compartments that instantiate trusted engines A, B and C. These are isolated sets of applications and trusted services. All the trusted services consumed by engines A, B and C are provided as a service by the DM's engine (the OS).

In Figure 5, the trusted engines A, B, and C execute applications on behalf of a Communications Carrier, Service Provider, and User. The Device Manufacturer's engine provides radio access only to the Communications Carrier. The Communications Carrier engine provides network access for the Service Provider engine, and the Service Provider engine provides an address book service for the User. The User engine stores the User's data. These services are communicated via the OS, which ensures that services come from the correct source and go to the correct destination. Each application is provided by the relevant stakeholder and stored in unprotected NV storage in the platform during manufacture.

In this example, only the Device Manufacturer's engine has access to basic platform resources, which comprise a computing engine with a user interface, debug connector, a radio transmitter and receiver, Random Number Generator, and SIM interface. The computing engine has a protected execution environment as well as a less privileged execution environment. The protected execution environment is used for critical security functions, in this case the Roots-of-Trust and an OS kernel. The Roots-of-Trust are (of course) implemented as Protected Capabilities and Shielded Locations, defined in the "Protection" section of "TPM Main Part 1 Design Principles". In this design, the Roots-of-Trust and OS kernel can co-exist without explicit separation mechanisms because the OS kernel is designed never to read memory used by the Roots-of-Trust. The less privileged environment accesses the protected environment exclusively through a defined API, and cannot subvert the protected environment. Both environments have access to volatile storage. The protected environment has access to write-once non-volatile storage in the form of ROM but has no access to protected write-many non-volatile storage. The protected environment therefore always boots into the same fixed state when power is completely removed and then reapplied.

The protected environment boots from a dedicated RTE. On boot, the RTE builds an allocated RTM, RTS, RTR and RTV in the protected environment, and then starts the kernel of an Operating System in the protected environment. The kernel builds an OS in the less privileged DM environment.

The OS provides normal OS services and also provides isolated execution environments that instantiate and isolate other trusted engines. The trusted engines are built according to specifications provided by the Device Manufacturer, designed to satisfy the requirements of the other platform stakeholders. The trusted services in those other engines are provided as a service by the OS. Applications (ie. measured services) executing in different trusted engines are therefore isolated from each other by the OS, are measured and reported via the OS, and receive TCG trusted services (such as Protected Storage functions and time stamping functions) from the OS. The OS also provides AIKs for each of the other trusted engines.

The Trusted Services API is currently undefined and is therefore currently manufacturer specific. A Trusted Services API may be defined in future TCG specifications.

***End of informative comment*.**

5. Measurement and Verification

As discussed in Section 4.1.2, engines in a mobile platform are generally **not** REQUIRED to support an RTV. All normative statements in this section referring to the RTV, Verification Agents, RIMs, RIM_Certs, RIM_Auths and Validity Lists are applicable to an engine *conditional* on that engine supporting an RTV.

The DM's Engine MUST support an RTV, and any other engines with remote owners (i.e. owners who are not local Users of the platform) SHOULD support an RTV. An engine with a local owner (i.e. a User engine) MAY support an RTV, but in that case the local owner SHALL have full control over what measurements within the engine are verified, and what are considered "correct" values of those measurements.

All engines MUST support an RTM, RTS, RTR and a transitive chain of measurement, to provide an authenticated boot. Hence statements in this section referring to the RTM and Measurement Agents are applicable to all engines.

5.1 Root-of-Trust-for-Verification and Verification Agents

5.1.1 Measurement Verification

Start of informative comment:

In the Mobile Phone platform, additional functionality to normal TCG measurement is provided by **verifying** the results of measurement. This verification can be done as the measurements are performed, in a model which can be described as "Measure → Verify → Extend".

Each verification agent works in close conjunction with a measurement agent, and is called immediately after each measurement to be verified. In fact the two agents may well be implemented together. The very first verification agent for an Engine should be conjoined with the RTM for an Engine and is called a Root of Trust for Verification (RTV).

To perform verifications, the measurement agent – for example - retrieves from a stored list each item to measure (called a target object) and makes the actual measurement (TIM). Then the verification agent retrieves the corresponding expected result of the measurement, the Reference Integrity Measurement (or RIM).

Each RIM may also be provided with the expected value of one or more PCRs before the PCR extend has occurred. This acts as a check that previous intended measurements have been performed (i.e. none were skipped), that they were performed in the right order, and that the collective results were as expected.

Note that the RTM and RTV are heavily dependent on secure storage within the Engine (provided by the Root of Trust for Storage). This gives rise to a couple of plausible implementation models.

- The RTS and RTR are implemented together as one unit within an Engine, while the RTM and RTV are implemented together as another unit within the engine.

  In this case, an interface is needed between the RTM/RTV and RTS/RTR to create and retrieve RIMs and to extend measurements. The combined RTS and RTR are what is called the "MTM" and the interface is defined in "TCG Mobile Trusted Module Specification" .

- RTS, RTR, RTM, and RTV are all implemented together as a common unit within the engine.

  In this case, it is not necessary to expose an interface to a "MTM" when implementing the RTM and RTV, just ensure that this interaction between the roots of trust functions correctly.

End of informative comment.

5.1.2 Operation

The Root-of-Trust-for-Verification verifies measurements of the components of its engine against Reference Integrity Metrics (RIMs). Hence, the Root-of-Trust-for-Verification MUST have access to an integrity protected list of RIMs, contained in RIM Certificates. For details about RIMs and RIM Certificates, see "TCG Mobile Trusted Module Specification" [5].

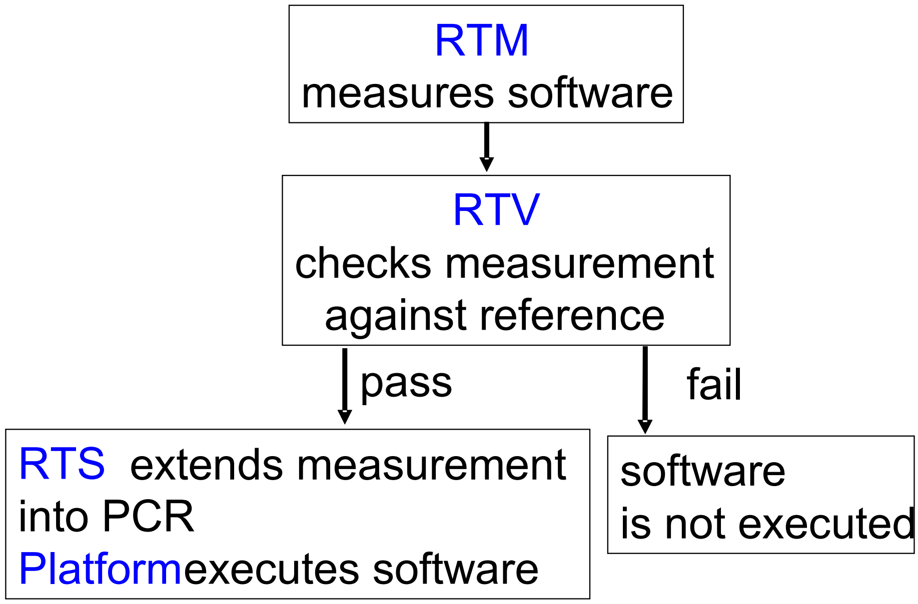

Figure 6. Relationship of RTV and RTM

Start of informative comment:

Figure 6 illustrates this process. The RTV (not the RTM) does the extend operation into a MTM. The RTM measures events and then passes the measurement to the RTV. The RTV compares the measured value against a reference integrity metric (RIM) contained in a RIM Certificate. If the values match, the RTV attempts a **verified** extend (using MTM_VerifyRIMCertAndExtend, [see "TCG Mobile Trusted Module Specification"] of the measured value into the RTS (and the measured software is permitted to execute). Otherwise, if the measured value is different to the reference value, or if the verified extend fails, then the measured software is not executed.

End of informative comment.

At later stages of boot, the Root-of-Trust-for-Verification SHOULD pass the responsibility for verifying measurements to other Verification Agents (see Section 5.5). The RTV MUST either pass the responsibility for verifying measurements to other verification agents, or be the sole verification agent throughout the boot process. The interaction between such a Verification Agent and the associated Measurement Agent is exactly as shown for the RTV and RTM above. Requirements in 5.1.3 concerning configuration, upgrades and customization of the RTV also apply to other verification agents: however these requirements for other verification agents SHALL be met using a transitive chain of trust from the RTV (see Section 5.4).

An instantiation of the Root-of-Trust-for-Verification SHALL do the following:

1. The RTV SHALL be resistant to all forms of software attack and to the forms of physical attack implied by the platform's Protection Profile.

2. The RTV SHALL supply an accurate report of the availability of the corresponding Reference Integrity Metrics.
3. The RTV SHALL supply an accurate report of the relationship (equal or not equal) between the measurements and corresponding Reference Integrity Metrics.
4. Upon verification failure, the RTV SHALL trigger the transition of the Engine to a FAILED state. The RTV MUST NOT continue the transitive trust boot process in the same manner as if there is no failure.

5.1.3 Configuration

The Root-of-Trust-for-Verification of an engine SHALL be provisioned by the stakeholder of that engine. The supplier of the Root-of-Trust-for-Verification SHALL be responsible for the security of the provisioning process, i.e., the supplier MUST make sure that authenticity and integrity of the Root-of-Trust-for-Verification are preserved.

5.1.3.1 Upgrades

The supplier also SHALL be responsible for secure upgrades of the Root-of-Trust-for-Verification, i.e., for the authenticity and integrity of the provisioning of a new version of it and its credentials. The supplier SHOULD establish a mechanism in order in allow for authentic and integral upgrades of the Root-of-Trust-for-Verification.

Start of informative comment:

There are different mechanisms to achieve a secure upgrading process: the supplier may dictate that upgrades can only be done in a secure environment (i.e., he may provide organisational and logistical means to achieve security) or he may implement a cryptographic authentication mechanism into the Root-of-Trust-for-Verification so that it can identify authorized upgrading requests.

End of informative comment.

The supplier MUST make sure that no rollback attacks can occur after the Root-of-Trust-for-Verification has been upgraded.

5.1.3.2 Customization

The Root-of-Trust-for-Verification SHALL be customized by the engine's stakeholder in order to dictate which services have to be measured and against which RIMs the measurements have to be verified.

If the stakeholder wants to re-customize the Root-of-Trust-for-Verification at a later point in time he MAY replace the whole customisation by a new version of it or only components of the customisation (for instance, he MAY only want to change the lists of services which have to be measured or the list of RIMs and RIM Certificates).

In any case, the stakeholder SHALL be responsible for preserving authenticity and integrity of this process.

The stakeholder MUST make sure that no rollback attacks can occur after the list of measured services or the list of RIMs and RIM Certificates have been changed.

5.2 Secure Provisioning of RIMs

Start of informative comment:

The access of the Root-of-Trust-for-Verification to the integrity protected list of RIMs is very security sensitive. The stakeholder of the engine has to make sure that Root-of-Trust-for-Verification has access to Reference Integrity Metric values in a way so that authenticity and integrity of them are preserved. (In fact, the same requirements apply for accessing the RIMs of any Verification Agent.)

This requirement has two aspects:

- The stakeholder has to securely deliver the Reference Integrity Metrics. That is, the stakeholder has to make sure that the integrity during the provisioning process is preserved and that the RIMs cannot be modified in an unauthorized way.
- The stakeholder has to provide a mechanism that ensures the security of the access of the Root-of-Trust-for-Verification to the RIM values with respect to their authenticity and integrity.

These two aspects are met, respectively, by *external* RIM Certificates, and *internal* RIM Certificates.

The stakeholder has to customize the Root-of-Trust-for-Verification at least with an authorized list of RIM values and their description. The stakeholder should also provide a mechanism to update RIM values so that their authenticity and integrity is preserved. When doing so, the stakeholder has to make sure that no rollback attacks can occur.

End of informative comment.

The Security Properties that a RIM provisioning method MUST support can be summarized as follows:

1. Source: authentication, authorization, integrity

   The Engine MUST be able to correctly establish the source of the RIMs being provisioned to the Engine. Further, having established the source, the Engine MUST be able to decide whether that source is authorized to supply RIMs. In addition, it MUST be able to determine that the RIMs have not been corrupted since leaving that source. If this determination is not possible, the Engine becomes susceptible to *Impersonation* or *Man In The Middle Attacks*.

   A related issue concerns protection of RIMs once they are installed on the Device. The Engine MUST be able to determine if its installed RIMs have been completely removed by an attacker and replaced by an unauthorized set. If this determination is not possible, the Engine becomes susceptible to *Reflash* attacks.

2. Newness

   The Engine MUST be able to determine that RIMs being provisioned by the source are newer than RIMs already installed on the Device. If this determination is not possible, the Engine becomes susceptible to *Replay* attacks.

   Again, a related issue concerns protection of RIMs once they are installed on the Device. The Engine MUST be able to determine if its installed RIMs have been replaced by a set that was once valid, but older than the replaced set. If this determination is not possible, the Engine becomes susceptible to a special type of reflash attack: *Version Rollback*.

3. Currency

   The Engine MUST be able to determine that RIMs being provided by the Source are still considered to be valid by the Source. If this determination is not possible, the Engine becomes susceptible to installing RIMs that have been *Revoked* by the source. This determination MAY be implicit: if the Engine can detect that a particular source will *never* revoke any of its RIMs, it does not need to retrieve explicit information on whether a given RIM is still valid.

In the following paragraphs (Sections 5.2.1 to 5.2.5), a standardized method is defined for provisioning RIMs to meet these security properties. The RIMs are originally supplied by authorized external parties (called "RIM_Auths") and are provided for use by the RTV or any other Verification Agents of an Engine.

It is **not** REQUIRED that this method be used, or that is the only way to provide RIMs to a Device. But this method SHOULD be supported for interoperability. Further, any alternative methods MUST have security properties at least as strong as the method defined below, and MUST NOT prevent the defined method operating alongside them.

5.2.1 RIM_Auths

Each Engine has a pre-configured public key called the Root Verification Authority Identifier (RVAI). This key MUST be integrity protected using shielded storage (e.g. ROM or NV storage in the MTM, or the RVAI is signed with a key stored in the MTM). Each verification agent also has access to monotonic counters (see Section 6).

The party that owns the RVAI private key (the Root Verification Authority) is the stakeholder for the Engine. The Root Verification Authority acts as the "Root CA" of a certificate hierarchy. It can issue certificates to "primary" CAs, which can in turn issue certificates to sub CAs, and so on in a chain until **end entity** certificates are reached. The end entity keys are then used to sign *RIM_Certs*, using the TPM_RIM_Certificate structure defined in "TCG Mobile Trusted Module Specification" [5], while providing a public key signature as the proprietary authData [1].

For simplicity, the Root Verification Authority MAY directly sign the RIM_Certs that must be checked by the RTV (or other verification agents) of the Engine. However, for a general verification agent, the Root Verification Authority SHALL be able to delegate to other authorities. All delegates in the hierarchy are called RIM_Auths, and the proofs of delegation are called RIM_Auth_Certs. Delegates that are authorized directly by the Root Verification Authority are called Primary RIM_Auths; the Primary RIM_Auths MAY in turn delegate further RIM_Auths. End entity RIM_Auth_Certs correspond to end entity RIM_Auths, as these entities are not able to delegate further, just sign RIM_Certs.

The structure TPM_Verification_Key defined in "TCG Mobile Trusted Module Specification" SHOULD be used for RIM_Auth_Certs, and this structure MUST be used where the RIM_Auth_Certs need to be verified using a MTM. Again a public key signature SHALL be provided as the proprietary authData. Where a RIM_Auth is able to act as a CA, it MAY also issue X.509 certificates to other RIM_Auths.

