TCG Specification

TPM 2.0 Mobile Reference Architecture

Family "2.0"

Level 00 Revision 142

16 December 2014

Contact: admin@trustedcomputinggroup.org

TCG Published

TCG

Disclaimers, Notices, and License Terms

Acknowledgements

TCG acknowledges the following contributors to this specification:

Alec Brusilovsky, Andreas Fuchs, Andrew Martin, Ariel Segall, Atul Shah, Carlin Covey, Carlton Northern, Cedric Colnot, David Wooten, Dean Liberty, Emily Ratliff, Gilles Peskine, Graeme Proudler, Hadi Nahari, Hervé Sibert, Ira McDonald, Jan-Erik Ekberg, Janne Uusilehto, John Mersh, John Padgette, Jon Geater, Joshua Schiffman, Kathleen McGill, Ken Nicolson, Michael Chan, Michael Peck, Niall O'Donoghue, Nicolas Ponsini, Paul England, Paul Waller, Rob Spiger, Ronald Aigner.

CONTENTS

1 Scope and Audience

The Trusted Computing Group TPM 2.0 [1] Specification defines a Trusted Platform Module (TPM).

This specification defines the reference architecture for the implementation of a TPM in modern mobile platforms. This TPM executes within a Protected Environment (Section 6) and is referred to as a TPM Mobile.

The architecture allows any possible implementation of the Protected Environment that meets the security requirements defined in Section 6. Several example implementations (Annexes A, B and C) are included. The implementation examples in this document do not in any way limit the allowed implementations.

Designers, developers and implementers of Trusted Computing technologies in mobile platforms are the target audience for this specification.

This specification supports the use cases defined in [7].

1.1 Key words

The upper-case key words “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,” “SHOULD NOT,” “MAY,” and “OPTIONAL” in this document indicate normative statements and are to be interpreted as described in [4].

1.2 Statement Type

There are two distinctive kinds of text throughout this reference architecture: *informative comments* and *normative statements*.

Most of the content consists of informative comments that explain why the architecture is as defined, but do not constrain implementation in any specific way.

Normative statements SHALL be obeyed if a standards compliant implementation is to be created. Normative statements are indicated by the presence in a statement of any of the upper-case key words listed in Section 1.1.

1.3 References

[1] Trusted Computing Group, Trusted Platform Module, Version 2.0, Parts 1-4, 2013

[2] GlobalPlatform Device Technology TEE System Architecture, GPD_SPE_009 [INFORMATIVE]

[3] Unified EFI Forum, UEFI Specification version 2.3, [INFORMATIVE]

[4] IETF, RFC 2119, March 1997

[5] IETF, RFC 4122, July 2005

[6] Trusted Computing Group, Virtualized Platform Architecture Specification, 2011

[7] Trusted Computing Group, Mobile Trusted Module 2.0 Use Cases, March 2011

[8] NIST SP 800-57 Part 1 rev 3, July 2012

[9] NIST SP 800-131A, January 2011

[10] IETF, RFC 4086, June 2005

[11] ISO/IEC 18031:2011 Information technology – Security techniques – Random bit generation Edition 2

[12] TCG PC Client Platform TPM Profile (PTP) Specification, February 2014

[13] TPM 2.0 Mobile Command Response Buffer Interface, February 2014

[14] Global Platform Card Specification V2.2.1, June 2010 [INFORMATIVE]

[15] ISO/IEC 7816-3:1997 Identification cards - Integrated circuit(s) cards with contacts - Part 3: Electronic signals and transmission protocols [INFORMATIVE]

[16] ETSI TS 102 226 (Release 6) Smart cards; Remote APDU structure for UICC based applications, European Telecommunications Standards Institute Project Smart Card Platform (EP SCP), 2004 [INFORMATIVE]

[17] GlobalPlatform Java Card API and Export File for Card Specification v2.2.1 v1.5 [INFORMATIVE]

[18] https://www.commoncriteriaportal.org/files/ppfiles/PP9911.pdf [INFORMATIVE]

[19] http://globalplatform.org/compliance.asp [INFORMATIVE]

1.4 Document structure

This document is divided and organized into Sections so that concepts are introduced in a logical sequence.

Section 2 contains a reference glossary defining the terms and definitions used throughout the specification.

Section 3 contains an introduction to the concepts and the rationale for the specification.

Section 4 provides an informative overview of typical mobile platform architectures.

Section 5 defines the boot sequences used by various platforms.

Section 6 defines the requirements, properties and capabilities that the Protected Environment needs to support to host a TPM Mobile.

Section 7 defines the requirements that a TPM Mobile implementation needs to meet to operate securely within a Protected Environment.

Section 8 deals with detailed TPM Mobile implementation aspects.

Annexes A, B and C describe possible implementation models for the Protected Environment and the TPM Mobile, including system aspects arising from each model.

2 Basic Definitions

2.1 Glossary

Glossary Term	Description
Application-specific integrated circuit (ASIC)	A microchip that is custom-designed for a specific application (in contrast to a general-purpose chip such as a microprocessor).
Attestation	The process of vouching for the accuracy of information. External entities can attest to protected locations and Roots of Trust. A platform can attest to its description of platform characteristics that affect the integrity (trustworthiness) of a platform. Both forms of attestation require reliable evidence of the attesting entity.
Critical Security Parameter (CSP)	Security-related information (for example, secret and private cryptographic keys, authentication data such as passwords and Personal Identification Numbers [PIN]) whose disclosure or modification can compromise the security of a cryptographic module.
Device	A shorthand in this document for a Mobile Device.
Device Manufacturer (DM)	The manufacturer or brand of a Device, typically an Original Equipment Manufacturer (OEM).
Device Owner (DO)	The legal owner of the device. The device owner could be an End User (consumer), an enterprise, a communications carrier, or some other entity. The Device Owner can customize all aspects of the TPM except the Platform Hierarchy.
Direct Memory Access (DMA)	**A** feature that enables certain hardware subsystems within a system to access system memory independently of the central processing unit (CPU).
End User	The ultimate consumer of mobile applications and services, particularly the user for whom the device is designed. The End User can also be the Device Owner.
Endorsement Primary Seed (EPS)	A primary seed (see Primary Seed) that the TPM uses to generate an endorsement hierarchy.
Enhanced Authorization (EA)	One of the mechanisms used in the TPM 2.0 Library Specification [1] to control access to the protected capabilities of the TPM.
Fuse	A mechanism of internal switches within the ASIC that can be electrically blown to enable write-once non-volatile storage of information within that ASIC.
Integrity Measurement	A value representing a platform characteristic that affects the integrity of a platform.
Measured Boot	A boot process where images are measured (for example by calculating their hashes) and the measurements are extended into the PCRs of a TPM. See Section 5.4 for more details.
Mobile Device	A physical entity encompassing all the hardware, firmware, software, and data necessary for it to function and provide services to an end user. Also known as a Mobile Platform.
Platform Configuration Register (PCR)	A shielded location within a TPM containing a digest of integrity measurements.

Primary Seed	A large random value persistently stored in a TPM (see TPM 2.0 Library Specification [1]). The TPM uses primary seeds to generate symmetric keys, asymmetric keys, other seeds, and proof values.
Protected Environment	A functional element that has its own execution and memory resources that are isolated from other components. See Section 6 for details.
Rich Operating System (Rich OS)	An environment created for versatility and richness where, for example, device applications such as Android, Symbian OS, and Windows Phone are executed. It is open to third party download after the device is manufactured. Security is a concern here but is secondary to other issues.
Root of Trust (RoT)	A component that performs one or more security-specific functions, such as measurement, storage, reporting, verification, and/or update. It is trusted always to behave in the expected manner, because its misbehavior cannot be detected. Note: A platform should have a set of Roots of Trust with at least the minimum set of functions to enable a description of its characteristics that affect the trustworthiness of the platform.
Root of Trust for Confidentiality (RTC)	A Root of Trust providing confidentiality for secrets stored in shielded locations accessed using protected capabilities.
Root of Trust for Integrity (RTI)	A Root of Trust providing integrity for integrity measurements stored in shielded locations accessed using protected capabilities.
Root of Trust for Measurement (RTM)	A Root of Trust that resets one or more PCRs, makes the initial integrity measurement, and extends it into a PCR.
Root of Trust for Reporting (RTR)	A Root of Trust that reliably provides authenticity and non-repudiation services for the purposes of attesting to the origin and integrity of platform characteristics.
Root of Trust for Storage (RTS)	The combination of the RTC and RTI.
Root of Trust for Update (RTU)	A Root of Trust for Verification that verifies the integrity of update payloads and the authorization to perform the update. Upon successful verifications, it initiates the update process.
Root of Trust for Verification (RTV)	A Root of Trust that verifies an integrity measurement against a policy.
Secure Boot	A boot process where each image is validated before execution. See Section 5.3 for more details.
TPM Installing Authority	The authority that installed the TPM Mobile into the protected environment. Typically this is the Device Manufacturer and it could be another entity who has the appropriate management privilege in the Protected Environment.
TPM Mobile	A TPM that complies with this specification.
Trusted Application (TA)	An application that runs as an isolated process within the Protected Environment.
Unified Extensible Firmware Interface(UEFI)	A software interface between an operating system and platform firmware. See [3] for details
Universally Unique IDentifier (UUID)	An identifier conforming to RFC 4122 [5].