Each RIM_Auth_Cert MUST contain at least the following information; this information is automatically contained if the structure TPM_Verification_Key is used:

- An Identifier for the Issuer Keys and Subject Keys of this Certificate
- Flags indicating whether this RIM_Auth can sign RIM_Certs directly and/or can sign delegation structures for other RIM_Auths and/or can revoke what it has signed.
- The RIM_Auth's public key
- The Signature of the private key that issued this RIM_Auth_Cert

The RIM_Auth_Cert MAY also be securely bound to any of the following lists restricting key usage (e.g. via the TPM_Verification_Key extension digest field). Such lists MAY contain wild-cards, from/to ranges etc. If any of these lists are present, then the Engine MUST respect the relevant restrictions on the RIM_Auth's authority. If any lists are not present, then the Engine MUST assume that the RIM_Auth has no restrictions in the relevant aspect(s).

- A list of platforms and Engines for which this RIM_Auth is allowed to provide RIMs.
- A list of PCRs that this RIM_Auth is allowed to instruct verification agents to extend.

[1] The algorithm, key-size and hash function must be chosen to be consistent with current TPM Main specs. For example at the start of 2006, they would be RSA-PSS with 2048 keys and SHA-1 hashing.

- A list of target objects (labels) for which this RIM_Auth is allowed to provide RIMs.

The RIM_Auth_Cert MAY also be bound to advisory information (e.g. via the TPM_Verification_Key extension digest field). An Engine MAY ignore such information. Such information could include:

- A list of target objects (labels) for which this RIM_Auth is expected to provide RIMs. If the Engine processes this list, and notices it is missing a RIM for an object on this list, it SHOULD attempt to obtain one.
- A URL for the RIM_Auth, indicating where the most recent information signed by that RIM_Auth (e.g. a full set of RIM_Certs and validity lists) MAY be obtained.

The exact format of the restriction lists, and any other advisory information, is proprietary to the RIM_Auth and corresponding verification agent(s).

5.2.2 RIM_Auth Validity Lists

If the Root Verification Authority (or other RIM_Auth acting as a CA) wishes to retract delegated authorization, then it SHOULD do so by signing a periodic *RIM_Auth_Validity_List* indicating the key identifiers of which of its delegates are still valid. Every RIM_Auth which signs Validity Lists MUST ensure that it always has signed a Validity List whose “valid from” and “valid to” fields in UTCtime format enclose the current date and time. Whether a RIM_Auth signs RIM_Auth Validity Lists or not MUST be indicated by a key-usage flag in the TPM_Verification_Key structure (see “TCG Mobile Trusted Module Specification”).

If any of the RIM_Auth CAs for an Engine signs RIM_Auth Validity Lists, then the Engine MUST be able to process them. If it is known at Engine design that none of the RIM_Auths will **ever** sign RIM_Auth Validity Lists (i.e. that no RIM_Auths will ever be revoked), then this processing functionality MAY be omitted from the Engine. However, for future proofing, it is strongly RECOMMENDED that all engines can process Validity Lists.

Start of informative comment:

This key usage mask is defined in the "TCG Mobile Trusted Module Specification” – it defines key usages which need to be recognized by verification agents, but not by the MTM.

```
#define TPM_VERIFICATION_KEY_USAGE_AGENT_MASK 0x0f00
```

End of informative comment.

```
#define TPM_VERIFICATION_KEY_USAGE_SIGN_RIM_AUTH_VALIDITY_LIST 0x0100

typedef UINT32 TPM_RIM_AUTH_VALIDITY_LIST_HANDLE;
typedef struct TPM_RIM_AUTH_VALIDITY_LIST_STRUCT {
        TPM_STRUCTURE_TAG tag;
        TPM_VERIFICATION_KEY_ID  signer_id;
        UTCTIME         validFrom;
        UTCTIME         validTo;
        BYTE validityListSize;
        [size_is(validityListSize)] TPM_VERIFICATION_KEY_ID validityListKeyIds[];
        UINT32 integrityCheckSize;
        [size_is(authSize)] BYTE integrityCheckData[];
} TPM_RIM_AUTH_VALIDITY_LIST;
typedef BYTE UTCTIME[13];
```

Parameters

Type	Name	Description
TPM_STRUCTURE_TAG	Tag	This field MUST contain the value TPM_TAG_RIM_AUTH_VALIDITY_LIST. It is used to identify the structure.
TPM_VERIFICATION_KEY_ID	signer_id	This is an arbitrary identifier whose function is to help the Engine determine which RIM_Auth signed this validity list.
UTCTIME	validFrom	This is the date and time at which the validity list becomes valid.
UTCTIME	validTo	This is the date and time at which the validity list ceases to be valid.
BYTE	validityListSize	This MUST be the number of entries in the validity list (note the maximum size of 255 entries)
TPM_VERIFICATION_KEY_ID[]	validityListKeyIds	This MUST contain a list of all key identifiers of RIM_Auth keys that are still valid delegates for the RIM_Auth which signed this validity list.
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*

Start of informative comment:

A validity list is preferred to a revocation list for at least the following reasons:

i) It is bounded in size, whereas a revocation list can grow without prior bounds;

ii) It is possible for the Engine to detect, having received the validity list, that it may be missing some valid RIM_Auth_Certs.

End of informative comment.

5.2.3 Storage and Use of RIM_Auth_Certs and RIM_Auth_Validity Lists

Each Engine of the Device MUST have available the following root authorization data in an integrity-protected form (this MAY be stored or MAY be provided externally to the Engine e.g. over a network interface):

- All the RIM_Auth_Certs associated with RIM_Certs that the Engine is currently using, and that are signed directly by the RVAI.
- If the RVAI key provides revocation information, then the **most recent** revocation information the Engine has been shown, that is signed by the RVAI private key.

In addition, each Engine of the Device MUST have available a full tree of authorization data in an integrity-protected form:

- Every RIM_Auth_Cert associated with RIM_Certs that the Engine is currently using, that presents a certificate chain up to the RVAI.
- For each RIM_Auth CA that provides revocation information, then the **most recent** revocation information the Engine has been shown. This MUST be a current Validity List if the RIM_Auth CA signs Validity Lists.

If stored in the Engine, such authorization data MUST be updateable **only** by an authorized process.

Start of informative comment:

The storage can be protected from tampering using a signing key in the MTM that is usable only by the authorized process. Or using an area of Non-Volatile storage in the MTM which can only be updated by the authorized process.

For example, the authorized process could be controlled by the RIM Conversion Agent in the Engine, which processes RIM_Auth_Certs and revocation information before generating internal RIM_Certs. (See Section 6.3.4.) The authorization data is signed using a key stored in the Engine's MTM. This signing key is bound to a set of PCRs ensuring the Engine blocks use of the key by all processes except the RIM Conversion Agent.

End of informative comment.

The above information MUST be available to the Engine for at least the following uses:

1. Authorization of new RIM_Certs.

The Engine MUST be able to determine for a given external RIM_Cert whether the certificate was signed correctly using a RIM_Auth's public key. The Engine MUST also be able to determine whether the RIM_Auth was authorized to sign that sort of RIM Certificate (i.e. has a correct delegation chain with nothing revoked on that chain, and the RIM_Auth_Cert has the correct – optional - PCR and label settings). This is discussed further in Section 6.3.4 below.

The Engine MUST ensure whenever using Validity Lists that the information contained therein is still current, according to the most reliable clock the Engine has available. (If no clock is available then the Engine MUST just use the most recent Validity List that it has). If the Engine detects that a given RIM_Auth signs revocation information, and detects that the revocation information it has is no longer current, then the Engine MUST attempt to retrieve current revocation information using an online protocol (for example, by accessing the web-site of the relevant RIM_Auth). If the Engine cannot retrieve such information it MUST abort the current operation which relies on this information.

2. Remote Attestation

The Engine MUST be able to present its current root authorization data, or full tree of authorization data, when attesting its trust state to service providers. This is discussed further in Section 6.3.4 below.

5.2.4 RIM Validity Lists

RIM Certificates signed by RIM_Auth keys are called "external" RIM_Certs (to distinguish them from "internal" RIM_Certs signed within the Engine itself – see Section 6.3.4). RIM_Auths that are able to sign RIM Certificates SHOULD be able to revoke such certificates (typically also issuing a replacement). If a RIM_Auth is able to revoke its RIM_Certs, then it SHOULD do so by signing a periodic *RIM_Validity_List* indicating the serial numbers of which of its certs are still valid. Every RIM_Auth which signs Validity Lists MUST ensure that it always has signed a Validity List whose "valid from" and "valid to" fields in UTCtime format enclose the current date and time. Whether a RIM_Auth signs RIM Validity Lists or not MUST be indicated by a key-usage flag in the TPM_Verification_Key structure (see "TCG Mobile Trusted Module Specification"). As discussed in

Section 5.2.2, if any of the RIM_Auths for an Engine signs RIM Validity Lists, then the Engine MUST be able to process them; in any case, it is strongly RECOMMENDED that all engines can process Validity Lists.

```
#define TPM_VERIFICATION_KEY_USAGE_SIGN_RIM_VALIDITY_LIST 0x0200
typedef UINT32 TPM_RIM_VALIDITY_LIST_HANDLE;
typedef struct TPM_RIM_VALIDITY_LIST_STRUCT {
        TPM_STRUCTURE_TAG tag;
        TPM_VERIFICATION_KEY_ID signer_id;
        UTCTIME         validFrom;
        UTCTIME         validTo;
        BYTE validityListSize;
        [size_is(validityListSize)] RTV_RIMCERTSERIAL validityListSerialNumbers[];
        UINT32 integrityCheckSize;
        [size_is(authSize)] BYTE integrityCheckData[];
} TPM_RIM_VALIDITY_LIST;
typedef BYTE RTV_RIMCERTSERIAL[12];
```

Parameters

Type	**Name**	**Description**
TPM_STRUCTURE_TAG	Tag	This field MUST contain the value TPM_TAG_RIM_VALIDITY_LIST. It is used to identify the structure.
TPM_VERIFICATION_KEY_ID	signer_id	This is an arbitrary identifier whose function is to help the Engine determine which RIM_Auth signed this validity list.
UTCTIME	validFrom	This is the date and time at which the validity list becomes valid.
UTCTIME	validTo	This is the date and time at which the validity list ceases to be valid.
BYTE	validityListSize	This MUST be the number of entries in the validity list (note the maximum size of 255 entries)
RTV_RIMCERTSERIAL[]	validityListSerialNumbers	This MUST contain a list of all serial numbers of RIM_Certs that are still valid, and that were signed by the RIM_Auth which signed this validity list.
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*

Description

1. The RIM_Cert SerialNumber is a sequence of 12 bytes formed as follows. The first 8 bytes in the sequence are the label field in the RIM_Cert (of type BYTE[8]). The final four bytes in the sequence are the rimVersion field in the RIM_Cert (of type UINT32), ordered according to the big-endian convention.

For example: if label = 0xFFEEDDCCBBAA9988 and rim_version = 0x00000002, then the combined twelve byte serial number = 0xFFEEDDCCBBAA998800000002

Start of informative comment:

2. A validity list is preferred to a revocation list for at least the following reasons:

i) It is bounded in size, whereas a revocation list can grow without prior bounds;

ii) It allows the RIM_Auth to distinguish a RIM_Cert which is revoked with no replacement generated, from one which is revoked where there is a valid replacement (with a higher version number)

iii) It is possible for the Engine to detect, having received the validity list, that it may be missing some valid RIM_Certs.

End of informative comment.

5.2.5 Storage and Use of RIM Validity Lists

Each Engine of the Device MUST have available the following data in an integrity-protected form:

- For each RIM_Auth_Cert that provides revocation information for its RIM_Certs, and which has signed RIM_Certs that the Engine is currently using, the **most recent** revocation information from that RIM_Auth that the Engine has been shown. (Note that if the Engine can detect that a given RIM_Auth never revokes its RIM_Certs, then nothing needs to be stored for that RIM_Auth.)

If stored in the Engine, again such integrity protected storage MUST be updateable **only** by an authorized process. The revocation information MUST be available for at least the following uses:

1. Revocation checking of new RIM_Certs.

The Engine MUST be able to determine whether a given external RIM Certificate has been revoked or not by the RIM_Auth. (If revoked, it will no longer appear on the Validity List.

2. Preventing Replay of an old Validity List

The Engine MUST be able to tell if a supplied validity list is older than the one it has currently stored. This prevents replay (and some reflash) attacks, even if the Engine does not have access to an entirely accurate clock.

The Engine MUST ensure whenever using a Validity List that the information contained therein is still current, as discussed under Section 5.2.3 above.

5.3 Measurement of Platform Behavior

5.3.1 PCR Allocation

5.3.2 Reservation of PCRs in the RTS

Start of informative comment:

It is assumed that the Device platform supports several Mobile Trusted Modules, which provide a TPM interface and / or a TSS interface. Each measurement agent is associated with a preferred MTM (the MTM of its own Engine), although it is possible that several measurement agents all extend measurements to the same MTM. This gives rise to some possible conflicts: what if several measurement agents try to write to the same PCR of the same MTM, and don't know of each others actions? In that case, when the PCR is finally read, the verification check performed by a verification agent may fail.

End of informative comment.

Each Engine supplier MUST allocate PCRs within the Engine's MTM consistently. In Section 5.2.1, the concept of a "RIM_Auth" was introduced as an entity authorized to provide RIMs to be checked by a given verification agent.

- Each measurement agent that needs to extend to a given MTM SHALL have exclusive access to at least one dedicated PCR. However, once the measurement agent has finished running, its PCR MAY then be assigned to another measurement agent which has just started up. Thus there MUST be at least as many PCRs as *concurrent* measurement agents within a given Engine.
- Where more than one RIM_Auth is entitled to provide RIMs to be verified in a given Engine, then the MTM SHALL contain at least one dedicated PCR for each RIM_Auth. Thus there MUST be at least as many PCRs as RIM_Auths for a given Engine.

Start of informative comment:

In the case of a very simple Engine with only the RTM/RTV (i.e. just one measurement agent and verification agent) and only one RIM_Auth, there could in theory be only one PCR.

The optional extension to RIM_Auth_Certs discussed in Section 5.2.1 allows the reservation mechanism to be enforced on the Device. Any RIM_Certs inconsistent with the reservations in the RIM_Auth_Certs (e.g. because they need to extend the wrong PCR, or use the wrong label) would then be rejected as invalid.

The PCR allocation could need to be larger than the reservations for a single engine would imply, for example in the case where the MTM of one Engine provides a MTM Service to another Engine. There could be a MTM as a dedicated resource (such as a physical chip) in the Device Manufacturer's Engine. Rather than requiring multiple TMT chips (one per Engine) or forcing other Engines to use software-only MTMs, the Device Manufacturer could arrange that the keys/PCRs etc. of the other Engines' MTMs are held protected in the dedicated resource MTM of its own Engine. In that case, the Device Manufacturer's MTM would need at least as many PCRs as there are combinations of (Engine, measurement agent) or (Engine, RIM_Auth). At the logical level, MTMs cannot be shared between Engines: each Engine will just perceive that it has its own MTM with just the necessary number of PCRs for that Engine.

Some special PCR reservations apply to the MTM updated by the Device Manufacturer's RTM. In some sense this is a distinguished MTM, because it is the only one that **has** to exist (the DM's RTM has to write its measurements somewhere). The PCR allocation of other stakeholder engines is currently undefined.