3 Introduction

The evolution of mobile computing has led to the emergence of smart devices that are in almost constant personal and business use, for example for access to email, to social media, to application stores, to financial institutions, and other uses. Device Manufacturers need to incorporate trusted computing mechanisms to secure and support these and other mobile use cases. Such security mechanisms include techniques to ensure device integrity, attestation, isolation and protected storage.

The diversity of the mobile market place and the underlying computer architectures requires a model to support the TPM and to take advantage of the variety of security mechanisms, such as the protected environments provided by these platforms.

This specification adapts the mechanisms specified in the TPM 2.0 Library Specification [1] to take advantage of as many of these new architectures as possible and to extend the Trusted Computing Architecture into the mobile space.

3.1 Protected Environment-based TPM Implementation

The foundation of this architecture is a TPM Mobile implemented as a Trusted Application running within a Protected Environment.

This specification defines a set of requirements that the Protected Environment needs to meet to host a TPM Mobile (Section 6). There are no limitations on how the Protected Environment is implemented, providing the requirements are met.

Some examples of Protected Environment implementations are:

- Hardware isolation techniques
- Hypervisor within or outside an OS
- Separate cores within the same ASIC
- Separate ASIC

3.2 Example Implementations

To aid the understanding of the architecture, three example implementation models of the Protected Environment are described in informative annexes A, B and C.

4 Mobile Device Architecture - INFORMATIVE

This section is an informative introduction to the security-related architectural components of a contemporary mobile device. The focus is on features implemented in hardware or closely related to hardware. Although the reference setting is mobile phones, the same spectrum of hardware features can be present in other mobile devices such as, for instance, music players and tablets.

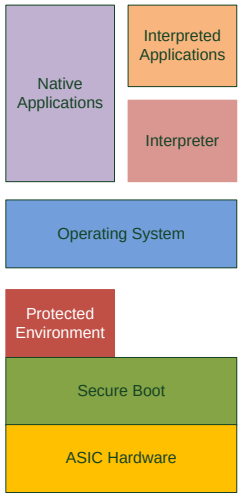


Figure 1 - Device with a Protected Environment

Figure 1 illustrates an architecture that is present in many mobile devices, and which is suitable for an implementation of the TPM Mobile specification. Processor features isolate the Protected Environment from the code executing in the Rich OS within the mobile device.

If the Protected Environment executes solely in on-chip memories, or if secure virtual memory techniques are used, the constructed isolation boundary resembles that of dedicated security hardware. If device secrets are protected by these mechanisms, their integrity and correctness is no longer contingent upon the integrity and correctness of any software component booted into the operating system or application domains.

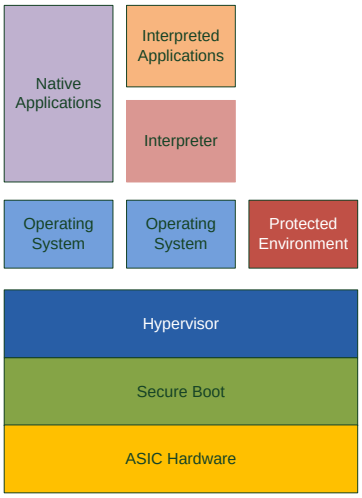


Figure 2 - Device with hypervisor

Figure 2 illustrates a hypervisor-enabled architecture that is useful for adding another operating system, such as a mobile protocol stack, to a mobile device running on single-core hardware. It has a significant advantage in reducing the exposure of secrets implemented in the hardware. Early in Secure Boot, the processor memory management unit (MMU) can be configured to give the hypervisor sole access to device secrets.

Hypervisor-enabled architectures can use Secure Boot and hypervisor-provided isolation to implement the Protected Environment requirements defined in Section 6. A TPM can be run in such a Protected Environment and can act as the TPM for a single guest OS or for the hypervisor itself.

This specification does not define the complete architecture necessary to support hypervisor-enabled environments. TCG has defined the architecture for such support; see [6] for more details.

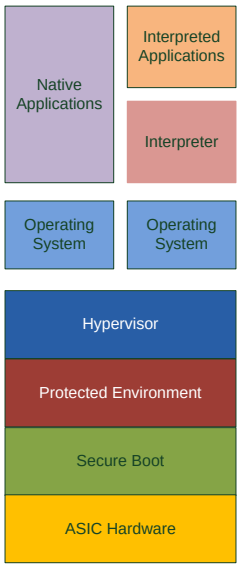


Figure 3 – Architecture with Hypervisor and Protected Environment

Figure 3 illustrates a combined architecture where a hypervisor is run above the Protected Environment. This requires a combination of the features of the hypervisor (Figure 2) and hardware isolated implementations (Figure 1).

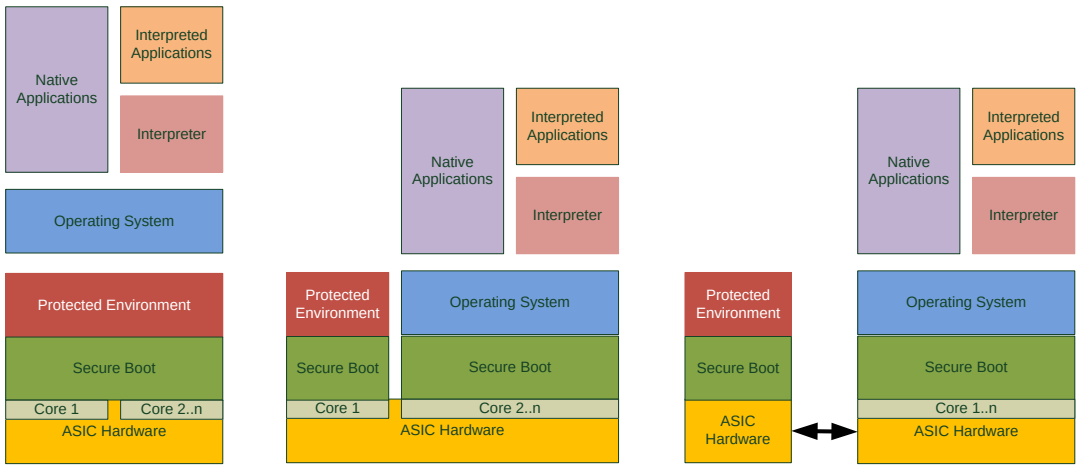


Figure 4 - Multi-core and multi processor architectures

Three models using multiple processor cores are shown in Figure 4:

- The code running on each processor core could be individually securely booted and be associated with a core-specific Protected Environment.
- One or more processor cores could provide a secure, isolated environment for the Protected Environment while the other cores run the Rich OS.
- It is also possible to place the Protected Environment in a separate ASIC to achieve isolation.

The TPM Mobile Reference Architecture supports all of these implementation choices for the Protected Environment and all can support the implementation of a TPM Mobile.

Note that the same ASIC can support all the architectures described in this section with no hardware changes. The only difference between these architectures is the software running on the ASIC.

5 Device Boot Sequences - INFORMATIVE

This section describes the various stages that can form the boot sequence of any device, but with particular relevance to those devices where support for the TPM is not included in hardware (that is, where there is no fixed location in device memory at which a TPM can be accessed).

Mobile devices incorporate some form of Secure Boot sequence to establish the secure foundation upon which subsequent secure device operations can be built.

Many existing standards (for example in the areas of radio standards, payments, etc.) require that certain device values are stored and used securely, and that a mechanism is provided to ensure that devices boot in a way that minimizes the possibility that these important values are used incorrectly or replaced. In this specification, this mechanism is called Secure Boot.