Observe that different engines may be allocated disjoint PCR index ranges, as this facilitates an implementation where a single underlying Mobile Trusted Module provides a MTM Service to several engines (PCR values can then be quoted by the underlying Mobile Trusted Module without any massaging of index values). These other engines could just use a translation of the DM engine allocation into different indices (e.g. add 16 for Communications Carrier's Engine, add 32 for Device Owner's Engine etc.).

End of informative comment.

The DM engine's MTM MUST have at least 16 PCR registers (this is the same number of PCR registers as defined in [2] for TPM_PERMANENT_DATA). The following PCR allocation is RECOMMENDED for the engine of the device manufacturer.

PCR 0:

- Relevant (non-identifying) characteristics of the HW platform.

PCR 1:

- Relevant (non-identifying) information pertaining to the Roots of Trust is to be measured into PCR 1

PCR 2:

- Engine-Load events for the DM Engine are to be measured into PCR 2.

PCR 3-6:

- PCRs 3-6 are to be used for DM proprietary measurements

PCR 7:

- DM Engine Operating System is to be measured into PCR 7

PCR 8-12:

- PCRs 8-12 are reserved for DM proprietary measurements

PCR 13-15:

- Unallocated at present.

Note that the number of "extend" events into PCR 1 may depend on platform implementation e.g. if the RTS/RTR/RTV/RTM are implemented using allocated or dedicated resources.

It is RECOMMENDED that PCRs 0 to 7 are verifiedPCRs, as defined in [5].

5.3.3 Concrete measurement into PCRs 0 to 2

Start of informative comment:

The state of an engine can also be described in terms of a sequence of concrete TPM_Extend or MTM_VerifyRIMCertAndExtend events recorded by the RTM, RTV, verification agents or measurement agents. This sequence is implementation and policy specific. This specification does not require a certain state machine to be followed for TPM_Extends or MTM_VerifyRIMCertAndExtends.

This specification does define a set of syntax for describing certain events during a boot-cycle of an engine. The purpose is to facilitate interoperability between software components that generate events and software components that configure policy. Both components need to have exactly the same notion of the syntax used to describe an event.

The TCG PCR mechanism only allows checking the relationship of events that have been extended into the same PCR. This specification imposes no required or mandatory ordering for events, but the assignment of PCRs tries to accommodate any needs for checking relationships of events.

End of informative comment.

This specification defines a set of syntax for describing certain events during a boot-cycle of an engine. The events SHOULD be described using the following set of parameters:

- Name: Name of the event
- Syntax: The actual byte-level representation of the event in BNF. Strings are in US ASCII. The following parameters are used in the definition of events.

  - The non-terminal *<image>* represents a SHA1 hash of the target binary in lower-case hexadecimal without any additional spacing and without a leading 0x, e.g. 0badc0de0badc0de0badc0de0badc0de0badc0de.
  - The non-terminal *<object>* is a US-ASCII name of the object/target of the event, e.g. a filename.
  - The non-terminal *<engine>* is a US-ASCII name of an engine.
  - The non-terminal *<event>* is the US-ASCII non-terminal that represents the actual event. This can be for example a representation of the *label* field in a RIM Certificate (see “TCG Mobile Trusted Module Specification”)
- Description: Description of when the event is generated.

The actual measurement is a SHA1 hash over the byte-string formatted according to the syntax of the event.

5.3.4 Diagnostic

Name RTE Diagnostic

Syntax Event ::= Diagnostic: <engine> : <object> : [<image>]

Description The RTE Diagnostic event SHOULD be generated to record diagnostic information about the trusted resources (RTS, RTR, RTE, RTM, RTV).

5.3.5 Engine Load

Name Engine Load

Syntax event ::= Engine-Load: <event> : <object> : [<image>]

Description The Engine Load event SHOULD be generated whenever new code or configuration that may affect its integrity is loaded into an engine.

5.3.6 Debug mode Entry

Name Debug Mode

Syntax event ::= Debug Mode: <engine> : [<image>]

Description The Debug Mode event SHOULD be generated whenever an engine or its RTV enters debug mode. The <image> non-terminal may in this case be a random-value.

5.4 Transitive chain of trust for Measurement and Verification agents.

Before completion of execution, the RTM MUST measure (and the RTV MUST verify) the executable load image of at least one other measurement agent (and at least one associated verification agent). The new measurement/verification agent(s) are then given control and continue with measurement and verification.

Measurement Agents which execute following the RTM can then measure further measurement agents, and other functions giving rise to a dependency tree. Moving down the tree there is likely an increase in complexity of code, but a drop in privileges as more parts of the OS get loaded (e.g. a drop to kernel mode then user mode, then functions in Java virtual machines).

Any function which is required to be executed as part of the tree, and which is required (by a RIM_Auth) to be verified before it is executed, is referred to as a *mandatory function*. Typically, measurement and verification agents other than the RTM/RTV are themselves mandatory functions, but there will usually be others.

There are some restrictions on the structure of the dependency tree, as follows:

- Any measurement agent which is a parent node to a mandatory function MUST also be, or be associated with, a verification agent.
- A measurement or verification agent SHALL NOT be a parent to a mandatory function (such as a MTM or TSS) when the agent already needs to use that function (to store PCRs etc.)
- Leaf measurement (and/or) verification agents never hand-over control to other measurement or verification agents. Leaf agents MAY therefore be candidates for run-time measurement (and verification) agents i.e. entities which continue to measure after boot.

5.5 Measurement agent Operation at Higher Layer

Each measurement agent SHALL perform measurement functions in the same way as the RTM, as defined in Sections 4.1.3 and 4.2. In particular:

The measurement agent's code/configuration data SHALL implicitly or explicitly point at a list of Target Objects of measurement. Once each measurement is made, the measurement agent MUST either itself attempt a PCR extend in a MTM with which the measurement agent can communicate or, where verification is required, MUST pass the measurement to a corresponding verification agent.

Each associated verification agent - if present - SHALL perform verification functions in the same way as the RTV. Requirements 1-3 for the RTV, as listed in Section 5.1.2, SHALL apply.

Where verification of measurements is required, the associated verification agent MUST be able to retrieve corresponding Reference Integrity Metrics (RIMs). The verification agent MUST verify each measurement against a RIM, and if successfully verified, MUST attempt a PCR extend or verified extend in the MTM. If the verification agent detects a verification failure, or the MTM reports a failed verified extend, then this failure MUST trigger a Reactive Response (see Section 7).

All higher layer extend or verified extend actions SHOULD also be recorded in a PCR Event Log (e.g. this will happen whenever the measurement agent communicates through a Mobile TSS).

6. Lifecycle Management

6.1 Initialization

6.1.1 Generation of Storage Root Key

The Storage Root Key (SRK) which is used as the foundation of the Root of Trust for Storage (RTS) component must be generated using a cryptographically strong process that meets or exceeds the requirements for the strength and equivalent security of the RTS itself [refer to TCG 1.0 Architecture Overview section 4.3.1.7]. This key is a critical part of the on-platform storage function that needs to be robust and long lived. This key should be an appropriate size to support the algorithm used with in the cryptographically secure storage function.

The SRK MAY be generated on the platform [refer to TCG 1.0 Architecture Overview section 4.3.1.5 for Random Number Generator guidelines] [refer to FIPS 140-2 for general guidance on RNG and key generation]. For the DM engine, the SRK MAY be generated externally and inserted into the engine during manufacture time based on limitations of the engine performance. If the AIK is pre-generated and installed during manufacture time the SRK MUST be generated and installed at the same time. (Because the AIK is stored under the SRK.)

Start of informative comment:

The RTS block must provide the appropriate controls for storing this key so that once it is generated and loaded it can not be modified by subsequent use of the generation request. The RTS must provide integrity protection for the storage of the SRK that is capable of detecting when a key has been altered in any way. The SRK may be implemented as either a symmetric or an asymmetric key for the RTS [note: the current TPM 1.2 specification only provides RSA as an option. The concept of a symmetric SRK is under consideration]. The storage element for the SRK must provide protection so that the private key or the secret key, depending on the implementation type, is protected from being made visible to any external entity. If this key is allowed to be replaced then all stored data becomes irretrievable and may result in a non-recoverable state. This key may also be generated external to the platform by a key generation tool and installed at the time of device or platform manufacturing. When this external method is used it should have the generation method verified to provide strong keys, there should be no linkage between the key that was loaded and the device or platform identity or other generated keys, and it should be loaded in a reasonably secure environment. In this context a secure environment is intended to offer physical protection of the key generation tools, physical access control to the area the generation and loading is performed within, appropriate access control to systems that control the process for operators and protection of the network environment to protect sniffing and subversion from a network based attack. Industry best practices should be employed wherever feasible to accomplish this protection. The level of security established for external key generation and handling should be at the same level of protection that is required to protect the data.

End of informative comment.

6.1.1.1 Key Guidelines

The RTS MUST provide integrity protection for the storage of the SRK that is capable of detecting if either the private key or the public keys have been altered in any way. TCG specifications require that the SRK storage must provide protection so that the private key is also protected from being made visible to any external entity. This key SHOULD be at least 2048 bits for an RSA key or an equivalent Elliptic Curve key of size 224 with an appropriate curve, or another key type permitted by the TPM specification for use as an SRK.

6.1.2 Creation of Endorsement Key

Start of informative comment:

The Endorsement Key (EK) is an optional element for the Mobile environment. If it is used, then it must be created and bound to the device. This key must not be migratable. The EK for the DM's engine (and if necessary, other mandatory domain engines) will be generated by the device manufacturer and installed into the RTS. There are two approaches to key generation and insertion; 1) generate keys off-chip and insert as part of the platform creation, 2) generate keys on-chip using available commands. If the off-chip generation and installation method is used by the device manufacture, then the approach must provide a way to block removing this key pair at a later time or replacing it with a new key pair once it is fielded. For the second option, the generation command must be disabled once the capabilities are ready for shipment. Once the EK key pair is generated the device manufacturer needs to build the EK credential and make it available once the device is ready to be shipped.

This follows the definition of the TCG IWG for OEM EK generation when the MTM is integrated into the processor and the device manufacture at this point is equivalent to the OEM.

If the engine includes the use of the EK, there are several optional Mobile Trusted Module commands that will need to be supported (e.g. they become mandatory). These are defined in "TCG Mobile Trusted Module Specification".

End of informative comment.

6.1.3 Creation of Identity Keys

The Identity keys, or AIKs, and their associated certificates are used by the platform to authenticate an Engine and to attest to the state of an engine. The AIK and certificates can be generated and installed using several methods that may or may not depend on the presence of an EK for the platform.

6.1.3.1 AIK generation with an EK

When an engine has an EK available it can generate the AIK key pair and communicate with a privacy CA using the EK to provide the AIK certificates. The privacy CA will validate that the engine's certificates describe a genuine engine and will generate the AIK certificate. The AIK certificate will then be sent to the platform for storage in the RTS. The Privacy CA is trusted to correctly evaluate Integrity Assertions and the Owner-specific policies as input into the process of issuing AIK-Credentials. The Privacy-CA in practice can be a local AIK-Credential issuer (e.g. Enterprise IT Administrator) or it can be a public certificate authority in the sense of a Classic CA. For the mobile phone platform, the engine stakeholder may be the only entity that generates the AIK key pairs. For additional information see the definition of TPM_MakeIdentity and TPM_ActivateIdentity in the main specification [3].

6.1.3.2 AIK generation without an EK

The main purpose of the EK is to enroll identities for the MTM. However, an engine may not include the EK and the identity of the MTM functions will be assumed to be assigned by the device manufacture. If this option is used, the manufacturer MUST generate and install the AIK and associated certificates during the manufacturing process of the engine and bind them to the device. There is not an option for self generation of the AIK credentials, so they must be generated off-chip and then installed into the engine.

6.2 Taking Ownership

Start of informative comment:

The Owner is the actor that controls access by platform users to the cryptographic functionality of a MTM. Users cannot access the MTM cryptographic functions without permission from the Owner.

The MTM flags “disable” and “ownership” determine a MTM ability to accept an Owner. The flags can be set TRUE or FALSE via TCG’s Physical Presence interface, and can be used to deny access by software to the MTM capability that installs an Owner. If disable==FALSE and ownership==TRUE, TPM_TakeOwnership can be executed. TPM_TakeOwnership installs the Owner’s chosen authorization value in the MTM, and creates the SRK and tpmProof. If the flag “deactivated”==FALSE, the cryptographic functions in the MTM may then be used.

The MTM flag “deactivated” may be used to prevent the (obscure) attack where a MTM is readied for TPM_TakeOwnership but a remote rogue manages to take ownership of a platform just before the genuine owner, and immediately has use of the MTM’s facilities. To defeat this attack, a genuine owner should set disable==FALSE, ownership==TRUE, deactivated==TRUE, execute TPM_TakeOwnership, and then set deactivated==FALSE after verifying that the genuine owner is the actual MTM owner.

Note that the “disable” and “deactivated” flags also affect the MTM after an Owner has been installed, but those properties are not described here.

(See also the main specification [3] – TPM take ownership)

End of informative comment:

6.2.1 Remote or Local Owner

Start of informative comment:

Any engine within this specification may be designed to have a remote owner, who does not have physical possession of the mobile platform. A local owner could, of course, also use an engine designed for a remote owner, although in this case all of the management and control functions would need to operate through the remote owner interfaces. If the engine is not designed for a remote owner, the engine must have either a local owner, or else have no owner but allow a local User to take ownership. Such a local owner will have possession of the platform, and so can (for example) use assertions of Physical Presence.

End of informative comment:

The “TCG Mobile Trusted Module Specification” [5] defines two types of MTM to meet the requirements of these two types of owner. An engine designed to have a remote owner (e.g. the Device Manufacturer’s engine) MUST support the type of MTM defined as a “MRTM”. An engine designed to have or permit a local owner MUST support the type of MTM defined as a “MLTM”.

In the case of a remotely owned engine, a general model is that the engine’s MRTM is already enabled and activated, and already has an owner set when the User takes possession of the device. This MUST be true of the Device Manufacturer’s Engine, for example. However, a remote owner MAY be able to take ownership at a later date if not already set through TPM_TakeOwnership (which is OPTIONAL within the MRTM command set). The remote owner MAY also be able to relinquish ownership through TPM_OwnerClear (which is also OPTIONAL for a MRTM). However, in all cases, the remote owner MUST be protected from a User attempting to remove the remote owners ownership of the engine, or attempting to disable or de-activate the engine’s MRTM. In the case that the user is the Device Owner and the engine is on a DO controlled list then the user can always remove the engine entirely. A remotely owned engine MUST therefore support secure boot, to ensure that the engine loads the way the remote owner is expecting. This is the fundamental difference between the “Mobile Remote owner Trusted Module” (which supports secure boot) and the “Mobile Local owner Trusted Module” (which may not use or support a secure boot operation).

In the case of a locally owned engine, a general model is that the engine's MLTM does not yet have an owner set when the User takes possession of the device. It MUST be possible to establish the owner through the TCG's Physical Presence Interface e.g. the DM Engine should allow the User to assert a physical presence, and then present that status to a User engine. The User engine's MLTM MAY already be enabled and activated, or the User MAY need to set those flags before taking ownership. The User engine's MLTM MUST allow the local User to change the flags using assertions of Physical Presence. If there is already an owner set in the User's engine's MLTM (e.g. someone previously used the device), then the local User MUST be able to remove that owner using an assertion of Physical Presence (i.e. TPM_ForceClear, which MUST be supported by a MLTM).

6.2.2 Local Owner Control of Secure Boot

An engine with a local owner (i.e. a User engine) MAY also provide secure boot functionality. If so, the engine's MLTM MUST support the local verification commands defined in Section 7 of "TCG Mobile Trusted Module Specification". Also, the engine MUST allow the local owner to act as the stakeholder in control of that secure boot functionality.

In particular:

- The local owner MUST be able to set the RVAI public key for the engine (see Section 6.3.1), that is to be used by the RTV and other verification agents of that engine.
- The local owner MUST be able to control which PCRs are set as verifiedPCRs (see Section 6.3.1.2) in that engine's MLTM .
- The local owner MUST have exclusive control over which RIM_Auths' TPM_Verification_Keys can be loaded into the engine's MLTM, and hence over which external RIM_Certs are accepted by the engine's MLTM. In particular, the MTM's integrityCheckRootData (see Section 6.3.1.1) MUST be set to NULL and the flag loadVerificationRootKeyEnable MUST be set to FALSE.
- The local owner MUST have exclusive control over which internal RIM_Certs are generated by the engine's MLTM (see Section 6.3.4.1), and over the counterRIMProtect used to revoke such internal RIM_Certs.

Where such control affects a MLTM, it is fully specified within "TCG Mobile Trusted Module Specification". Note for instance that where the ownerAuth data is set, it is used to gate control over the local verification commands in preference to verificationAuth data.

A secure boot User engine MAY have no owner set yet: in particular this will be the case on a User engine's very first boot. In that case, there will be no valid internal RIM_Certs, and so the engine MUST follow a pristine boot process, as defined in Section 6.3.3.1. As this requires an RVAI public key to be set (and possibly IntegrityCheckRootData to be set as well), any "owner-less" secure boot SHOULD use a default combination of RVAI/IntegrityCheckRootData/RIM_Auths/RIM_Certs provided by the Device Manufacturer for that engine. This MUST provide a limited secure boot, just sufficient to enable the local User to securely take ownership, and establish their own secure boot control (including setting own RVAI key etc.)

If the local owner chooses to relinquish ownership, it is RECOMMENDED that the default boot settings are restored, enabling a local User to take ownership again if so desired (and if the MLTM flags permit).

6.3 Lifecycle of a Secure Boot Engine

Start of informative comment:

This section outlines the basic view of the major processes during the lifecycle of a secure boot engine from a security perspective. Each section will provide an outline of the basic functions that are used at that point in time and can provide a high level view of how the major elements come together to form a trusted environment.

Engines which do not support an RTV are **not** REQUIRED to be compliant with this section.

End of informative comment.

6.3.1 Generation of the Root Verification Authority Identifier

Start of informative comment:

The Root Verification Authority Identifier (RVAI) is the public key that will be used to verify the RIM_Auths (the source providing and authorizing the external RIM_Certs) for each engine. This public key should be integrity protected and either the key or integrity protection information must be stored in tamper resistant memory. (see Section 5.2.1 for requirements)

End of informative comment.

This key will be generated by the engine's stakeholder. The RVAI for the device manufacturer's engine will be installed on the device during the platform manufacturing process, and (if known) the RVAI for other secure boot engines may also be installed at manufacture. Once this key is installed, at any time if the RVAI (or integrity protection value) is found to have been tampered, the engine MUST go to a "FAILED" state and block the platform from entering a "SUCCESS" state (see Section 7.1.2).

Start of informative comment:

Multiple methods may be used to install and protect the RVAI on the platform. The public key may be directly incorporated as part of the ROM image on the device and used directly by the RTV to verify RIM_Auth_Certs. It may also be programmed into One Time Programmable (OTP) storage either on the processor, or on a device that is cryptographically bound to that processor so that the memory device can not be replaced by exchanging an inexpensive component. In this case the RTE would have to verify the cryptographic binding between the processor and the storage device, and then the RTV would load the RVAI from the memory device and then use it to verify the RIM_Auth_Certs. It may also be possible to store a hash value of the public key in either ROM or in OTP and the key provided in general FLASH memory. For this approach the RTV may load the key from memory and validate the value against the hash before it is used. In the case where the ROM holds the key or the hash, the device manufacturer would provide the value to the processor vendor as part of the ROM mask and would receive the processors with this key or hash preprogrammed. For the cases where the key or a hash is programmed into OTP this process should be done early in the product manufacture cycle to reduce the risk of rogue RVAIs from being introduced.

End of informative comment.

Special considerations apply either 1) where the RVAI for an engine is not known at manufacture, or 2) where the engine is only created on the platform after manufacture.

In case 1), the Device Manufacturer MAY choose to initialize each engine with a copy of the DM's own RVAI, but assign each copy a distinct key identifier ("my_id"). The DM can then hand over control of an engine after manufacture by signing a special RIM_Auth_Cert (i.e. a TPM_Verification_Key structure which gives the delegate key exactly the same key usage as the DM's RVAI key had originally, in particular the ability to sign further delegates). The Device Manufacturer can control which engine is handed over by using a "parent_id" in the RIM_Auth_Cert that matches the "my_id" in the engine concerned.

In case 2), the Device Manufacturer MAY provide a means to specify the RVAI key when an engine is created. Typically, the Device Manufacturer would define a proprietary command to create a new engine, and the RVAI key could be a parameter to that command. Alternatively, insertion of an RVAI key MAY be postponed until someone has taken full ownership of the new engine, in which case the engine would provide a proprietary owner authorized command to set the RVAI.

6.3.1.1 Setting of IntegrityCheckRootData

Except in the case of a User engine, it is RECOMMENDED that either the RVAI key itself, or a hash of the RVAI key, is stored in the Mobile Trusted Module by use of the field integrityCheckRootData defined in the MTM_ Permanent_Data (see "TCG Mobile Trusted Module Specification".) This integrityCheckRootData MAY be set at manufacture of the platform, or MAY be set at Engine creation, or MAY be set when taking ownership of an Engine (see above). It is **not** REQUIRED that any of these three options is used, but one of the three options SHOULD be used if integrityCheckRootData is set. In case integrityCheckRootData is set, the flag loadVerificationRootKeyEnabled SHOULD be permanently set to FALSE, as the MTM would never be required to load a verification key without integrity checks or authorization.

If no such record or integrity check of the RVAI is held in the MTM, then the MTM is dependent on the RTV of its Engine to load in the correct RVAI key at the start of boot. In such cases, the flag loadVerificationRootKeyEnabled SHOULD be initially set to TRUE on each power-up cycle, to enable the RTV to load in the RVAI key without integrity checks. The RTV MUST set the flag to FALSE (using MTM_LoadVerificationRootKeyDisable) before handing over execution control.

6.3.1.2 Setting of VerifiedPCRs

If the loadVerificationRootKeyEnable flag is set permanently to FALSE, then the verifiedPCRs selection in the MTM Permanent Data MUST be set permanently at manufacture, for remotely owned engines, or MUST be set/reset under owner authorization, for locally owned engines (using MTM_SetVerifiedPCRSelection). If the flag is initially set to TRUE on each power-up cycle, then the RTV MUST ensure correct settings for the verifiedPCR selection at each power-up (using MTM_SetVerifiedPCRSelection).

6.3.2 Monotonic Counters

To protect against *Version Rollback* attacks, and in general the installation of *Revoked* RIMs, three sorts of monotonic counter are defined within this specification.

1. Each engine which supports secure boot MUST utilize its CounterBootstrap counter to verify external RIM_Certs during "pristine" boot. This counter SHOULD reside on the main processor (or else be crypto-graphically bound to the main processor), SHOULD be nonvolatile and SHOULD resist tampering to reset it to a previous state. See MTM specification [5] section 6.1.4 for the reasoning behind the requirements of counter properties.

This counter acts as a method to prevent roll back to old, revoked, memory images via physical attacks which completely erase the flash image of the device and attempt to reload it from scratch (e.g. as if the device had just been manufactured).

A value of this counter MUST be installed in the external version of each RIM_Cert used to verify a pristine boot, and MUST be checked during pristine boot to ensure that the RIM_Cert being used is not a revoked version. As the same external RIM_Certs are used across multiple devices, the value of this counter needs to be globally synchronized for a class of devices.

This value would be known to each RIM_Auth and would only be incremented if the security position of the platform is compromised by configurations authorized by previous RIMs. This counter does not need to be incremented for each release of a new RIM, only for those cases where new sets of RIMs must be released to maintain the security posture for the platform. This counter value would be shared across an entire platform configuration to provide reasonable maintenance. Based on the use scenario, this value is not anticipated to be overly large.

2. Each Engine MUST utilize a dedicated CounterRIMProtect to verify internal RIM_Certs during “standard” boot. This counter SHOULD reside on the main processor (or else be crypto-graphically bound to the main processor), SHOULD be nonvolatile and SHOULD resist tampering to reset it to a previous state.

This counter acts as a method to prevent roll back to old, revoked, memory images via attacks which attempt to restore the image of the target Engine to a previous copy of the image for that target Engine.

A value of the counter MUST be installed in the internal version of each RIM_Cert used to verify a normal boot, and MUST be checked during normal boot to ensure that the RIM_Cert being used is not a revoked version. As internal RIM_Certs are unique to a device, the counter value here will also be unique per Engine instance.

3. Each Engine SHALL support an additional monotonic counter for all other purposes. This counter SHOULD be protected on (or bound to) the main processor, but MAY be used to support the off-chip storage images of other counter values.

This counter will be used to provide protection of stored images that in turn may contain data for other counters. This is a known technique for creating an indefinite family of counters out of a single protected counter.

6.3.3 Boot Processes

6.3.3.1 Pristine Boot (factory install)

Start of informative comment:

Pristine boot for this specification is defined as an engine that boots where there is currently no local generated information available. This condition will be experienced during the manufacture process, but may also be experienced if a significant upgrade causes all of the internal RIM_Certs to be removed or corrupted.

The first step for the platform is the initial build in the factory. The normal case for this set of operations is that the platform manufacturer is building the device and needs to load all of the initial parts of the system to get to an operational device. The rogue case for this scenario is attackers that have installed a fresh FLASH device as one attack vector to the information or services available to the platform. If the RTS is built using allocated resources on the processor, then the SRK is assumed to be installed by the processor provider, or is installed into the processor as the first step in the platform build process. If the RTS is provided by a separate device, then the SRK will reside on that device and that device should be bound to the processor at this time so that the device providing the RTS function can not be replaced as an attack to circumvent the security controls.

The DM's RVAI or the data element providing the integrity protection should be installed into the processor as noted above. If there will be an EK used, it needs to be installed by the processor supplier or generated and installed at this point in the process. The AIK and the AIK credentials may need to be generated or installed into the RTS at this time. The manufacturer may also at this point personalize the MTM with an RVAI key, so that this key does not need to be explicitly loaded during the boot process (see below). The extended software image now needs to be loaded into the platform memory to establish a working system. This extended image should include software, configuration information, and all external RIM_Certs that are required to validate the software that is being installed. The public RVAI key will be used to verify a hierarchy of keys (belonging to RIM_Auth_entities) used to verify RIM certificates. Each engine may be initialized with the RVAI of the Device Manufacturer, which later hands over control of the individual engines to their respective RIM_Auths.

When the platform comes out of reset, the DM engine's RTE (if present) should perform a self consistency check and then build all the Roots of Trust and pass control to the RTM/RTV. The RTM should detect that there is software on the platform, and the RTV should check for a valid set of internal RIM_Certs to be used during the verified boot process. The mechanism of the status check will be left to the platform designer.

End of informative comment.

If there are no valid applicable internal RIM_Certs, then the DM engine MUST attempt a pristine boot. The platform will need to know where to find the store of external RIM_Certs to start the construction process. This first level of construction is an operation of the RTV. The RTV will take each external RIM_Cert required to verify next level of operation and verify that it is signed by a valid RIM_Auth_Cert that is authorized to issue RIMs for this platform. If the RVAI is not already loaded into the MTM, then it SHOULD be loaded by the RTV using the MTM_LoadVerificationKey command with the parentKey field null. For the DM engine, this will be the only verification root key that SHOULD be used by the MTM, and the MTM_LoadVerificationRootKeyDisable command SHOULD be issued to prevent any further root keys being loaded.

While it is possible that each RIM_Cert may be signed directly by the RVAI for limited performance platforms, standard best practices recommend that the RVAI only be used to sign authorization certificates. Starting from the RVAI key, the trust chain for each RIM_Auth_Cert required to reach the next level of operation will be verified by the RTV using the MTM_LoadVerificationKey command. Once each of the RIM_Auth_Certs has been verified, then each of the external RIM_Certs can be verified by using the MTM_VerifyRIMCert command with the external RIM_Cert in the rimCert field and the RIM_Auth_Cert in the rimKey field. This process SHOULD be used to validate the structure and trust chain of the external RIM_Certs.

During the above validation process, the RTV SHOULD read the value of the counterBootstrap counter using the TPM_GetCapability command (see [5], Section 8.1). If any RIM_Cert or RIM_Auth_Cert indicates that the counterBootstrap counter has been increased, then the appropriately authorized certificate that will authorize the incrementing of the counter SHOULD be identified and the MTM_IncrementBootstrapCounter SHOULD be called. If the counter values in the certificates are the same value as the current counterBootstrap value, then the construction task may continue with no action required.

Start of informative comment:

The RTM can then measure the boot loading process and the RTV retrieves the external RIM_Certs that describe the boot loading process. The RTV verifies that the measurement report by the RTM matches the measurement in the RIM_Cert describing the boot loading process. The RTV uses MTM_LoadVerificationKey to load any RIM keys needed from the hierarchy that are not already included in the MTM. The RTV then calls the RTS using MTM_VerifyRIMCertAndExtend to cause the TRS to verify the certificate using the loaded key, and then extend the measurement value stated in the certificate into the PCR stated in the certificate. When the RTV receives a SUCCESS indication, it executes the boot loader process and passes control to that verified software.

End of informative comment.

If any errors are encountered during the pristine boot process, then the Engine MUST go to a "FAILED" state (see Section 7.1.2). This pristine boot process MUST be completed and in a "SUCCESS" state before any "RIM Conversion Agent" (at least one per Engine) can run to do a certificate conversion (see Section 6.3.4.1), creating internal RIM_Certs ready for the next (standard) boot. During this conversion process, ownerAuth data or verificationAuth data MUST be entered into the engine, or otherwise made available to the engine. In the case where verificationAuth data is stored on the platform, it MUST therefore be sealed to the expected PCR state that will exist after the pristine boot is successfully completed, but before the conversion process runs.

A similar process MAY apply to other engines on the platform during pristine boot i.e. an engine is started, discovers it has no internal RIM_Certs and attempts to boot using external RIM_Certs. Alternatively, the pristine boot process MAY be designed so that the DM Engine completes its own building, and then creates the other engines fully built, but in a simplified state. The other engines MAY **not** need their own pristine boot sequences.

6.3.3.2 Standard Boot

Start of informative comment:

From the perspective of the platform user, the boot process is defined as the time from applying power until the user is able to perform interesting functions. The primary boot will be defined from the time that power is applied until the fundamental capabilities are in place. These capabilities include the roots of trust and system mechanisms that provide resource separation and management. This primary boot operation is defined by the device manufacturer and includes all of the mandatory security functions for at least the DM engine. The secondary phase begins once the primary boot has completed. This phase includes the operating system configuration, user configuration and user applications that will be installed.

The process of verifying the software on any Engine is performed by verification agents (as discussed in Section 4). Each verification agent has access to a list of internal RIM_Certs.

During standard boot the verification agent does not necessarily have to verify externally created signatures or do certificate parsing itself. All that is required is that there is a process (supported by the MTM) whereby the verification agent can load an internal RIM_Cert, and check the internal cert's validity (i.e. that it is correctly signed by the Engine's internalVerificationKey, and that the counter value is not less than the current CounterRIMPprotect value). Provided the internal RIM_Cert is accepted as valid, then its contents can be used for verifying a current measurement. This verification process is provided by the MTM_VerifyRIMCert or MTM_VerifyRIMCertAndExtend commands.