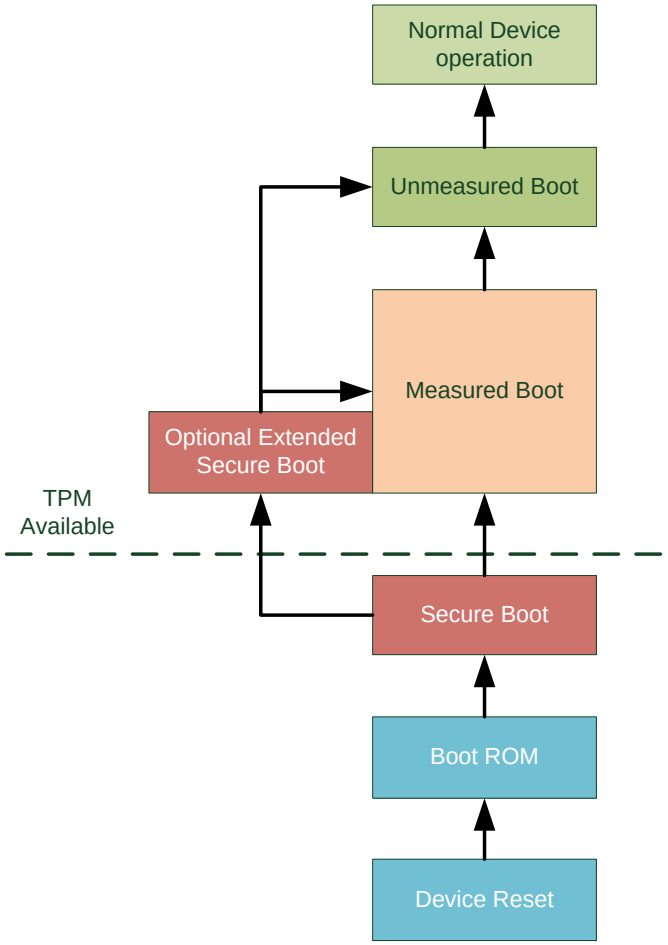


Figure 5 – Sequence of possible boot phases of a device

The possible boot phases are illustrated in Figure 5.

5.1 Device Reset

At device reset, hardware initialization occurs. No software or firmware is executed at this time.

5.2 Boot ROM

Once the hardware is initialized, it starts to execute software that is stored in an area of Read-Only Memory (ROM) that is built into the ASIC containing the processor which is booting. The attack cost of modifying on-chip ROM is considerable and falls outside the bounds of most commercial threat models and security requirements.

The software in the Boot ROM performs further hardware initialization and checks the integrity of secrets stored in write-once memory (such as Fuses). These secrets support higher-level software such as the Root of Trust for Storage (RTS) and the Protected Environment (if implemented in this ASIC) since they include unique keys for this device.

The Boot ROM acts as the hardware Root of Trust for Verification (RTV) and forms the root of a transitive chain of verification. If any of the roots of trust are updatable, the Boot ROM acts as the Root of Trust for Update (RTU).

The software in the Boot ROM locates the first boot phase software in some form of non-volatile storage (unless the system executes wholly from ROM). The Boot ROM could then perform the following sequence of operations:

- Measures the located image and determines the validity of the measurement using an expected value based on keys and other data that are stored or derived from values stored in write-once memory. An example could be verification of the signature on the image using a key stored in non-volatile write-once storage (such as Fuses).

- Checks that the version number stored within the image is valid based on values stored in write-once memory. For example, a check that the version number is greater than or equal to a value stored in non-volatile storage such as Fuses.

- If all checks have passed, then the device executes the loaded software. If any checks fail, then the device can enter a remediation mode implemented in the boot ROM, or return to the reset state.

5.3 Secure Boot

The most widely understood and implemented boot mechanism on mobile devices is Secure Boot.

In Secure Boot, each module of code acts as a proxy for the Root of Trust for Verification to form a transitive chain of verification tracing back to the Boot ROM (Section 5.2). This phase could also be referred to as Verified Boot, but the term Secure Boot is widely used in the Mobile industry; therefore this specification uses the term Secure Boot.

The Secure Boot process is such that whenever an additional module of firmware code (or a data file in some cases) is loaded, the following steps are executed:

- The code module is measured.

- The validity of the measurement is determined based on information stored either in write-once memory or in previously verified code modules.

- The version number stored in the code module is validated to ensure that it is the expected one.

- If a suitable storage mechanism such as a TPM or a Secure Boot PCR Store (Section 5.3.2) is available, the measurement is extended into that storage mechanism, otherwise the measurement is discarded.

- If any of these checks fail, the software enters a remediation mode defined by a previously validated code module, or returns to the reset state.

If the Boot ROM (5.2) is immutable and trustworthy, code that fails validation cannot be launched during the Secure Boot process.

Secure Boot is usually used only for the early-booted device firmware up to the point of launching the main operating system (OS) boot loader. The trustworthy OS loader, for example UEFI (Unified Extensible Firmware Interface) [3], can use either Secure Boot or Measured Boot as needed.

The Secure Boot phase can

- connect to an external TPM, or
- create the Protected Environment within which it executes the TPM as a Trusted Application.

5.3.1 UEFI Secure Boot

Many platforms make use of a mechanism known as UEFI [3] to perform their boot sequence.

UEFI includes a mechanism known as UEFI Secure Boot [3].

If the feature is enabled, the UEFI Secure Boot mechanism launches only drivers and images that are signed by a known key. If the resultant signature verification fails, the UEFI firmware initiates OEM-specific recovery to restore trusted firmware.

This mechanism is very similar to the Secure Boot mechanism described in this document; it has similar security properties. Typically, Secure Boot validates the UEFI executables and then passes control to UEFI Secure Boot. This gives the Rich OS a secure platform upon which to build.

5.3.2 Secure Boot PCR store

The mechanism defined in this section is OPTIONAL.

TPM functionality may not be available during the early phases of Secure Boot because the TPM or its communications mechanism is not yet available.

A lower functionality alternative can be implemented on some devices that makes the Secure Boot process more verifiable.

The Secure Boot PCR Store (SBPS) makes use of additional processing resources within the ASIC (such as a processor that is not otherwise used in the boot process) to securely maintain a single PCR-like value; that is, it acts as a Root of Trust for Integrity (RTI).

If an SBPS is implemented:

1. The SBPS MAY be implemented as a hardware peripheral.
2. If the SBPS is implemented in software code then that code SHALL be protected to the level defined in Section 6.1. The code of the SBPS could for example be located in the Boot ROM or in the initial firmware loaded by the Boot ROM.

3. The SBPS SHALL store a single PCR-like value of an appropriate strength as defined by Section 6.1.

4. The SBPS SHALL provide a mechanism to extend measurements to the stored value.

5. Code executing during the Secure Boot phase SHALL extend every generated measurement into the value stored by the SBPS.

6. The SBPS SHALL provide a mechanism to retrieve the stored value.

7. Once the TPM Mobile is available the SBPS MAY be destroyed after its value has been extended into PCR0 in the TPM.

5.4 Measured Boot

Measured Boot is the process followed once a TPM is available during the boot process.

A previously loaded module measures each module of code or data.

The measurement is then extended into one or more PCRs within the TPM (where the measurements are stored).

The code module is then executed or the data is used regardless of the value of the measurements.

Once measurements are available in the TPM, it becomes possible to perform binding or attestation, as defined by the TPM 2.0 Library Specification [1].

6 The Protected Environment

A Protected Environment uses platform resources to provide an isolated execution environment. The isolated execution property of the Protected Environment distinguishes the Protected Environment from the Rich OS environment.

A device can implement one or more Protected Environments using any mechanism that meets the requirements in this section. This includes implementations where the Protected Environment is located in a separate ASIC, or on a separate processor within the same ASIC, or in a special mode of the main processor.

Protected Environments execute Trusted Applications that are secured by the Protected Environment from interference from other Trusted Applications and from code executing outside the Protected Environment.

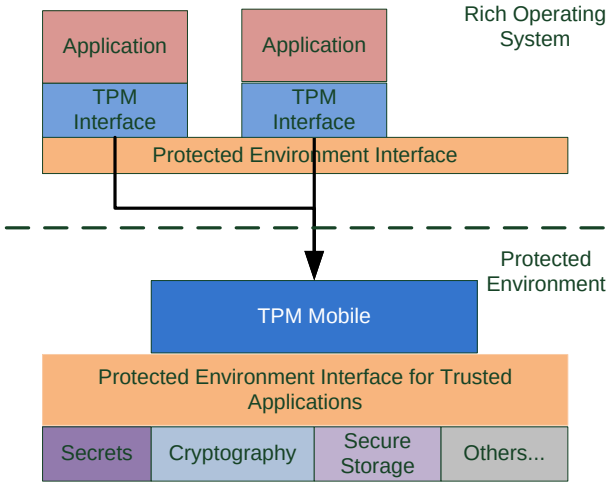


Figure 6 - Architecture of a Protected Environment running a TPM Mobile

All implementations of trusted services in a Protected Environment such as storage, measurement and verification depend on the underlying Roots of Trust.

The following sub-sections define the properties of the Protected Environment. The properties, rather than a particular implementation, define the Protected Environment. The implementation could be of any form, as long as it matches the requirements.

6.1 Boot Time Requirements

At boot time, the device SHALL implement the Device Reset (Section 5.1), Boot ROM (Section 5.2), Secure Boot (Section 5.3), Measured Boot (Section 5.4), and Normal Device Operation phases.

Once the TPM is instantiated, the device MAY continue the Secure Boot phase (Section 5.3) and then the measurements are passed to Measured Boot.

6.2 Integrity level Requirements

The integrity of the Protected Environment and its associated resources MAY be protected in many ways and this specification does not constrain the implementation to any particular solution.

The level of integrity protection relies on the type of device implementing the Protected Environment and the cryptographic strength [8] of the trusted services that depend on the underlying Roots of Trust.

Where parts of the Protected Environment are protected by cryptographic means, the protection provided SHALL match or exceed the greater of the following:

- An equivalent cryptographic strength [8] of 112 bits [9].
- The largest cryptographic strength [8] by which any Trusted Application (including the TPM) executing within the Protected Environment stores or manipulates.

Where the integrity of Protected Environment resources relies on non-cryptographic means (such as the use of ROM, physical separation, etc.), the supplied protection SHALL be sufficient to defeat all software attacks and at least simple hardware attacks.

The resources within the system that SHALL be protected to the security level defined above and can include the following:

- Hardware Roots of Trust.
- Protected Environment integrity.
- Protected Environment-provided non-volatile storage.
- Protected Environment non-volatile storage rollback: all unauthorized attempts to modify current data or re-instate old data SHALL be detected.

6.3 Protected Environment Requirements

1. All Protected Environment capabilities SHALL be protected by a transitive chain of trust from the Roots of Trust on the processor where the Protected Environment executes.
2. Trusted Applications instantiated in the Protected Environment SHALL be protected by a transitive chain of trust from the Roots of Trust on the processor where the Protected Environment executes.
3. The Protected Environment SHALL provide execution resources for Trusted Applications that are isolated from entities external to the Protected Environment with a level of protection as defined by Section 6.2.
4. The Protected Environment SHALL be protected from all other execution environments with a level of protection as defined by Section 6.1.
5. The integrity of the Protected Environment SHALL be protected to the level defined by Section 6.1.
6. The Protected Environment SHALL verify the integrity and authenticity of each Trusted Application executable image before execution with a level of assurance as defined by Section 6.1.
7. The Protected Environment SHALL provide evidence identifying every device firmware and software component that is used to support and implement the Protected Environment. This evidence SHALL be sufficient to identify at least the manufacturer and the version of each component (see Section 8.6).
8. The Protected Environment SHALL provide Trusted Applications access to non-volatile storage locations protected to the level defined in Section 6.1.

9. The Protected Environment SHALL ensure the integrity and confidentiality of its non-volatile storage locations to the level defined in Section 6.1.

10. The Protected Environment SHALL protect information stored in the non-volatile storage locations provided to a Trusted Application, whether cryptographically or by other means, such that only the owning Trusted Application and the Protected Environment can access the information.

11. The Protected Environment SHALL prevent replay or rollback of the provided non-volatile storage locations.

12. The Protected Environment SHALL report error messages to the accessing Trusted Application when attempts to access data in its protected storage locations fail.

13. The Protected Environment SHALL ensure that any provided cryptographic functions have the same isolation properties as described in Section 6.1.

14. The Protected Environment SHALL provide Trusted Applications with access to an entropy source that complies with IETF RFC 4086 [10]and ISO/IEC 18031:2011 [11].

15. The Protected Environment SHALL provide Trusted Applications with a trusted source of time of day information that meets the requirements of the TPM 2.0 Library Specification [1].

16. If the Protected Environment provides debug facilities (for example, for troubleshooting live devices) these facilities SHALL NOT allow information stored either in the run time memory of Trusted Applications or that stored in non-volatile storage locations by Trusted Applications to be divulged.

17. The Protected Environment SHALL ensure that modified versions of the Protected Environment or any of the software that supports the Protected Environment are verified and authenticated to the level defined in Section 6.1 before such modified versions are installed.

18. The Protected Environment SHALL ensure that new or modified versions of Trusted Applications are verified and authenticated to the level defined in Section 6.1 before they are installed.

19. The Protected Environment SHALL prevent rollback of software implementing the Protected Environment and Trusted Applications to older versions.

20. The Protected Environment SHOULD provide the capability to run multiple Trusted Applications concurrently. If multiple Trusted Applications can be run concurrently:

    a. The Protected Environment SHALL provide isolation between these Trusted Applications to the level specified in Section 6.1.

    b. The Protected Environment SHOULD provide a mechanism to enable communication between Trusted Applications that is secured to the level defined in Section 6.1.

    c. The Protected Environment SHOULD provide a reliable indication of whether communications to a Trusted Application originate from within or outside the Protected Environment.

    d. The Protected Environment SHOULD provide a reliable indication of the source Trusted Application when communicating between Trusted Applications.

6.4 Example Protected Environment Implementation - INFORMATIVE

An example implementation of a Protected Environment is a Trusted Execution Environment (TEE) as defined by GlobalPlatform [2].

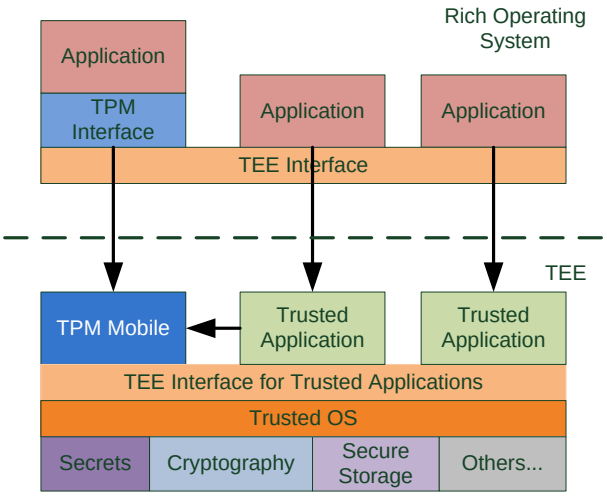


Figure 7 - TPM Mobile implemented using a GlobalPlatform TEE as the Protected Environment

A GlobalPlatform compliant TEE that:

- is certified to a suitable level of protection as defined in 6.1
- includes the anti-rollback features

should be capable of meeting all the Protected Environment requirements.

Such a TEE can therefore host a TPM Mobile. Figure 7 illustrates such an implementation. GlobalPlatform standards [2] define all the interfaces apart from the TPM Interface that the TCG defines.

This specification does not require that the Protected Environment in which the TPM Mobile executes is compliant with GlobalPlatform standards. Commercial products such as Trusted Operating Systems could be available that already implement the standards.

7 TPM Mobile Implementation

This section describes the details of a TPM Mobile implementation and its relationship to the Protected Environment and other entities.

7.1 Use of Protected Environment Capabilities

The TPM Mobile SHOULD use the capabilities provided by the Protected Environment as defined in Section 6.

- The TPM Mobile SHOULD execute as a Trusted Application within a Protected Environment.
- The TPM Mobile SHOULD use the non-volatile storage mechanism defined by the Protected Environment to store its non-volatile data.
- If the Protected Environment provides cryptographic mechanisms then the TPM Mobile SHOULD make use of these where appropriate.
- The TPM Mobile SHOULD accept commands from the Rich OS using the communications mechanism provided by the Protected Environment.
- A TPM Mobile SHOULD protect the separation and integrity of TPM resources, NV storage, and other TPM assets owned by any specific application (in the Protected Environment, in the Rich OS, or elsewhere) from disclosure to or modification by any other application.
- The TPM Mobile SHOULD use the entropy source provided by the Protected Environment for random number generation.
- The TPM Mobile SHOULD use the time of day information provided by the Protected Environment.

7.2 Application Programming Interfaces

The TPM Mobile SHOULD use the interface provided by the Protected Environment to receive commands and return responses.

- The TPM Mobile SHOULD implement one or more of the application programming interfaces defined by TCG for use by applications running in the Rich OS, such as the TPM 2.0 Mobile Command Response Buffer Interface [13]

7.3 Command Profiles

This architecture supports TPM Mobile implementing the TPM 2.0 Library Specification [1] and any platform profile(s) of that specification which TCG MAY publish in the future. Devices that are compatible at an application level to those running on PCs SHALL implement the profile defined in [12]. Embedded devices using other profiles are also supported.

A TPM Mobile SHALL implement at least one TCG-defined profile.

7.4 Roots of Trust

The architecture defined in this specification, whereby the TPM Mobile executes within a Protected Environment that is logically isolated from the Rich OS, has intrinsic consequences for the system's Roots of Trust.

One consequence of this architecture is that many of the Roots of Trust are verifiable, for example, by the RTV during Secure Boot. Something can be considered a Root of Trust only if it is unverifiable; that is, it is intrinsically trusted. Whether or not something is technically a Root of Trust, the component that implements that function is for practicality referred to as a Root of Trust.