End of informative comment.

During standard boot, each measurement agent (starting from the RTM) MUST perform its Target Measurements in order of execution, as defined by its measurement configuration data, and pass these to a verification agent to check the results against RIMs. The verification agent SHALL identify the correct RIM_Cert for this measurement by using the label and PCR index values passed by the measurement agent.

The verification agent SHALL check that the measured value provided by the measurement agent (and the PCR index to extend) matches the expected value in the RIM_Cert. If the values match then the processing will continue by the verification agent using a MTM_VerifyRIMCertAndExtend (or simple TPM_Extend) to extend the measurement into the target PCR. Otherwise if the measurement does not match the RIM_Cert value the engine SHALL transition to a "FAILED" state.

The verification agent MUST also check for the condition where a target object which **must** be verified (according to measurement configuration data) nevertheless has no RIM available. Then in this case the Engine MUST also transition to a "FAILED" state.

The verification agent MUST also ensure that before extending, the PCRs in the MLTM match the prerequisite state (see section 5.2 in the MTM spec) defined by the RIM_Cert. This can be done in several ways:

- The check is done by the MTM, when the verification agent asks the MTM to perform a MTM_VerifyRIMCertAndExtend command.

- The verification agent asks the MTM to perform a TPM_Quote of the relevant PCRs and checks the quote using the public half of the MTM's AIK.
- The MTM has some key sealed to the expected PCR values, and the verification agent attempts to use the key.

If there is a mis-match, the Engine MUST transition to a "FAILED" state.

Start of informative comment:

The check being performed by the MTM when the verification agent requests a MTM_VerifyRIMCertAndExtend command is the preferred approach, as it is simpler, and any verification failure is immediately detectable by the MTM, leading to suitable response mechanisms by the MTM (e.g. quarantining of sensitive data, shut-down, alarms detectable by the rest of the Engine). Note that all the above methods require the MTM to be Enabled and Active, and to have an Owner installed.

Note that a prerequisite states maybe inserted into a RIM Cert to enforce a specific sequence of boot.

End of informative comment.

6.3.4 Updates and Revocations

6.3.4.1 External RIM_Certs and Internal RIM_Certs

Start of informative comment:

The full set of external RIM_Certs, RIM Validity Lists, RIM_Auth_Certs and RIM_Auth_Cert revocation information (RIM_Auth Validity Lists etc.) defines a complex privilege structure.

It is not required that each verification agent (especially the RTV) is able to process this whole structure during each boot and hence determine what is really a valid RIM.

This problem is addressed by using a special RIM Conversion Agent to process all of the external RIM_Certs and map from External RIM_Certs to Internal RIM_Certs. These Internal RIM_Certs can then be more easily handled by the RTV and other verification agents.

End of informative comment.

The rules for when to use Internal RIM_Certs and when to use External RIM_Certs are summarized as follows:

1. For pristine boot, an Engine MUST use external RIM_Certs directly. The RIM_Certs themselves MAY be revoked using the counterBootstrap monotonic counter.

2. For standard boot, the Engine SHOULD use internal RIM_Certs previously converted from external RIM_Certs. It MAY use internal RIM_Certs previously converted from legacy Device Management protocols that are not RIM aware. These internal RIM_Certs MAY be revoked using the counterRIMProtect monotonic counter. If the counterRIMProtect is incremented by the platform that any internal RIM_Certs that have a lower value should be considered revoked.

Any trusted RIM Conversion Agent MUST check that both the external RIM_Certs themselves, and any RIM_Auths in chains used to sign the external RIM_Certs, have NOT been revoked before doing the conversion. This check SHOULD be done by accessing up to date Validity Lists (but it MAY use any equivalent revocation checking mechanism for legacy DM, or MAY be implicit.) The RIM Conversion Agent MUST check that each RIM_Auth is authorized to sign each external RIM_Cert, by checking all the optional constraints on Device and Engine identifiers, PCRs, timlabels etc. in the RIM_Auth certificates.

Any trusted RIM Conversion Agent MUST be authenticated to the MTM using verificationAuth or ownerAuth data (as defined in "TCG Mobile Trusted Module Specification"). The verificationAuth data SHOULD be static and assigned at manufacture for remotely owned Engines. The ownerAuth data MUST be used for locally owned engines. The verificationAuth (or ownerAuth) is necessary to run a RIM_Cert conversion command

(MTM_InstallRIM). In most implementations, the trusted RIM Conversion Agent will reside on the device and the verificationAuth MUST be sealed to a PCR state which ensures that only a trusted conversion agent can access that data. Where ownerAuth data is used instead of sealed verificationAuth data, then the operation to create internal RIM_Certs MUST require owner input and control.

Start of informative comment:

The trusted nature of the RIM Conversion Agent will often require that function to exist on the platform and be validated by the RTV before allowing it to convert the RIM_Certs from external to internal format. There may be some special use cases that will cause this agent to exist external to the platform. For these unique cases the process should be designed and implemented to use a trusted channel [OS-AP] between the MTM and the external agent.

End of informative comment.

The RIM Conversion Agent **re-validates the authorization for** all valid external RIM_Certs by calling the MTM_InstallRIM command. This creates a set of *internal* RIM_Certs. The MTM_InstallRIM command inserts the Engine-specific monotonic counter value Counter_RIMProtect into all the internal RIM_Certs (and this counter will be accessible to the MTM and verification agents during subsequent boots).

These re-validated internal RIM_Certs can then be sorted into convenient lists of RIMs addressed to each verification agent in the Engine, and matched against Target Objects of measurement addressed to each measurement agent in the Engine. The label fields in the valid RIM_Certs help here.

3. For standard boot, the Engine MAY use external RIM_Certs directly.

However, any verification agent that is trusted to call MTM_VerifiedRIMCertAndExtend with an external RIM_Cert MUST check that both the external RIM_Cert, and any RIM_Auth in the chain used to sign the external RIM_Cert, is fully authorized and has NOT been revoked before using the cert. This check SHOULD be done by accessing up to date Validity Lists (but it MAY use any equivalent revocation checking mechanism, or MAY be implicit).

The verification agent MUST also check that each RIM_Auth is authorized to sign each external RIM_Cert, e.g. by checking all the optional constraints on Device and Engine identifiers, PCRs, labels etc. in the RIM_Auth certificates.

6.3.4.2 RIM Updates

Once a platform is fielded there are often requirements to authorize updates to the software elements that were included in the original release. The engine MUST be in a “SUCCESS” state before this update process begins. If the engine cannot get to a “SUCCESS” state to perform this update, then the platform SHOULD be forced into a mode where the FLASH can be reloaded and the procedure for pristine boot can be followed.

The following process describes an update to a single software object, and single RIM, but updates to multiple software items and RIMs at once can be handled in a very similar way.

Each RIM_Auth whose target object needs updating SHALL prepare the software update package and include a new RIM_Cert that defines that package. If the RIM_Auth signs Validity Lists, it SHALL sign a new Validity List containing the new RIM_Cert (and possibly omitting previous versions of that RIM_Cert).

This package is then passed to the Engine via any suitable update protocol, as described in Section 6.3.4.6. A RIM Conversion Agent in the Engine SHALL receive the package and verify/validate the corresponding external RIM_Cert.

The RIM Conversion Agent SHALL receive the new RIM_Cert and verify the RIM_Auth using a certificate chain (if applicable) back to the RVAI key. The RIM Conversion agent MUST determine what is a valid RIM by using a full path verification process (including checking for revocation status, checking all the optional constraints on Device and Engine identifiers, PCRs, labels etc. in the RIM_Auth certificates).

Once the RIM_Auth has been verified, the information on the engine and the label in the RIM_Cert can be used to identify which RIM is being updated.

The RIM Conversion Agent MUST validate that the external RIM_Cert counter value (if present) is the same or greater than the CounterBootstrap value associated with the platform, and if not, it MUST reject this new RIM_Cert. If the certificate count is greater than the CounterBootstrap version associated with the platform then the CounterBootstrap version MUST be updated to match the new value in the certificate. Because this counter is only used to validate incoming external RIM_Certs none of the existing internal RIM_Certs need to be modified based on this new input.

If verified and validated, the RIM Conversion Agent SHOULD create a new Internal RIM_Cert using MTM_InstallRIM, and pass on the updated software for installation (and use during next boot). The old Internal RIM_Cert that corresponded to the update request SHOULD be removed from the Engine's internal store of RIM_Certs. Any new Internal RIM_Cert MUST be added to the Engine's internal store of RIM_Certs.

6.3.4.3 Conversion of Legacy Device Management to Internal RIM_Certs

Start of informative comment:

A number of existing protocols for managing Mobile Devices have been defined by OMA, 3GPP and other bodies. In many cases, these protocols will be providing or updating code or other objects on the Device which will impact on an Engine's security state.

It may be assumed by the Device that where such sensitive objects are updated through "legacy" (i.e. non TCG-aware protocols) then any authorized update is implicitly providing a new target to measure and a new expected value of the measurement (i.e. the functional equivalent of the RIM).

End of informative comment.

A RIM ConversionAgent MAY also be called upon during such legacy updates (as well as upon updates of external RIM_Certs), and be asked to create (or add to) the set of valid internal RIM_Certs.

This technique provides a smooth way of migrating from legacy Device Management protocols towards protocols which are aware of the TCG specs and which can thus explicitly include external RIM_Certs.

Note that where legacy Device Management is used, it MUST have the same security properties as were defined for the framework of external RIM_Certs (Source, Newness, Currency). Conventional code-signing (X.509) certificates with an effective revocation framework (e.g. OCSP) would be acceptable here.

6.3.4.4 Post factory RIM installation

The operation for adding a new RIM_Cert once a platform is fielded is exactly as for the update path, defined in 6.3.4.2, except that no early version needs to be removed.

As new software is required to be measured and verified, the Engine may find it also needs to modify the measurement configuration data of an associated measurement agent. The RIM Conversion Agent MAY create a further internal RIM_Cert to protect this modified configuration data. Alternatively, the measurement agent and its configuration data SHOULD have been updated as well as part of the software update package. The exact implementation decision here is specific to the Engine stakeholder.

6.3.4.5 RIM revocation

During an update process, it may well happen that existing Internal RIM_Certs need to be removed from the set of authorized RIMs (because of revoked RIM_Auths or external RIM_Certs). If Internal RIM Certs are not utilized then this section does not apply.

It may be necessary to remove the requirement for a software component entirely, based on a larger update that makes an old component obsolete. The platform MUST be in the "SUCCESS" state before this revocation process begins.

The RIM_Auth sends an updated validity list (or lists) to the Engine via any suitable update protocol, as described in Section 6.3.4.6. Refer to sections 5.2.3 and 5.2.5 for Storage and use or RIM validity lists.

The RIM Conversion Agent MUST review the RIM_Cert validity list to confirm that all of the external RIM_Certs that were used to create current internal RIM_Certs are contained on that validity list. If it finds an internal RIM_Cert whose external counterpart is no longer on the validity list then it MUST remove that RIM_Cert from the Engine's internal store. If there is a RIM_Auth that is no longer on the RIM_Auth validity list signed by the parent RIM_Auth, then the RIM Conversion Agent MUST identify all internal RIM_Certs that were authorized by that RIM_Auth and remove them all from the internal store.

Once all of the revoked internal RIM_Certs have been removed, then the RIM Conversion Agent MUST re-validate the complete set of Internal RIM_Certs for the Engine by running MTM_InstallRIM with an incremented counter value. Once the new set of Internal RIM_Certs has been created, the RIM Conversion Agent MUST then increment the Engine-specific CounterRIMProtect. All the old RIM_Certs MUST be removed from the Engine's internal store of RIM_Certs and the new RIM_Certs MUST be added.

Where a RIM is being removed with no replacement, then the corresponding software MAY also need to be removed from the Engine. This depends on whether the wider change makes the whole software obsolete, or just means it no longer needs to be verified during boot. The Engine MAY need to modify the measurement configuration data of an associated measurement agent, either to skip a measurement entirely, or to measure without verifying. This MAY entail the RIM Conversion Agent creating a further internal RIM_Cert to protect this modified configuration data. Alternatively, the measurement agent and its configuration data MAY have been updated as well as part of the software update package. The exact implementation decision here is specific to the Engine stakeholder.

6.3.4.6 Protocol for Updating RIM_Auths and External RIM_Certs

This version of the specification does not define a specific protocol for providing (external) RIM_Certs, RIM_Auth_Certs and Validity Lists to an Engine.

Start of informative comment:

In abstract, such a protocol may be summarized as follows:

Authorized Party **Engine**

(Optional) Get_RIMAuthsAndCerts_Req({Parameters}) ←

→ Get_RIMAuthsAndCerts_Resp ([RIM_Auth_Certs],

[RIM_Auth_Validity_Lists], [RIM_Certs], [RIM_Validity_Lists], {Other Data})

(Optional) The Engine discovers it needs updated RIM_Certs, RIM_Auth_Certs or Validity Lists. This may happen because the Engine discovers one of its existing Validity Lists has expired, or the Engine may receive an update for a target object that is currently protected by a RIM, and needs to find an updated RIM_Cert in order to install that object.

The Engine contacts an authorized party to receive updated data. The "authorized party" may be a RIM_Auth or the Engine Owner, may be a Device Management Server, or may just be a web site pre-configured in the Engine. The Engine may pass optional parameters in the update request indicating which objects it needs updating (e.g. labels for RIM_Certs or RIM_Auth key_ids).

(Required) An authorized party provides an update package of data to the Engine containing the following:

- List of RIM_Auth_Certs, as requested by the Engine, or where necessary to build a certificate chain for other certs delivered in the package.
- List of RIM_Auth_Validity_Lists, as requested by the Engine, or where necessary to validate RIM_Auth_Certs delivered in the package.

- List of RIM_Certs, as requested by the Engine, or where necessary to protect other data delivered in the package.
- List of RIM_Validity_Lists, as requested by the Engine, or where necessary to validate RIM_Certs delivered in the package.
- Any other data that might be associated with RIMs (e.g. updated objects themselves, updated measurement agents or MA configuration data).

Note that an update package may be provided by an authorized party at any time, with or without a prior request from the Engine.

There are no assumptions on the security of the underlying protocol used to deliver the update package (or to send the request for updates). Nor are there any restrictions on means used to transport the update to the Device: a variety of methods are acceptable (e.g. http, email, WAP-push, broadcast, memory card, Bluetooth.)

***End of informative comment*.**

6.3.4.7 Protocol for Reporting RIM_Auths and RIM_Certs

This version of the specification does not define a specific protocol for an Engine to report its current set of RIM_Certs, RIM_Auth_Certs and Validity Lists to an Authorized Party. However, it is expected that such reporting is done as part of remote attestation.

This version of the specification does not define a specific protocol for an Engine to report its current set of RIM_Certs, RIM_Auth_Certs and Validity Lists to an Authorized Party. However, it is expected that such reporting is done as part of remote attestation.

It is **not** RECOMMENDED to provide certs which uniquely identify the device during attestation.

Start of informative comment:

In abstract, such a protocol may be summarized as follows:

Authorized Party **Engine**

(Optional) → Get_RIMAuthsAndCerts_Req({Parameters})

Get_RIMAuthsAndCerts_Resp([RIM_Auth_Certs],

[RIM_Auth_Validity_Lists], [RIM_Certs], [RIM_Validity_Lists], {Other Data}) ←

(Optional) An Authorized Party requests to know some information about an Engine's current set of RIM_Certs, RIM_Auth_Certs or Validity Lists. This request may be implicit or explicit during a remote attestation event, or may occur during an update event as described above, e.g. where the Authorized Party wishes to know what the Engine currently has before providing any updates.