Since it is isolated from the Rich OS, the Protected Environment MAY use a separate set of Roots of Trust to ensure its correctness and security.

Root Of Trust	Protected Environment	Rich OS
Root of Trust For Measurement (RTM)	Yes	Yes
Root of Trust For Confidentiality (RTC)	Yes	No, not required
Root of Trust For Integrity (RTI)	Yes	No, not required
Root of Trust For Verification (RTV)	Yes	Yes
Root of Trust for Reporting (RTR)	Yes, in the TPM Mobile implementation	No, not required
Root of Trust for Update (RTU)	Yes	Yes

7-1 Roots of Trust and where they are used

In some implementations, these Roots of Trust MAY be shared (for example Annex A) but in many cases at least some of them are discrete (for example Annex B and Annex C).

All of the Roots of Trust (apart from the RTR) SHOULD be provided by the platform and SHOULD be based on trust of the device Boot ROM and non-volatile write-once storage. In some cases Roots of Trust MAY be based on certification of the platform supporting the Protected Environment (for example by use of a Secure Element as in Section C.6).

The RTM is located in two locations: in the TPM initialization code and in the Rich OS code that extends the PCRs once the TPM is available. The Protected Environment RTM provides information about the firmware running on the processor that supports the Protected Environment. The Rich OS RTM extends PCR 0 with information about the firmware running on the processor that supports the Rich OS environment.

The RTV and RTU can be located in one or two locations; this depends solely on whether the Protected Environment executes on the same or a different processor to the Rich OS.

The executable code that forms the RTC is located within the Protected Environment implementation and it is verified by the chain of trust that traces back to the RTV. The RTC makes use of device-specific values that are stored in non-volatile write-once memory when the device is manufactured. These device-specific values are used to derive the keys used to protect the non-volatile storage provided by the Protected Environment.

8 TPM Mobile Identity and Ownership

8.1 TPM Mobile Installation

Typically, the Device Manufacturer installs the TPM Mobile. In some cases, where the platform includes a suitable Protected Environment that can install additional Trusted Applications, the TPM Mobile MAY be installed at a later point by a suitable authority. The authority that installs the TPM could also be a Mobile Network Operator or an Enterprise.

The TPM Installing Authority is responsible for the management of TPM Mobile updates. Updates MAY make use of the TPM 2.0 Library Specification [1] field update mode but could also use other mechanisms associated with the Protected Environment.

8.2 Platform Hierarchy Ownership

The TPM Installing Authority SHALL own the platform hierarchy of the TPM Mobile if it is implemented.

The TPM Installing Authority specifies the hierarchy enable, authValue and authPolicy for the platform hierarchy. The values of the hierarchy enable, authValue and authPolicy required by the TPM Installing Authority are established when the TPM Mobile is initialized, as defined in Section 8.5.

If the TPM Installing Authority does not wish to use the platform hierarchy, it can be disabled by clearing the hierarchy enable.

8.3 TPM Mobile Ownership

The Device Owner owns the TPM Mobile.

In typical use, device ownership is pre-configured at the time of device manufacture and is modifiable only by tools supplied by the TPM Installing Authority. If Endorsement Key certificates are required during device manufacture, they SHALL be signed by the TPM Installing Authority.

8.4 Low Power Modes

The TPM 2.0 Library Specification [1] specifies a system with a very simple low power model consisting essentially of ON and OFF modes. The TPM Mobile SHOULD preserve the TPM Mobile state (resources, NV storage, keys) across all low power transitions (except OFF).

Mobile platforms support much richer low power models that differ considerably in functionality from those defined by the TPM 2.0 Library Specification [1].

The common feature of Mobile Platform low power models is that application processes, often including Trusted Applications running in the Protected Environment, are by design wholly unaware of the power state of the system.

The state of applications is maintained by the system through any low power transitions.

Consequently, the TPM Mobile SHALL implement the same low power model as the overall system. Applications have an arbitrary set of state active in the TPM Mobile at the point that a low power transition occurs and expect state to be preserved over the transition.

TPM Mobile need not be aware of low power transitions of the platform.

8.5 TPM Mobile Startup

The TPM Mobile SHOULD be started only when the device is booted. There is often no equivalent to the TPM 2.0 Library [1] TPM restart and TPM resume operations. All TPM Mobile start ups SHOULD be equivalent to TPM reset if TPM restart and TPM resume are not supported.

Many mobile devices do not have an equivalent of the PC BIOS that, in the TPM 2.0 Library Specification [1], is assumed to take ownership of the TPM's platform hierarchy as soon as the device starts up. If the platform hierarchy is implemented, a mechanism is needed that puts the platform hierarchy into a secure state as soon as it is started, to avoid the platform hierarchy being left in an insecure state.

If the TPM2_Startup command is not implemented, the TPM Mobile SHALL perform a set of initialization actions before executing the first received command from any application. This is equivalent to receiving the _TPM2_Init and TPM2_Startup messages, as defined in the TPM 2.0 Library Specification [1].

If the TPM2_Startup command is not implemented the initialization actions perform the following functions:

- The TPM Mobile SHALL check the integrity of the resources that it uses. This can include data stored in Protected Environment-provided non-volatile storage that is used by the TPM Mobile and other resources.
- The TPM Mobile SHALL perform self tests on all implemented cryptographic algorithms using the mechanisms defined by the TPM 2.0 Library Specification [1], unless the Protected Environment has already performed such self tests.
- If the platform hierarchy is implemented the TPM Mobile SHALL set the hierarchy enable, authValue and authPolicy (see TPM 2.0 Library Specification [1]) for the platform hierarchy as specified by the TPM Installing Authority). See Section 8.2 for details.
- The TPM Mobile SHALL set PCR 0 to 0.
- To identify the firmware supporting the protected environment, the version information in the TPM capabilities TPM_PT_VERSION_NUMBER_1 and TPM_PT_VERSION_NUMBER_2 SHALL be set as defined in Section 8.6.

If the TPM2_Startup command is implemented, it behaves as defined in the TPM 2.0 Library Specification [1]. The PCRs SHALL be set as defined in Section 8.6.

8.5.1 TPM2_Shutdown

There are circumstances when the system could know that a system shutdown is imminent (for example when there is an extremely low battery indication).

The TPM Mobile SHALL support the TPM2_Shutdown command to force any pending non-volatile memory updates to be written to secure storage. This ensures that intended device behaviour continues when the TPM reboots.

8.6 Platform Baseline Measurements

The validity of the firmware executed on the platform before the TPM Mobile starts is of vital importance to the integrity of the whole platform.

There are several relevant sets of firmware identifying information:

- The firmware identification information that supports the Protected Environment: This is written to the TPM_PT_VERSION_NUMBER_1 and TPM_PT_VERSION_NUMBER_2 properties in the TPM. This value is set by the TPM Mobile initialization code. For example, this could be a truncated hash.

- The identification information of the boot code and firmware that loads the Rich OS: This is extended into PCR 0 by the RTM in the Rich OS that executes as soon as a connection to the TPM Mobile is available. The TPM Mobile initialization code sets PCR 0 to 0.

The TPM Installing Authority SHALL make information available that defines the valid values for PCR 0 and the version number properties. This can be achieved via certificates signing the values or by any other mechanism that allows the capability and PCR values to be verified.

A. Protected Environment using a hardware isolation mechanism - INFORMATIVE

In this model (shown in Figure 8) the Protected Environment and the Rich OS share the same processors. The separation of the Protected Environment from the Rich OS is achieved by a hardware isolation mechanism that meets the requirements defined in Section 6.

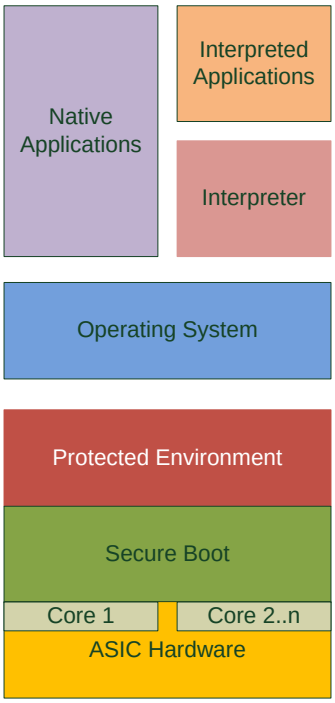


Figure 8 – Protected Environment using hardware isolation

Implementation options are:

- A logically separate trusted operating system running within the Protected Environment that hosts a Trusted Application implementing the TPM Mobile
- The TPM is an integral part of the Protected Environment.

In this model the Boot ROM, boot sequence, RTV and RTU are shared by the Protected Environment and the Rich OS.

A.1 Communications Mechanism

Communications with the TPM Mobile makes use of the mechanism provided by the Protected Environment. This mechanism is typically based on shared memory.

The mechanism used to communicate with Trusted Applications executing in the Protected Environment may not be fully available until the Rich OS has completed its boot sequence. During the Rich OS boot sequence an alternative, lower functionality, communications mechanism may be used. The state of the TPM Mobile must be preserved over the transition to the full functionality mechanism.

A.2 Boot Sequence

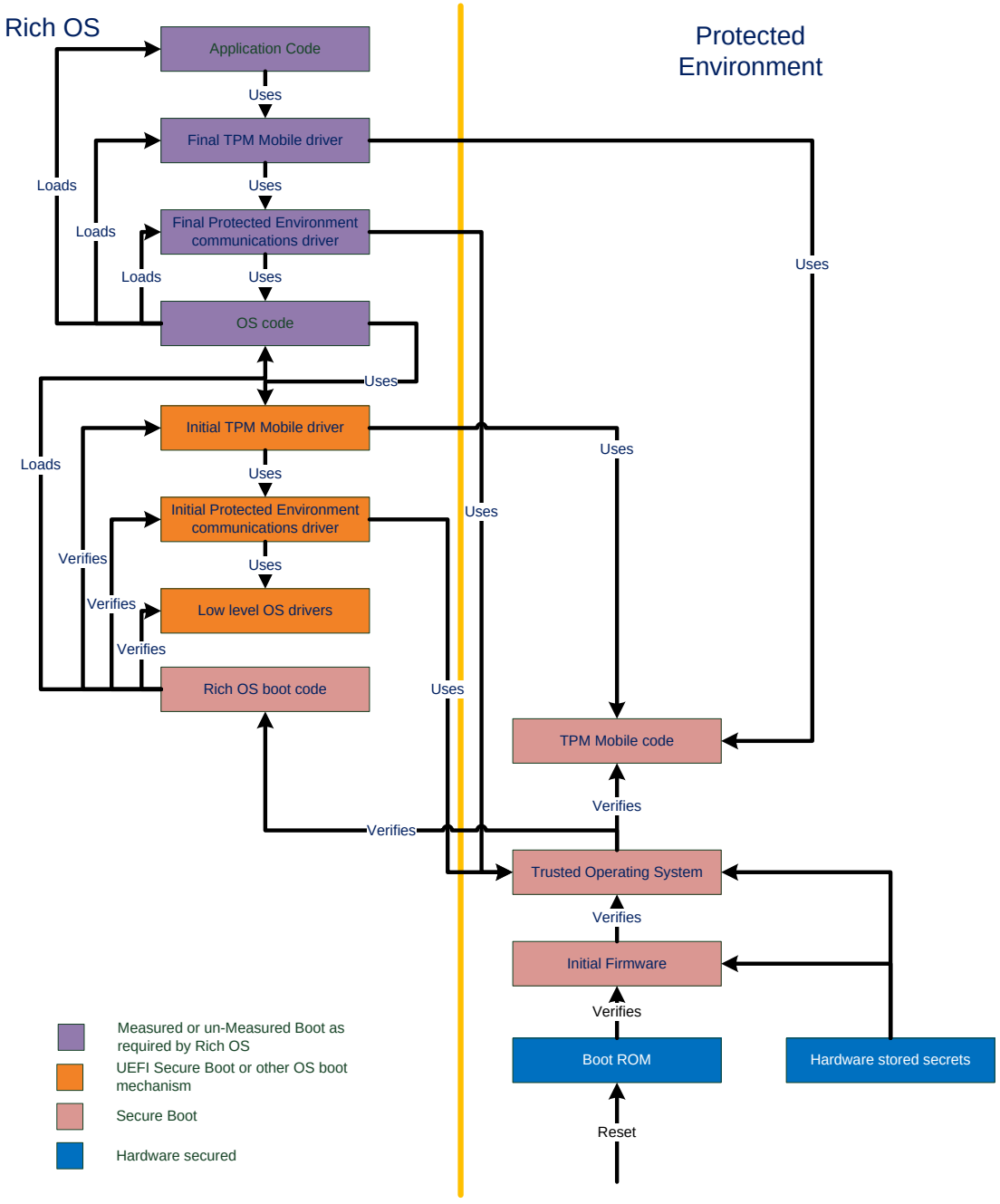


Figure 9 – A possible boot process when a hardware isolation mechanism is in use

There are many options for how to implement the boot sequence of a system where the Protected Environment and the Rich OS share the same set of processors. Figure 9 shows one possible boot sequence.

Typically in the boot sequence, the Protected Environment initializes first since it can complete its boot sequence without interference from code executing in the Rich OS. The Protected Environment also reserves the resources it requires, leaving the remainder for use by the Rich OS.

When the Protected Environment initialization is completed, the Rich OS boot process starts. The Rich OS boot procedure is the same as that in an isolated ASIC, except that a different communication mechanism is used with the TPM Mobile.

B. Protected Environment on a separate core in same ASIC - INFORMATIVE

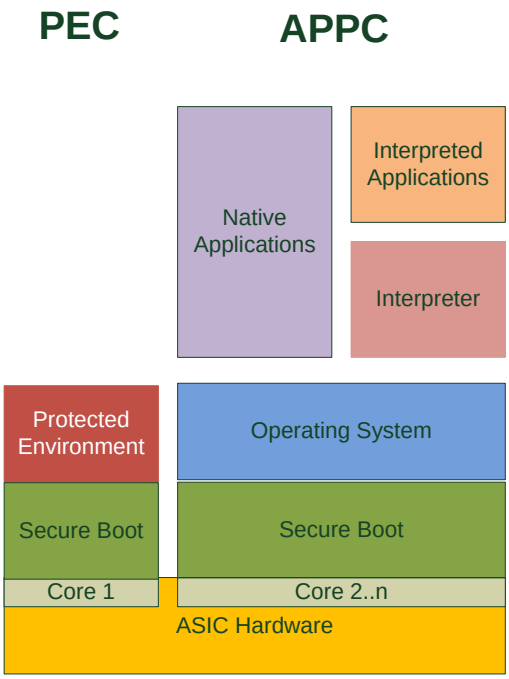


Figure 10 - Protected Environment in separate core in same ASIC

In this model, shown in Figure 10, the Protected Environment executes on a separate core (the Protected Environment Core (PEC)) within the same ASIC as the core on which the Rich OS operates (the Application Core (APPC)).

The security of this model depends critically on the degree of integration between the PEC and the APPC.

Implementation options include:

- One of a cluster of identical cores acts as the PEC. The PEC then shares cache, memory and peripherals with the APPC. A hardware isolation mechanism provides sufficient separation between the PEC and the APPC to meet the requirements in Section 6. The boot sequence of the PEC and the APPC are shared as in Annex A.

- A core that may already perform some other task within the device acts as the PEC. This core could be responsible for implementing power control, wireless connectivity or any other function. The boot sequence and Roots of Trust could be strongly linked between the PEC and the APPC, or could be independent. Typically access to shared peripherals, such as non-volatile storage, is via the APPC rather than the PEC.

The requirements for isolation between the Protected Environment and other entities, as defined in Section 6, also apply for isolating Trusted Applications running in the Protected Environment from any firmware or software implementing additional functions on the PEC.

If the PEC does all of the following:

- Boots independently of the APPC

- Does not share memory or cache with the APPC, apart from a mailbox
- May share non-volatile storage with the APPC
- Does not share peripherals with the APPC

then this model is the equivalent of implementing the Protected Environment in a separate external ASIC as described in Annex C.

If the PEC meets any of the following conditions:

- Interacts with the boot process of the APPC in some way other than possibly initiating it
- Shares memory or cache with the APPC even if separated by a hardware isolation mechanism
- Shares peripherals with the APPC

then the model defined in this section applies.

B.1 Communications Mechanism

If the PEC is in the same cluster of cores as the APPC, communication is likely to be via shared memory and some interrupt scheme.

If the PEC is on a loosely coupled core, it can use any communications mechanism that meets the requirements in Section 6. This communications mechanism could involve shared memory, a mailbox register, or potentially a full DMA (Direct Memory Access) link.

B.2 Boot Sequence

If the PEC is in the same cluster of cores as the APPC, the boot sequence is likely to be similar to that described in Section A.2, with the sole difference being that the Protected Environment runs on a separate core. The PEC and the APPC often share the same Boot ROM.