The Engine may pass optional parameters in the request indicating which specific objects it needs to know about from the Engine, and a nonce to be included in any signed response.

(Required) The Engine digitally signs a response to the Authorized Party containing any or all of the following information (e.g. as requested by the Authorized Party):

- List of RIM_Auth_Certs currently accepted as valid by the Engine
- List of RIM_Auth_Validity_Lists currently stored by the Engine
- List of RIM_Certs currently accepted as valid by the Engine (typically these will be internal RIM_Certs)
- List of RIM_Validity_Lists currently stored by the Engine
- Any other data that might be associated with RIMs and used to prove that the Engine is using RIMs correctly (e.g. measurement agent configuration data).

Note that such signed data may be provided by default to certain Authorized Parties, even without an explicit request.

End of informative comment.

An Authorized Party MUST be able to recognize a reporting key as belonging to the Engine concerned. For security reasons, this key MUST NOT be an Attestation Identity Key. This is because the AIK can not sign abritory data as this would allow forging of TPM_Quote results. Instead, the Engine SHOULD generate an additional key-pair for signing reports, and have the public half of the key signed by an Attestation Identity Key (using TPM_CertifyKey).

6.3.5 Backup, Recovery, Maintenance and Migration

6.3.5.1 Backup of internal RIM_Certs

In this version of the specification, backup mechanisms for the software components of the product are **not** REQUIRED and not defined. There are no defined protocols for backing up and recovering RIM_Certs.

Start of informative comment:

The internal RIM_Certs that have been created by the Device are used in the process of validating the software to establish the trust level. The internal certs are integrity protected using the MTM. The platform may have an option to read the entire set of internal RIM_Certs as a block of data that can then be exported to some external storage site. That external storage site may be accessed through any communication channel available to the platform for storage.

End of informative comment.

6.3.5.2 Recovery of internal RIM_Certs to original device

If a engine determines that the set of internal RIM_Certs are missing or corrupted then it MAY attempt to replace them from a backup version. The platform state will be in the “INITIALISATION” state (see Section 7.1.2) as it starts up and an optional feature of this state may be a limited communication channel that allows the backup set of internal RIM_Certs to be restored to the device. Once the information is restored then a reset MUST be triggered to start the standard boot process as defined earlier in this section.

Start of informative comment:

The set of internal RIM_Certs will be verified before use if they are restored to the same MTM, and their integrity will be validated. The boot process will verify the MTM_COUNTER_REFERENCE in each RIM_Cert against the Engine Counter_RIMProtect to protect against any attempt to roll back to previously valid but now revoked RIM_Certs. There should not be any security implications of backing up the internal RIM_Certs and allowing them to be restored to the same platform.

End of informative comment.

If this restore option is not available in the “INITIALISATION” state, then the only recourse will be to reload the platform memory and follow the procedure defined for a pristine boot.

6.3.5.3 Recovery of internal RIM_Certs to replacement device

At present this type of operation is out of scope for this version of the specification.

Start of informative comment:

While it might be possible to define a path using the maintenance features of the MTM (if supported) to obtain a set of internal RIM_Certs from a backed up set of internal RIM_Certs on another device, this path is not currently defined.

End of informative comment.

6.4 Debug Mode

Start of informative comment:

The platform has multiple levels of debug capability. A technician proves his permitted level of access using a cryptographic challenge and response protocol.

At the lowest debug level, the entire platform is rebooted into a special state, where the platform erases all data belonging to all Trusted Services in all engines. When debug has finished, the platform is rebooted and the user is prompted to load protected backups, to restore previous secrets. This mode is typically used when data backups are available, or can easily be reinstalled from scratch.

At an intermediate level, the technician can execute test routines but not touch sensitive data. This mode is typically used when data backups are not available and are difficult to reinstall from scratch.

At the highest level, the technician has full access to all platform resources with the exception that he cannot inspect or copy the values that the platform uses as keys. This mode is typically used when data itself appears to be causing faulty operation.

A platform may offer a trusted authenticated mode for the intermediate and high levels of trust. This may require a technician to authenticate the tools and/or personal credentials to receive authorization to temporarily enable these debug modes. In these cases it may not be required to reboot the platform if the control methods are proven sufficient to protect the information on the platform.

End of informative comment.

7. Requirements For Maintaining Integrity

As discussed in Section 4.1.2, engines in a mobile platform are generally **not** REQUIRED to support an RTV. All normative statements in this section apply to an engine *conditional* on that engine supporting an RTV.

The DM's Engine MUST support an RTV, and any other engines with remote owners (i.e. owners who are not local Users of the platform) SHOULD support an RTV. An engine with a local owner (i.e. a User engine) MAY support an RTV.

7.1 Operations

7.1.1 Introduction

Start of informative comment:

Previous sections (see especially Section 5) have addressed the requirement that the platform must boot into a pre-specified trusted state. This section focuses on maintaining and ensuring a trusted state after the boot process, in the face of detected runtime failures or threats. Run-time protection of data assets and a platform threat response is needed by all the use cases foreseen by the Mobile Phone Working Group [Ref: use Cases].

For simplicity, the five Engine states described in Section 4.1.1 are collapsed into three states. The "*Engine Reset*" and "*RoT Initialisation*" states are collapsed into an "INITIALISATION" state. The "*Engine-Loading*" and "*Engine-Verified*" states are collapsed into a "SUCCESS" state. The "*Engine-Failed*" state is relabelled as a "FAILED" state. The collapsed transition diagram between these states is shown below (Figure 7):

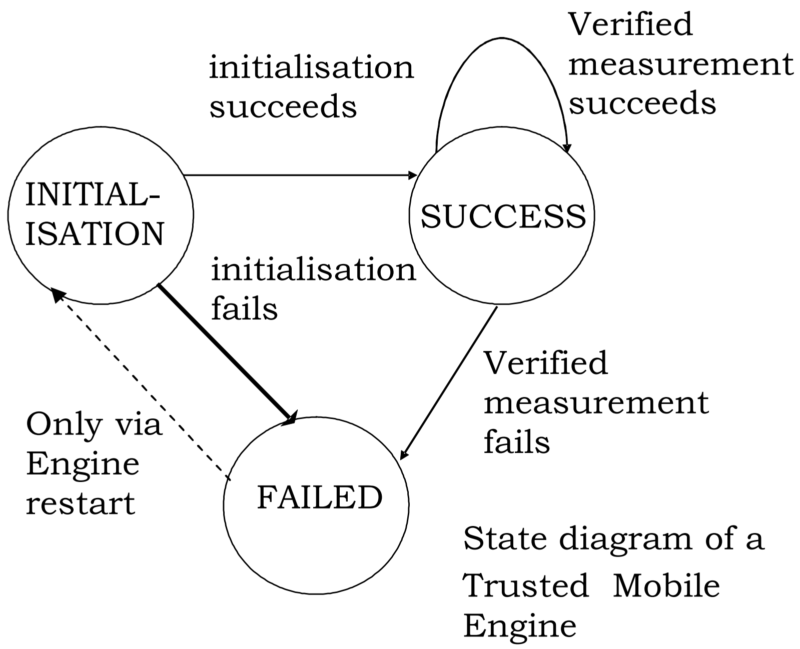

Figure 7. Engine State Machine and Transition Diagram

Ideally, the Device would completely prevent any changes, deliberate or accidental, that impact the trusted state of the Device. This is called a "Preventative Approach" and consists of mechanisms to ensure that the Device remains in a "SUCCESS" state throughout its run-time operation. In addition to such an approach, it is desirable to give some components high levels of protection against physical and software attack, and then use these components to provide fault detection and secure response services to the rest of the Device. Effectively, this foresees that the Device may at some point transition out of a "SUCCESS" state into a "FAILED" state, and provides mechanisms to limit the consequences. This approach is termed the "Reactive Approach".

End of informative comment.

7.1.2 Security States

The notion of a "SUCCESS" or a "FAILED" state needs to be phrased quite carefully to avoid artificially restricting possible implementations. To start with, consider a platform with just a single Engine, and then turn to the multi-Engine model.

Start of informative comment:

Platform State Machine

A "Platform Secure State Machine" here serves as an abstraction, rather than some specific running process. Crudely speaking, the abstraction splits the platform's numerous microstates into three big groups labelled "INITIALISATION", "SUCCESS" and "FAILED". Then, for example, whenever the platform transitions from any one of the Success states into any one of the Failed states, the (abstract) State Machine transitions from "SUCCESS" to "FAILED". And similarly for other transitions.

The existence of a "real" Platform Secure State Machine, coded within the platform, is not a requirement. In particular, there doesn't have to be a specific variable in protected storage on the platform which tells the platform whether it is "INITIALISATION" or "SUCCESS" or "FAILED" (and with the response by the platform dependent on the value of this variable).

***End of informative comment*.**

Difference between "SUCCESS" and "FAILED" States

To avoid making an arbitrary split, there is a simple stipulation. The "SUCCESS" state is at least one in which there is some confidential, protected information available on the platform. Such data MUST NOT be available in the "FAILED" state.

Thus the RTS MUST be operational in a "SUCCESS" state, and MUST NOT be operational in a "FAILED" state. If the RTS is "on", that means at least some secret keys are available (such as the EK and SRK). If "off", then clearly no secret keys or other confidential info are available, unless they were being temporarily stored outside the RTS (and if they were, they SHOULD now be erased to stay confidential).

Start of informative comment:

Notice that the implication of an RTS not available is that the RTR is also unavailable (it can't report the state without PCR values or keys). It also means that the RTM/RTV and other measurement/ verification agents can't work: they have no way to extend PCRs or verify their correctness against a RIM Cert. So shutting down the RTS effectively shuts down all the other Roots of Trust as well, or at least forces them to be suspended.

- There may be other consequences of a "FAILED" state as well (such as a forced shut down or suspension of all platform activities), but the above is the minimum the specification mandates. Out of scope for the specification and left to the platform implementation are the following decisions:
  - Whether there are one or more state indicator variables on the platform, and what processes are authorized to update them.
  - What, if any alerts will be delivered to the user, operator, other software, etc, when the platform state changes to "FAILED".
  - What services will still be available after the state changes to "FAILED" (for example emergency calls, flash light, camera, broadcast reception, etc).
  - Whether the platform can keep running or not without an RTS (or other RoTs). Clearly, if it can't by design, then the only thing that can happen next is a reboot. If it can keep running, then there SHOULD be a user warning, but the user could decide to keep the platform going with severely degraded functionality.

***End of informative comment*.**

Transition Control through RIMs

Simply having the RTS "on" or "off" can't be the sole *definition* of a "SUCCESS" state as this transition needs to be controllable (via RIMs). Also, there is the initial state in which an Engine starts up. This will be one where the RTS is not fully Operational, but is on the way to becoming Operational, and where nothing has yet been verified. So define the following equivalences:

"INITIALISATION" State

== RTS not (fully) Operational, but can become Operational

== Since platform start-up, no TIMs have yet been verified against RIMs.

"SUCCESS" State

== RTS Operational

== Since platform start-up, all TIMs requiring verification have matched corresponding RIMs

"FAILED" State

== RTS not Operational, and cannot become Operational

== Since platform start-up, some TIM requiring verification has failed to match a RIM

Start of informative comment:

This defines the transition from a "SUCCESS" to a "FAILED" state. It also defines the reverse transition – the only way to get "out" of a "FAILED" state is to do a reboot and hence go back through an "INITIALISATION" state. Finally, note that if the very first verification during boot fails, then the state will transition directly from "INITIALISATION" to "FAILED".

This may seem a bit restrictive, but it does make operational sense. In a "FAILED" state, the RTS has been switched off (or put into a non-functional state). This means the only way to get it "on" again safely (and with a safe and consistent set of PCR values), is via a reboot. The analogy in existing TCG specs would be with a MTM failing a self-test: it is not possible to recover from that without a reboot.

Also, it clarifies the concept of a "FAILED" state as in some sense a fatal error, rather than a recoverable error, requiring a special Reactive Response, Arguably, if a platform was able to securely "recover" itself through normal operation, then it never really left "SUCCESS".

Multi-Engine Transitions

As the "FAILED" state is rather fatal, greater flexibility requires a continuum of degraded states down from "SUCCESS" to "FAILED", rather than just a binary split. It does seem rather inflexible for a single RIM/TIM mismatch to be able to massively impair or force a reboot of the whole platform but not do something less drastic (e.g. just shut down some services).

Nevertheless, the same basic concept can give the required flexibility through the "Multi-Engine" design.

End of informative comment.

The above definitions related to a simple model where there is one platform wide security state, and one RTS for the platform. In effect, there is the DM's Engine and no other Engines.

If instead the platform has several Engines, each with its own RTS (possibly logical rather than physical), there are actually several *Engine specific* Secure State Machines:

DM's Engine "SUCCESS"

== DM's Engine RTS Operational

== Since DM Engine startup, all TIMs checked by DM's Engine have so far matched RIMs

Engine 2 "SUCCESS"

== Engine 2's RTS Operational

== Since Engine 2 startup, all TIMs checked by Engine 2 have so far matched RIMs

and so on.

Start of informative comment:

The implication is that a RIM mismatch could shut down the RTS for just one Engine (Communications Carrier's Engine, User Engine, SP Engine) but keep other RTS's on the platform going. This would enable the platform as a whole to have a partially degraded state without complete degradation.

It would still be the case that for any particular Engine, the only way to get that Engine out of a "FAILED" state and back into a "SUCCESS" state would be to restart that Engine. However, notice that the DM's Engine could attempt such a restart for another Engine on the platform without rebooting the whole platform.

The worst case degradation would be for the DM's Engine to go to a "FAILED" state. There is no way out of that without rebooting the DM's Engine (and hence the entire platform). So it makes sense to equate the maximally degraded "FAILED" state for the entire platform with the "FAILED" state for the DM's Engine.

Notes on Debug Mode

Depending on how debug mode works, a debug run of the Device can be modelled as a run which never actually enters the "SUCCESS" state in the first place. The Debug state may be a special sort of "INITIALISATION" state with its own boot-up rules.

Alternatively, if the Device can be switched into Debug mode mid-operation, this will generally induce a transition out of "SUCCESS" to "FAILED" and will involve all the Reactive consequences (RTS shuts down, user warnings etc.) Whether the Device is able to be meaningfully debugged under those circumstances is a bit doubtful; it certainly can't be switched out of debug mode again without a re-boot.

The final alternative is a form of limited debugging which enables the Device to remain in a "SUCCESS" state. The RTS remains operational while debugging, but its stored secrets are protected from exposure to the debug operator, and the platform integrity as a whole is not affected by the debug operations.

End of informative comment.

7.1.3 Protecting Mandatory Functions

The MPWG approach is really a combination of the Preventative and Reactive approaches. This specification provides for MANDATORY run-time integrity to protect the integrity of **mandatory** platform functions.

Start of informative comment:

Mandatory platform functions need to be understood here in an operational sense as "functions which are defined by the manufacturer – and other external authorities empowered by the manufacturer - as essential for proper and secure operation of the platform".

Exactly as discussed in Section 5 for boot-time integrity, the presence of such functions is defined by TIMs matching RIMs, which is equivalent to a "SUCCESS" state. The protection of such mandatory functions needs to be met in part by the RTS (see later), but also met by Verification Agent(s) running within the platform. These are not necessarily the Root of Trust (e.g. RTV), although the RTV must support such VAs through a transitive trust chain (again see later).

End of informative comment.

The protection of mandatory functions can be understood with reference to three classes of attack on the Device, of increasing power:

Attack 1. Threat handled by a preventative response. This sort of attack is resisted by the platform, enabling it to remain in a "SUCCESS" state throughout, and the mandatory functions are never impaired.