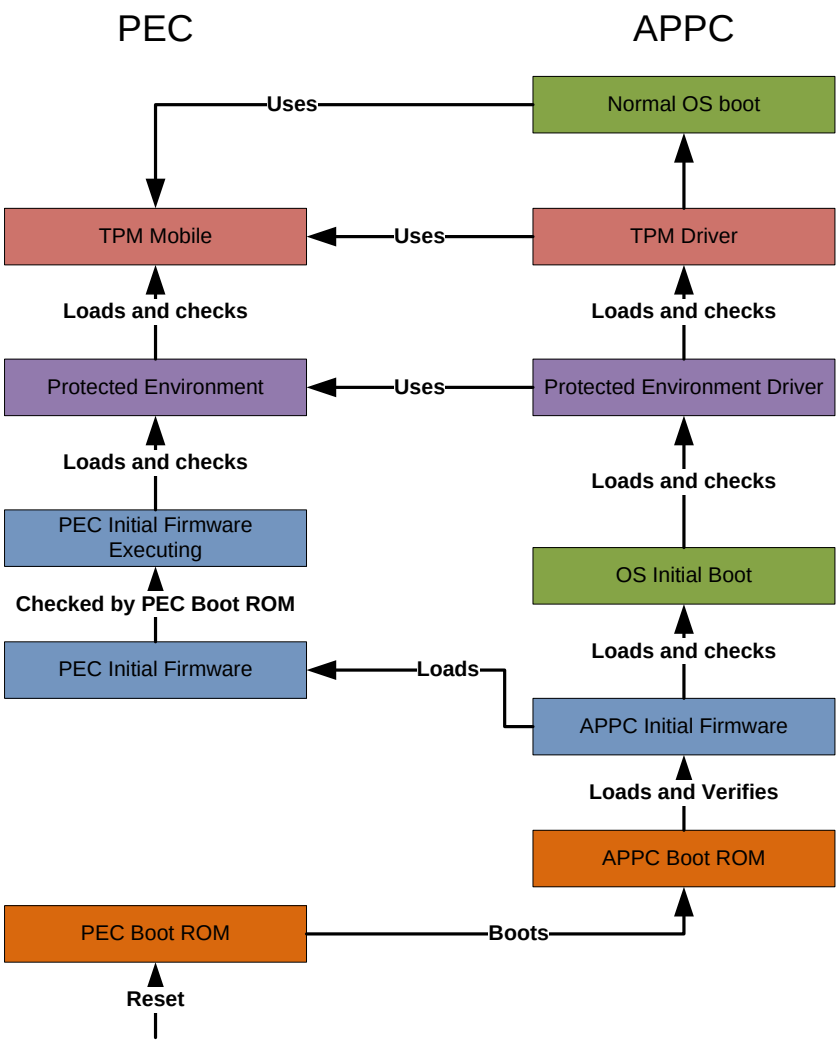


Figure 11 – Possible Boot sequence where PEC is also the power controller for the APPC

If the PEC resides on a separate core from the APPC, the boot sequence is likely to be complex. The PEC may have to boot to a certain state before the APPC starts booting. The PEC may not be able to complete Protected Environment initialization until the APPC is able to reference non-volatile storage; the PEC's final boot phase may have to wait until the TPM Mobile is used during Rich OS boot. Figure 11 is an example of how complex the boot sequence can be.

B.3 Roots of Trust

If the PEC is in the same cluster of cores as the APPC, the Boot ROM, boot sequence, RTV and RTU are shared by the Protected Environment and the Rich OS.

In more complex cases some Roots of Trust may be shared, while others may not.

C. Protected Environment in a separate ASIC - INFORMATIVE

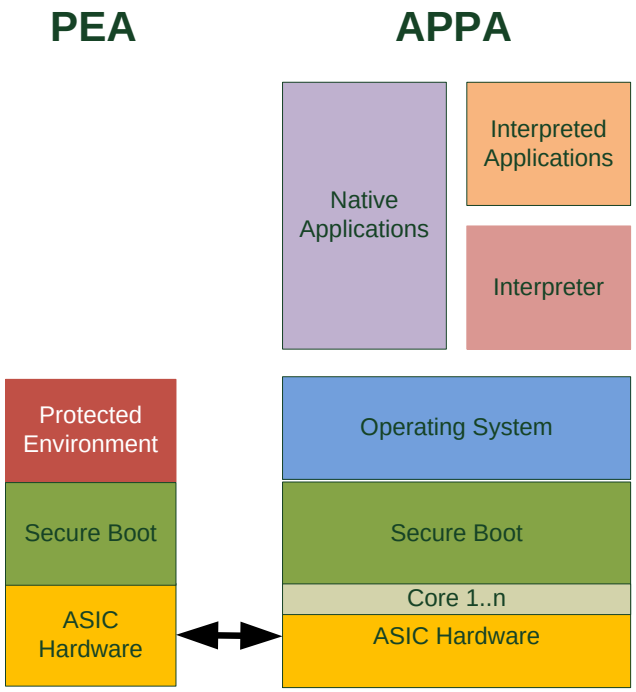


Figure 12 – Protected Environment in a separate ASIC

Figure 12 shows this model in a schematic form. In this model the Protected Environment is not located in the same ASIC as the application processors; a communications link is necessary between the Applications ASIC (APPA) and the Protected Environment ASIC (PEA).

The presence of the PEA implies a different boot procedure since the boot process of the two ASICs is independent. The only time when the boot states of the APPA and PEA interact is when the Rich OS running on the APPA connects to the TPM Mobile running in the Protected Environment.

C.1 Protected Environment ASIC

The PEA could perform additional functions (for example power control or modem functionality) in addition to facilitating the Protected Environment. Trusted Applications running in the Protected Environment are isolated from the firmware or software implementing these additional functions to the level defined in Section 6.2

A second TPM Mobile instance could be instantiated for use by the additional features running on the PEA. The PEA could possibly host a different Rich OS with its own TPM Mobile in addition to that supporting the APPA; that is, the PEA implements a system as defined in annex A.

The PEA typically includes non-volatile storage within the ASIC; if it uses external non-volatile storage, this is typically independent of that used by the APPA.

The PEA boots independently of the APPA. When the PEA boots is implementation defined. If for example the additional functions of the PEA control the power state of the APPA, the PEA must obviously boot first.

The latest point at which the PEA can boot is when the APPA connects to the TPM Mobile running in the Protected Environment.

C.2 Applications ASIC

The APPA is a standard application processor chip that runs a Rich OS or in some cases an embedded OS (for example, in a Smart Meter using a Secure Element [14] as the PEA).

The APPA boots using a standard boot sequence (Section 5). The Secure Boot phase (Section 5.3) connects to the TPM Mobile running in the PEA.

C.3 Communications Mechanism

The communications mechanism between the APPA and the PEA is implementation dependent. The only requirement is that it transports messages from the APPA to the TPM Mobile running on the PEA and responses from the TPM Mobile to the APPA.

C.4 Boot Sequence

In this model, the boot sequences of the APPA and the PEA are wholly separate.

If the PEA does not support a Rich OS, a simple boot sequence is sufficient. The PEA is likely to solely use a Boot ROM (Section 5.2) and Secure Boot (Section 5.3); there is no need for any other boot phases. For best security the PEA ideally boots from internal non-volatile memory.

The APPA follows a standard boot sequence as defined in Section 5, using the TPM Mobile implemented within the Protected Environment in the PEA once it is available.

C.5 Roots of Trust

C.5.1 PEA Roots of Trust

The PEA implements the following Roots of Trust:

- Root of Trust for Confidentiality – The RTC is located within the code of the Protected Environment and the TPM Mobile running within it. If the protected storage is within the ASIC, further protection may not be required. If the protected storage is external to the ASIC, values in non-volatile storage determine the key used when encrypting non-volatile values.

- Root of Trust for Integrity – The RTI is not required on the PEA unless it implements an SBPS (Section 5.3.2) and when the SBPS acts as an RTI

- Root of Trust for Measurement – The RTM is located within the TPM Mobile initialization code that executes within the Protected Environment. Its code is verified by the transitive chain of verification tracing back to the RTV in the Boot ROM.

- Root of Trust for Reporting – The RTR is located within the TPM Mobile code and it depends on the protected storage provided by the RTC. It only appears once the TPM Mobile is running. It is not used on the PEA.

- Root of Trust for Update – The RTU is located in the Boot ROM of the PEA along with non-volatile write-once storage within the ASIC. The RTU validates firmware updates.

- Root of Trust for Verification – The RTV is located within the Boot ROM of the PEA along with some non-volatile write-once storage within the PEA. Based on this and using the Secure Boot mechanism (Section 5.3), a chain of verification then extends up to the implementation of the Protected Environment and beyond into the Trusted Applications running in the Protected Environment.

C.5.2 APPA Roots Of Trust

The APPA implements the following Roots of Trust:

- Root of Trust for Confidentiality – The RTC is provided by the TPM Mobile.

- Root of Trust for Integrity – If the PEA implements an SBPS (Section 5.3.2), the SBPS acts as an RTI until the TPM Mobile becomes available. The TPM Mobile acts as the RTI for the APPA once it is available.

- Root of Trust for Measurement – The RTM is located within the Rich OS code at the point where the connection to the TPM Mobile becomes available. Its code is verified by the chain of trust leading back to the RTV.