Attack 2. Threat handled by a reactive security response. This sort of attack impairs some of the platform's mandatory functions (causing TIMs not to match RIMs).The platform transitions to a "FAILED" state, but the security response to such a state is not impaired. In particular, confidential material is no longer available to the platform. Or the whole platform operation may be suspended.

Attack 3. Threat not handled by any response. In this case, the attack evades all preventative responses, and also impairs the reactive response. Nothing can be guaranteed about what happens then. Behaviour in circumstances of Attack 3 is outside the scope of this specification.

Start of informative comment:

It is understood that manufacturers cannot prevent Attack 3 altogether, just ensure that it requires highly advanced techniques/powers outside the scope of the protection profile covering the platform's most protected capabilities. So the platform's compliance to this spec is covered by the conditions of the protection profile.

Notice that if mandatory functions were regarded as *absolutely* MANDATORY, then the spec would have to say that Attack 2 can't happen. In which case Attack 2 would require techniques/powers outside the scope of a protection profile covering *the entire platform*. This would be much more expensive/difficult to engineer. Hence we consider a reactive response as allowed within the specification, and define mandatory support to enable this response.

End of informative comment.

In addition to specifying mechanisms for, and requiring runtime integrity protection for mandatory functions and components, this specification also allows for these mechanisms to be used to protect the runtime integrity of discretionary platform components, and it is RECOMMENDED that the runtime integrity of discretionary platforms components is protected in this way. For simplicity, the protection mechanisms are defined to be the same as for protecting the mandatory functions: it is just that they are directed to functions running in the discretionary domain (e.g. for a platform Engine other than the Device Manufacturer's Engine, running in the discretionary domain).

7.1.4 Application Integrity and Data Integrity

Start of informative comment:

These mandatory features (and most of the recommendations) are addressed to the integrity of *functions*, that is, in general, to *applications* running on the Device. This does not mean that *data* integrity is unimportant, simply that it is difficult to address at the platform level in either a preventative or reactive response.

In most cases, it is expected that the application code to be protected on the Device is fixed, or else changes relatively infrequently (at install of new applications). This makes it relatively easy to compare the applications that are *actually* installed or running on the Device with reference images of what they are *supposed* to be.

By contrast, data will change frequently, on each run through an application, on input and output from the Device, and at the user's discretion. Also the location of such data is highly mobile: there is no reason to expect it to remain in a fixed location in the Device. This makes it virtually impossible to decide in advance what is "good" data (and hence prevent changes to it) or decide after the fact what is "bad" data (and hence trigger a reaction).

In general, where data integrity is vital, it is recommended to protect it *through* applications, and then to protect the integrity *of* those applications. Wherever the application writes data needing integrity protection, it should add a redundancy check or hash (which protects against accidental corruption) or a cryptographic MAC or signature (which protects against deliberate tampering). Wherever the application reads data needing integrity protection, it should verify the redundancy check or the MAC/signature. Further, the application must be designed to strictly segregate data from its own code (thus avoiding unpredictable behaviour and attacks such as buffer overruns). Finally, the application code can be integrity protected by the platform, giving a complete solution.

Thus it is sufficient for the spec to define means to protect application integrity (through the RTS, VAs etc.) and give applications the means to protect data integrity where needed (through secure storage of integrity keys by the RTS).

End of informative comment.

7.2 Preventative Methods

Start of informative comment:

The aim of preventative measures is to keep the state at "SUCCESS", either for the platform as a whole (mandatory functions) or for a particular Engine (discretionary functions). There is a choice of techniques for preventing transitions out of a "SUCCESS" state.

End of informative comment.

7.2.1 Hardware Protection

Within this specification, one or more forms of hardware protection MUST form the basis of any preventative approach. The most basic form of hardware protection is implementation of program code in ROM and storage of critical and unchanging data in ROM and/or One Time Programmable (OTP) memory. More advanced levels of additional hardware protection (epoxy coverings, protection against power analysis and fault induction attacks) can then be added to protect this ROM and OTP.

These levels of hardware protection will be defined in a TCG Protection Profile. The design intention is that capabilities protected by hardware cannot be interfered with either by rogue software (installed onto the Device post manufacture) or by physical attack on the Device.

Within this specification, such hardware protected capabilities are REQUIRED to implement the Roots of Trust, and are REQUIRED to enforce a security response on exiting a "SUCCESS" state. Note that this requirement does not mean the Roots of Trust and security response mechanisms must actually be implemented in hardware, but that their implementations are protected by hardware. For example the code implementing the Root of Trusts could be checked by a function implemented in ROM code (hardware) which compares a hash of the code implementing the roots of trust code with an expected value stored in OTP memory (hardware).

7.2.2 Software Isolation

Start of informative comment:

With regard to software-based preventative approaches, the basic method is code isolation, that is, more trusted code is made inaccessible to less trusted code. An example of such code isolation would be code running in dedicated hardware (e.g. a TPM chip). A more flexible example is a runtime "trusted execution environment" that is isolated from the main OS and used for security critical functions. An even more flexible example is "full virtualisation" whereby the total device code can be divided into many sections, all of which are isolated from each other. Effectively, each piece of code "appears" to be getting the machine all to itself, and cannot interrupt the function or access the protected memory of other code.

End of informative comment.

The design intention is that capabilities protected by software isolation cannot be interfered with by other "normal" software.

Within this specification, such software isolated capabilities are REQUIRED to implement the Roots of Trust and at least one run-time Verification Agent, and are REQUIRED to safeguard the Roots of Trust if the platform exits a "SUCCESS" state.

7.2.3 Software Simplification

Start of informative comment:

It is commonly accepted that it is easier to certify that a given program code has no errors, if that code is limited in size and complexity. This leads to the approach of using a small kernel for all resource access. This small kernel is executing at the maximum privilege level. All other code, applications and the largest part of the operating system is executing at lower privilege levels. To have a further distinction in privilege level between the operating system code and the application code, two more privilege levels are needed in addition to the maximum (kernel) privilege.

End of informative comment.

The design intention is to code a small kernel which does not have any errors. This gives full protection not only against malicious application code but also against undesirable behaviour arising from programming errors, and active exploits of such behaviour. This also ensures that any privilege restrictions intended to be imposed by the OS (see below) are in fact imposed.

It is conceivable that attack code acquiring kernel mode privileges could subvert the Roots of Trust, interfere with platform Verification Agents that detect an exit from a “SUCCESS” state, or prevent the correct security response on exiting a “SUCCESS” state.

The Device design MUST provide protection ensuring that attacks using kernel mode privileges could not subvert the Roots of Trust, OR ensure that the kernel is of sufficiently low complexity as to be certifiably resistant to such attacks.

7.2.4 Software Restriction

Start of informative comment:

At the Device OS level, a comparatively simple and inexpensive approach is to not provide any tools to import native code onto the platform. If download of applications is required (it may not be required), this is only enabled via an application environment based on a virtual machine. The virtual machine’s strict security model allows importing and executing of interpreted code only. The security policy for the virtual machines is set up to only allow certain critical operations to applications that are considered “trusted” according to the security policy of the virtual machine. An example of such a controlled environment is an OSGi framework on a Java virtual machine (see http://www.osgi.org) or MIDP 2.0.

A related approach is to use an operating system with strong access control and a policy system, which can be tailored flexibly to protect the resources. In particular, APIs made available to applications running in the OS are segregated into at least two privilege classes, and the OS prevents applications using the more privileged APIs unless they can be recognized at install as “trusted” according to the security policy.

Both the above approaches have limitations, notably the loss of flexibility in the programmability of the Device. Also they are not risk free. Here are some of the obvious risks:

1. Malicious application code can in theory be barred from all direct access to Device capabilities, or from all access to privileged APIs. But programming errors in either the main OS or the virtual machine could offer starting points for exploitation.

2. “Trusted” software (more precisely signed software verified using a valid certificate) may itself be malicious. It might use privileged APIs in an inappropriate way, and unless an active revocation-checking capability is in place, cannot be prevented from installing once signed.

3. “Trusted” software may itself contain (unintentional) programming errors, which could be exploited by malicious application code, allowing untrusted applications to make use of the privileged APIs.

On the other hand, both approaches do significantly improve the Device’s preventative response capabilities, and can be implemented with conventional processor architectures without change.

End of informative comment.

Given that preventative measures are not perfect, the Device MUST ensure that any failure to correctly restrict software privileges within the main OS either **cannot** impair the platform’s mandatory functions (so can’t force the platform out of a “SUCCESS” state) or else **cannot** impair the platform’s security response when leaving a “SUCCESS” state.

7.2.5 Software Load

Further security techniques MAY be deployed to prevent malicious (or just badly written) applications being loaded onto the device at all. These may be applied at application *install* (when the passive application code is first loaded onto the Device) or at application *launch* (when the application's active executable image is loaded into memory).

While these techniques MAY be used in general, they MUST be used in some restricted circumstances (see below), and thus capabilities to support them are REQUIRED by this specification. Also, this section references the RIM update protocol defined in Section 5.2, enabling external authorities (RIM Auths) to instruct the Device to use these techniques. Where the Device supports the RIM update protocol, it MUST follow the instructions of RIM Auths in this respect.

At Installation:

The Device MAY prevent certain applications from being installed onto the Device post manufacture. This MUST be controlled by a security policy, and the state of the security policy SHOULD be protected by a RIM. It is RECOMMENDED that at least the following criteria are applied to determine whether the application install is blocked:

1. The application code is supplied with additional data indicating compatible Devices, and this Device is not one of them
2. The application code is supplied with an internal integrity check, and the check fails.
3. The application code does not match a RIM which the Device has been provided with to check such an application pre-install.
4. The application code is intended to execute in an OS with a privileged API structure, but does not clearly declare what privileges (APIs) etc it requires to execute.
5. The application code declares privileged APIs, but is not recognizable as "trusted" according to the security policy. (For example, it is not signed, the signature is invalid, there is no code-signing certificate, the code-signing certificate is not valid or is issued by an untrusted CA).
6. The application code declares APIs whose use could harm one or more of the Device's stakeholders (especially the Device User), but the source of the application is not identifiable.
7. The application code is identifiable by the Device as revoked (e.g. it has a revoked code-signing certificate or matches a revoked RIM).
8. The application code matches known signatures for malware (viruses, Trojans etc.)

Alternatively, where the Device allows an install despite some of the above conditions, the Device SHOULD warn the Device User that there may be danger in installing the application. If the User continues anyway, the Device SHOULD restrict the application's function along the lines discussed in the section "Software Restriction".

Consistency between Installation and Launch:

The Engine (or its associated RIM Conversion Agent, if external to the Engine) MUST have a capability to predict the expected image of some installed applications, as they will appear at application launch. This prediction MAY be achieved in some cases simply by measuring the code image just after it has been installed. This expected launch image SHALL be composed into the form of an internal RIM Cert, created by a RIM Conversion Agent (see Section 4). The internal RIM Cert MUST be associated with a target object and time of measurement, indicating to a suitable Measurement Agent that the installed application code (target object) must match the RIM either **prior to** any application launch or **at** any application launch (target time).

Such an internal RIM Cert MUST be created for any installed applications whose execution could impair mandatory functions. In particular, where the application itself is defined as a mandatory function (by a RIM_Auth through an external RIM Cert) then an internal RIM Cert MUST be created. In addition, even where not defined as a mandatory function, an internal RIM Cert SHOULD be created for any OS updates, and SHOULD be created by default for applications using privileged APIs.

For simplicity, an internal RIM Cert MAY be created by default after *any* application install, however some care must be taken here. The impact if the application image at a later date does not match the application image created at install might be that the Engine has a Reactive response, detecting an exit from a "SUCCESS" state and transitioning to a "FAILED" state. If the application was not very sensitive (e.g. just a stand-alone game whose code has been accidentally corrupted) then this could be an over-reaction. Accordingly, if it is the default policy to create internal RIM Certs at install, it is RECOMMENDED that they specify a validity time of "at launch" rather than "prior to launch" as this enables a Preventative response. The semantics of this are discussed below.

At Launch:

Start of informative comment:

Applications may be launched into operation either at boot-time or in the post-boot environment. Boot-time launch will be covered by the boot-time integrity checking of Section 4 i.e. before boot-time launch, any application that is protected by a RIM Cert is checked against the RIM. If there is a mis-match then the platform is in a "FAILED" state and the boot is aborted. Precisely similar logic can apply to launches in the post-boot environment.

However, in the post-boot environment, some thought needs to be taken about what happens if an installed application object does **not** then match a RIM. From a conservative point of view, this indicates there has already been an "attack" on the Engine concerned, and so the Engine is in a "FAILED" state, thereby triggering a Reactive Response.

On the other hand, the application code may just have been randomly corrupted, or subject to relatively crude tampering, which is not much of an "attack". If the application is just prevented from running, and was not itself a mandatory function, then there is no reason why the Engine should not still be considered "SUCCESS". So just preventing the affected code from running would be acceptable: a Preventative Response.

End of informative comment.

It is RECOMMENDED that the decision about whether to "Prevent" or "React" is determined through the time of measurement associated with the RIM_Cert. If the time indication is "prior to launch" then this means that even if the application never launches, a TIM will have already failed to match a RIM and so the state is to be regarded as "FAILED".

Alternatively, if the time indication is "at launch" then the Engine MAY still measure the target application immediately before launch, and if necessary, pre-emptively block the launch. In effect it notices that *if* it continued with the launch *then* from that point a TIM would mismatch a RIM, thereby creating a "FAILED" state. However, the Engine tries to avoid a transition to a "FAILED" state if possible, so pre-emptively decides to prevent this condition arising. This approach is consistent with RIM semantics, while also giving the maximum chance to "prevent" rather than "cure". A similar approach to all "event-based" RIM triggers is applied in the next section.

It is RECOMMENDED that if an Engine does pre-emptively block a launch in this way, then it gives a warning to the Device User what has happened. The User SHOULD then have an option to uninstall the application concerned, or attempt a repair/re-install.

7.3 Mandatory Support and Recommendations for Reactive Methods

Start of informative comment:

The reactive approach to maintaining trusted state is to attempt to detect any undesired modification of the system and then deal with it appropriately.

For detection a subset of the same measurements can be performed as during the boot phase. They can be performed periodically, for example by a kind of watchdog process. Alternatively, measurement could be triggered by particular events. It could be expected that components that are continuously used, such as the kernel itself or file management systems, would be checked on a periodic basis whereas components that are irregularly used would be checked on an event-driven basis.

End of informative comment.

7.3.1 Mandatory Support for Reactive Methods

Each Engine which supports an RTV MUST enforce a Reactive Response dictated by the security policy of its stakeholder, whenever a TIM is found to not match its associated RIM in that Engine. When such a TIM validation is associated with mandatory engine functionality and integrity, this is termed an Integrity Failure (IF). Upon an IF an engine MUST immediately deny access to security sensitive assets, including the RTS: this functionality MUST only be restored when the engine's integrity has been restored, through a secure boot.

To assure this response, the Engine SHOULD utilize TCG protected capabilities, termed "TCG_Reactive" protected capabilities. Such protected capabilities MUST be available to the engine, and where used, MUST ensure that the RTS can be turned OFF upon an IF notification and that it will stay off until the next boot cycle. TCG_Reactive capabilities MUST be able to enforce the security policy set by the stakeholder: that policy may require an immediate engine RESET.

If the DM's Engine is affected, and the Reactive Response does not require an immediate engine RESET, then the Engine SHOULD inform the user of a serious security error. If an Engine other than the DM's Engine is affected, the DM's Engine MAY attempt to restart the affected Engine. If relevant, the Device SHOULD warn the user that data/running processes on the affected Engine could be lost or else functionality temporarily impaired. (Note that if the Engine doesn't have data that can be lost on restart, and the restart is quick, this is not a requirement, and the recovery back to a "SUCCESS" state may occur without the User knowing this is happening.)

Transitive Run-time Trust-Chain

As discussed in Section 5.4, at boot time the RTV of an Engine must verify (at least one other) MVA, which measures and verifies other MVAs and so on. Some of these MVAs will keep running in the post-boot environment. To extend trust from a hardware base throughout run-time, the Device MUST support the following, at least within the DM's Engine:

- The Engine MUST support at least one MVA which carries on running as a mandatory function after OS start-up, and this MUST run within protected capabilities. This continuing MVA is **not** REQUIRED to be the RTV, but can be thought of as a run-time continuation of the RTV. It shall be referred to as a "Primary Run-time MVA" (PRMVA). Typically as "protected capabilities" do not encompass the main OS (their protection profile has a restricted perimeter), the PRMVA runs outside the main OS. It may be "beneath" the OS, or "in parallel" to the OS (e.g. virtualised from the main OS). The PRMVA MUST perform at least one form of scheduled i.e. time-based integrity measurement, and verify it using a RIM_run Cert.

- Unless the PRMVA is able to perform all run-time integrity checks by itself, the Engine MUST support and use at least one Secondary RMVA (SRMVA) running outside protected capabilities. The PRMVA SHALL then use *RIM_run* Certs (see below) to check the operation and integrity of the SRMVA. At least some checks by the PRMVA on the SRMVA SHALL be time-based, so that tampering with the SRMVA can only occur for a limited time before being detected.

- If it exists, the Secondary RMVA SHALL then use RIM_run Certs to check the operation and integrity of other engine components, which MAY include further RMVAs, e.g. running as applications within the OS. The SRMVA MAY perform regular time-based measurements, or irregular event-based measurements (e.g. triggered by an alert that another RMVA wants to measure and verify something).

Other secure boot Engines MAY also have protected capabilities to support their own PRMVAs. If they don't, they MUST still have RMVAs, and there MUST be a transitive run-time trust chain from the Device Manufacturer's PRMVA through to the RMVAs of each other Engine. This MAY involve the DM’s PRMVA making direct measurement and verification of other Engines, or indirect measurement and verification through the DM's secondary RMVA, or even more indirectly through other RMVAs on the DM's Engine.

Start of informative comment:

The PRMVA may be able to measure the code of the SRMVA as it is executing e.g. the SRMVA may be part of the OS kernel, in which case it would suffice for the PRMVA to check the operation and integrity of the kernel. Or the PRMVA may just measure some regular output of the SRMVA, where the Engine design is such that tampering with the SRMVA would become evident from an unexpected output.

A minimal implementation is that the PRMVA expects to receive an “OK” message from the SRMVA within a given time interval (this message is designed to be hard to forge by an attacker who attempts to tamper with/replace the SRMVA). When the time interval elapses, then the PRMVA attempts to measure the message from the SRMVA; if no message has been received, or it is not the expected “OK”, then a TIM does not match a corresponding RIM, and the PRMVA triggers the TCG_Reactive protected capabilities. The sending of a message by the SRMVA may also trigger an immediate event-based measurement by the PRMVA, so the PRMVA does not have to wait until the time interval elapses to check if the message is an “OK”.

In such an implementation, the conjunction of the PRMVA (itself a protected capability) and the TCG_Reactive protected capabilities, can be regarded as a single protected capability, termed a Watchdog Timer (WDT). Such a WDT has the following functional specification:

1. WDT TIMEOUT/FAILURE MUST generate an RTS_OFF signal (as part of a secured platform response)
2. WDT TIMEOUT/FAILURE MAY be configured to cause an automatic platform RESET
3. WDT cannot be turned off or the automatic RESET re-configured, once started; it can only be retriggered with possibly a new different count.
4. WDT MAY be explicitly triggered for a TIMEOUT/FAILURE by a SW command.
5. WDT can only be removed from the TIMEOUT/FAILURE state by a platform RESET

Properties 1 and 2 are implied because a measurement not matching a RIM creates a "FAILED" state and triggers the TCG_Reactive capabilities. Property 3 is implied by the PRMVA and TCG_Reactive both being protected capabilities. Property 4 is implied, as the Secondary RMVA may deliberately produce a “Not OK” message, which will mismatch the PRMVA's RIM_run Cert (when the PRMVA measures this message, and compares it with an expected “OK”) thus creating a "FAILED" state. Property 5 is implied by the fact that an engine can only leave a "FAILED" state by an engine reset.

End of informative comment.

RIM_run Certs

Abstractly, a RIM_run Cert is any structure which defines the authorized expected value of a run-time measurement. Concrete implementations MAY have the same structure as the RIM Cert defined in "TCG Mobile Trusted Module Specification" [5], or MAY have a simpler (proprietary) structure. RIM_run Certs MAY be internal or external. Like any other RIM, failure of a TIM to match a RIM_run MUST cause the engine to transition to a "FAILED" state.

Within this specification it is REQUIRED that if there is a non-trivial transitive chain of trust (i.e. if the PRMVA does not perform all run-time verifications), then links in that chain are supported by integrity measurements, verified using a non-trivial set of RIM_run Certs (i.e. at least one for each link in the chain).

Each Engine which supports an RTV MUST have a capability to take measurements at intervals of some executing code and MUST have a capability to check each such measurement against an authorised expected value. This expected value may be obtained in some cases simply by measuring the code image just after it has launched. The code is not required to be actively executing at the time at which the measurement is made and compared with its expected value. The measurement intervals MUST be defined by the Engine's stakeholder.

In general, a RIM_run Cert MUST be created where the engine stakeholder (or a RIM_Auth delegate of the stakeholder) has instructed that the executing image be subject to run-time integrity. This instruction mechanism MAY be supported explicitly through the mechanism of external RIM_Certs, or MAY be fixed implicitly by Engine design, or MAY be updateable through trusted Device management.

Where explicit, this instruction SHALL be provided through an extension to a RIM_Cert [see "TCG Mobile Trusted Module Specification"] indicating when to make measurements. The semantics of this extension MUST therefore allow to describe the following options:

- A specified (one-off) event in the boot sequence, as discussed in Section 5

  And/Or

- A specified (possibly recurrent) event in the run-time environment, such as an application install, application launch, a hardware event (e.g. TPM command), a write event to certain files or areas of memory, or a system interrupt.

  And/Or

- A specified (necessarily) recurrent interval of measurement in the run-time environment, defined in seconds and exceeding a platform minimum, or if not specified, a platform default interval.

The exact format of the extension and the means for coding the above options is outside the scope of this specification.

Mandatory Use of a MTM

A MTM MUST be able to support run-time integrity in at least the following aspects:

- The MTM MUST be able to store keys needed to verify external RIM_Certs.
- The MTM MUST be able to create, export, import and process internal RIM_Certs.
- The MTM MUST be able to store PCR values corresponding to previous measurements e.g. for comparison with current measurements.

These properties will all be achieved by use of a MRTM, or use of a MLTM with support for local verification commands; see "TCG Mobile Trusted Module Specification".

7.3.2 Recommendations for Reactive Methods

Start of informative comment:

Need for Integrity Checks

In contrast to boot time integrity checking, it may not be necessary that all software image objects should or can be checked during run-time – it may be sufficient for relevant stakeholders that a subset of the software image objects are checked (either on a periodic or event driven basis).

In addition to the automatic periodic execution and event driven execution, this integrity checking should also be available as a trusted service (probably provided by mandatory security capabilities) that applications or the OS can call. The measurement process, as well as its scheduling, must be part of the trusted and measured code. The measurement process will take the form of checking the measurements of software image objects against known reference values (the Reference Integrity Metrics, RIMs).

Application Integrity: Consistency before and after Launch

The value of the reference integrity metric (RIM) which contains the expected measurement of an image object may be different at run time to its value at boot time. Or in general, it may be different during execution of an application than it is before launch. This means that an engine needs a run-time RIM (RIM_run) which may be different from the boot-time RIM for the object. Further, unless special arrangements are made, the RIM_run may change over different instances of the boot sequence and at different launches. For example, the image may be loaded into a different section of memory each time, changing the values of pointers and hence the value of the image object's RIM_run overall.

One possible way to ensure that the value of a RIM_run is the same for each instance of the boot sequence is to ensure that the software image for that RIM is loaded into the same location in memory on each boot. This sort of restriction should apply to the OS kernel and to any Secondary RMVA that needs to be verified by the Primary RMVA. (Recall here that the PRMVA runs outside the OS and so does not have a dynamic memory map.) However, this technique clearly reduces the flexibility of the system with respect to memory allocation and can only be used to a limited extent.

A more flexible approach is for the RIM_run to be generated afresh for each boot sequence and/or post-boot application launch. Preferably as soon as possible after boot is completed or a launch image has been verified (as discussed in Section 8.2), so that the assumption that the Engine is in a "SUCCESS" state still holds. In such a case, the authorizing entity for the RIM_run, the RIM_run_Auth, is part of the Engine itself (e.g. the MTM of that Engine).

When an application is actually running, it tends to have a static part (occupying a consistent place in – possibly virtual - memory) and a dynamic part. The area of memory to which the static part is loaded is often not predictable at launch; however, after launch, the static part is expected to be stable, and so can form the basis for further verification Nevertheless, the exact behaviour is OS dependent, and there is no firm guarantee that the "static" part will not also move around from time to time.

End of informative comment.

An internal RIM_run Cert SHOULD be created for any launched applications which are defined as mandatory functions, or whose malfunction would compromise mandatory functions. In particular, such a RIM_run Cert SHOULD be created for the OS kernel. Also, to ensure a transitive chain of trust, such a RIM_run Cert SHOULD be created for at least one run-time Verification Agent within the main OS e.g. for the SRMVA.

Any part of the executing code image of any launched application which is defined as a mandatory function which – by Device design - is expected to be static SHOULD be measured and composed into the form of an internal RIM_run Cert. The internal RIM_run Cert MUST be associated with a target object and time of measurement, typically indicating to a suitable Measurement Agent that the static part of the executing code is to be checked at a regular interval (a defined frequency) or at particular events.

Time-based and Event-based Integrity

Start of informative comment:

Time-based integrity checks necessarily count as a Reactive approach rather than a Preventative approach. As something "bad" is already running, there is some sense in which security measures have already failed. Also, there is necessarily a delay, so the bad application may have already have done some damage. Nevertheless, they have some clear implementation advantages:

- It is easy to specify the conditions for carrying out the check, namely just a recurrence interval
- Provided the interval is not too long, the integrity check is unlikely to “miss” something critical. It might be late, but detection will happen.
- The measurement and integrity checking can run as a low intensity background process to avoid spikes in processor demand, and interruption of other activities.

Optimising the background process demand has an impact on how regularly checks can be performed, rather than whether they can be performed at all. The ideal is probably that the low-level background process is always checking e.g. if checks are scheduled every 30s, each check takes just under 30s.

By contrast, Event-based integrity checks could in theory be either preventative or reactive. This characteristic generalises the discussion in Section 7.2 concerning prevention or reaction to “bad” application launch.

Notice that if a RIM_Cert defines that a TIM must match a RIM **just before** a given event, then the response is necessarily reactive. If the Device checks, and discovers a mismatch, before the event then even if it blocks the event it is too late for anything but a Reactive response: a RIM check has already failed and the state is “FAILED”.

Alternatively, if a RIM_Cert defines that a TIM must match a RIM **at** a given event, then the Device has more flexibility. It could wait for the event, and only then carry out the measurement/verification; if that fails, this still gives rise to a Reactive response. Or, more pro-actively, just before accepting the event, the Device could attempt to predict what the TIM would be if the event occurred. If this mis-matches the RIM, then the Device can block the event. As there is never actually a mis-match between a RIM and a TIM, the state remains “SUCCESS”. So this is a Preventative Response.

How feasible event prevention really is depends a bit on how the RMVA runs. In one model, the RMVA is always running, frequently monitoring areas of memory/storage/hardware etc. for triggering events. Alternatively, the RMVA is dormant, and certain triggers are detected by the OS and used to wake up the RMVA. If the trigger event that wakes up the RMVA is precisely the event at which an integrity check is needed, then it will clearly not be possible to prevent this event.

It must be noted that one issue with event-based checking (either reactive or preventative) is a sudden spike in processor demand. As the event is pretty much instant, an integrity check and response will also be required pretty instantly, which is much harder to engineer than a low-level time-based process.

End of informative comment.

Partly for performance reasons, and partly because suitable events may not be defined or easily definable by the RIM_Auths, the use of event-based checking is **not** REQUIRED. However, it is RECOMMENDED.

In particular, at the level of the PRMVA it is possible to identify lots of potential trigger events, some of which SHOULD be used to carry out an integrity check. These are special events which are internal to the Roots of Trust. Of course, at the level of the Secondary RMVA or higher, other trigger events are possible, and the semantics for defining them are likely to be very platform dependent.

Recommended Use of the MTM

As well as the mandatory uses mentioned in Section 8.2.2, the MTM SHOULD be used to detect certain triggering events for run-time measurements. For example, certain PCR extends or uses of certain high sensitivity commands (like migration, management, delegation or owner changes) MAY act as triggering events.

Start of informative comment:

It is an interesting question whether a PCR should be repeatedly extended on repeated run-time integrity checks of the same target object. Typically an integrity check just detects that something is the same as when it was last measured. In that case, it is inefficient to keep extending the same PCR with the same value, as the value stored in the PCR will keep changing and a new (internal) RIM Cert will be needed to verify each extend. Also, repeatedly extending the same PCR makes attestation very complex, as the verifying entity must allow for how many times that PCR has been re-extended to work out if its current value is correct. If it has been re-extended every 5 seconds since the platform booted up a month ago, that will be a long log of PCR events to transmit and check.

It is therefore advisable to just to leave a PCR as it is, so it consistently stores a boot-time measurement, or maybe the first run-time measurement of the target object. This could give rise to a very simplified representation of a run-time RIM Cert within the MTM. Basically, there is just a flag raised on certain PCR values to check that the value of any new measurement is consistent with the stored value. If they do match, the new measurement value can just be discarded by the RMVA, and the PCR does not need to be extended. If they don't match, this is an error and the RMVA can readily enforce (at least part of) the security response by forcing the MTM to shut down.

End of informative comment.

It MAY be important to record in the MTM the fact that a re-measurement has happened (e.g. a proof that run-time integrity is working may also be needed for attestation purposes); if so, just discarding repeat measurement values will not achieve this. In such circumstances, it is RECOMMENDED to extend repeated measurements into a **different** - but matching - PCR from the original PCR, and then keep this matching PCR changing while the original PCR retains its boot-time value (or other first extended value).

Start of informative comment:

In that case, the "flag" idea doesn't quite work, but can be modified. The **re-measurement** PCR has the flag, and a pointer to the PCR used for first measurement. On each extend to the re-measurement PCR, the RMVA just checks that the value being extended matches that extended to the first measurement PCR.

Finally, in cases where a re-measurement value is **expected** to be different from the original measurement (i.e. because it has been changed by an event, and this change is expected from an external RIM_Cert), the simple flag model will break down. Rather than use up one more PCR for each new measurement value, it makes more sense to extend the changed value into the original PCR. However, there is now no easy way to detect that any further measurements will match the (updated) expected value, as the PCR value is now a function of two extends, rather than just the most recent extend.

In practice the only way to cope with this is by formally creating a new (internal) RIM_Cert whenever an expected measurement value changes, and importing it back into the MTM when verification is necessary. This is less efficient than a simple flag on certain PCRs of course.

End of informative comment.

[End of document]