- Root of Trust for Reporting – The RTR is provided by the TPM Mobile once it becomes available.

- Root of Trust for Update – The RTU is located in the Boot ROM of the APPA along with non-volatile write-once storage within the ASIC. The RTU validates firmware updates.

- Root of Trust for Verification – The RTV is located within the Boot ROM of the APPA along with some non-volatile write-once storage within the ASIC. A chain of verification based on this then extends up to at least the point where the TPM Mobile becomes available to the Rich OS.

C.6 Example using a Secure Element to host the Protected Environment

A Secure Element (SE) is an example of an external ASIC that hosts a Protected Environment.

Secure Elements are highly secure ASICs designed to maintain high value assets (for example, financial secrets in credit cards, access secrets for Mobile Networks) in mobile devices, even in hostile environments.

The electrical and mechanical interfaces and security mechanisms of Secure Elements are defined by a number of ISO [15], ETSI [16] and GlobalPlatform [14] standards. Common Criteria certification [18] and compliance testing [19] are required. Secure Elements are protected against both hardware and software attacks.

A set of standards from GlobalPlatform [17] define an execution environment within an SE where it executes programs written in a subset of Java. Secure Elements implementing these standards are known as JavaCards.

The capabilities and protection level offered by JavaCards exceed the requirements for a Protected Environment as defined in Section 6. Secure storage is provided within the SE but is limited by the available resources. The amount of memory available for programs varies. A TPM Mobile executing in such a Protected Environment must be written in Java.

The JavaCard environment is tightly controlled. All management is performed by Trusted Service Managers (TSM) that are remote entities. The TSM creates a secure VPN using a shared key provisioned at the time of device manufacture to send the SE management commands; the SE accepts management commands only over this VPN.

The external interface to an SE is based on the exchange of Application Protocol Data Unit (APDU) messages with a maximum length of 256 bytes and it is necessary to map the TPM 2.0 Library Specification interface [1] on top of this mechanism.

No details of the SE boot sequence are available. Common Criteria certification [18] and compliance testing [19] ensures that the SE boot sequence is secure.

D. Key Management in a fTPM - INFORMATIVE

D.1 Introduction

This annex addresses the issues of key management in an fTPM. An fTPM is a TPM that is implemented as an application in a Protected Environment where the Protected Environment has to be reconstituted on each device reset.

In a discrete TPM, the resources used by the TPM are identifiable when the device is reset. For an fTPM, a Verified Boot is used to bring up the fTPM and allocate its resources. After the TPM application is running within the Protected Environment, normal system boot may continue as if the TPM were a discrete device (with the only difference being the physical interface).

A TPM has a considerable amount of persistent state that needs to be stored in some non-volatile (NV) memory. In a discrete TPM, NV memory is normally within the same package as the TPM processor so that memory can be considered to be a Shielded Location. However, in an fTPM system, the TPM firmware does not always have full control over access to NV. This means that the fTPM is required to encrypt and integrity protect the data in NV. Encryption of the NV data requires a key and, depending on how it is done, integrity protection may also need a key (such as an HMAC key). Since there is no permanent memory dedicated to the TPM where the TPM's keys can be stored, a method is needed to regenerate TPM-specific keys on each boot.

The key generation process has a few requirements for the keys that the TPM uses to protect its NV data;

1) Only the correct Protected Environment and TPM Application may re-generate the keys. A device may support multiple different Protected Environments and multiple versions of TPMs but only a specific combination of Protected Environment and TPM can be allowed to access the NV protection keys for that TPM.

2) A method is needed to allow data to be migrated when the firmware in the Protected Environment or the firmware in the TPM are updated. The NV keys used by the TPM are, according to requirement 1, required to be bound to a specific set of firmware. This means that, when the firmware changes, the keys change. In some instances, it might be desirable for all the TPM state to be lost on a firmware upgrade if that upgrade was to fix a severe security problem. However, if the update is simply to add a feature, losing TPM state would not provide an acceptable user experience. So, any key management scheme must provide a secure way to allow TPM state migration to new firmware while preventing migration to old, possibly compromised, firmware.

It is probable that a security problem will be found in the code of the TPM, the Protected Environment, or the code used to boot the Protected Environment. Updating the firmware to repair the problem is only half of the problem of re-establishing trust in the device. The other part is knowing that the problem has been fixed. It needs to be possible for at least one trustworthy entity to be able to validate that the TPM firmware has been updated (that entity could then issue a new certificate for the device). Without this, if the device compromise is serious enough, there would be no way of ever re-establishing trust in the device short of taking the device back to the manufacturer.

The representation of policy is complex and is vendor specific. The policy could, for example, require that each module be cryptographically signed by a specific signing key (normally a key known only to the device manufacturer). The policy could also be that a digest of the code match a value that is on a manifest (list of digests). Regardless of the method of validating a measurement of the boot code, a measurement is made and verified against policy. Typically, the policy will require that the measurement be a digest of the code in a certificate that is signed by the boot authority (or a designee of the boot authority). In addition to the digest of the code, the certificate would include the name of the module and its version number.

Section 5.3.2 describes an optional hardware PCR for recording the measurements made during measured boot. The method in 5.3.2 has limited value as there is no process by which the PCR measurements may be used reliably if the code that implements the PCR policy has been compromised. This annex describes a process of using these measurements to provide strong protections for the secrets used by a Protected Environment and "Trusted" Applications. Additionally, the methods in this annex enable reliable verification of the firmware running in a Protected Environment.

D.2 Qualifying Information

D.2.1 Certificates

The key generation processes in this annex rely on certificates. These certificates contain the values used by the Verified/Secure Boot process to determine if code to be executed is in policy for the device. This certificate would minimally contain a digest of the code to be executed and a signature over that code. For this implementation, it is also required that the certificate contain the name and version number of the module.

When a drawing or text indicates that a certificate is used in a computation, this does not mean that the entire certificate value has to be used. However, the value used must either contain the digest, name, and version number of the code to be executed or a value that is cryptographically bound to those values (such as a hash of the required values).

Before a certificate is used for key generation, validation of the associated code must be complete. That is, keys for code should not be generated unless it is known that the code will be run.

D.2.2 Boot Sequence

In the illustrated examples, the following boot sequence is assumed:

- **ROM** – Immutable code that cannot be modified after device manufacturer.
- **Initial Firmware** – Mutable code that completes preparation of the hardware and loads the Protected Environment.
- **Protected Environment** – Mutable code that host "Trusted" Applications The Protected Environment is able to protect its memory from access from any code not part of the Protected Environment including "Trusted" Applications.
- **"Trusted" Application** – An application that is run within the protected environment. Only the application and the Protected Environment can directly access keys and data used by the "Trusted" Application. Note: this definition means that the fTPM's Shielded Locations are accessible to the Protected Environment. The Protected Environment is required not to modify the Shielded locations of the fTPM because the Protected Environment does not provide Protected Capabilities as defined in the TPM 2.0 specification.

Other boot sequences are allowed with more or fewer steps. Each step that is present should have the same behaviour with respect to key generation as any other step.

NOTE The key generation scheme only applies to certified units. If a certified unit has several different steps as part of its execution, then it would still only count as a single unit for purposes of the key generation scheme

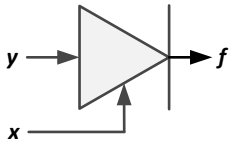
D.2.3 eFuses

The illustrations refer to *eFuses* but any technology with equivalent functionality may be substituted.

A mechanism to disable access to *eFuses* is required. The ROM code will disable *eFuses* access and erase any copy of the *eFuses* from memory before passing control to any other code.

D.2.4 Symbols

In the drawings associated with the description of the various methods, a special symbol is used to represent a one-way function involving two inputs. The symbol is:



The methods in this section do not require that the one-way function be implemented using a specific algorithm. It could be, for example, an encryption function, a hash function, or an HMAC function. However, the presumption is that, if the chosen function uses a key, then the *x* input is the key value. If the function is an extend operation, then the *x* input represents the 'current' value of the parameter to be extended and *y* is the input to extend by:

$$f := \mathbf{H}(x \,||\, y) \tag{1}$$

D.3 Generation of Protection Seeds

Figure 13 shows how the protection seeds would be generated for each stage of the boot process.

NOTE The input to a stage is a seed because crypto-hygiene may require that this value be used in a key-derivation function before being used for either encryption or integrity protection.

Each stage combines its seed (received from the previous stage) with the certificate of the next stage to produce the protection seed for the next stage (the description of the process is simplified by treating the *eFuses* value as an input from the prior stage).

It should be obvious from the drawing that if a certificate contains the digest of a next stage, then any change to a next stage will cause the seed for that stage, and all subsequent stages, to change. Since the *eFuses* value is unique to each device, the seed value of each stage is unique to the device and the digest of each prior stage.

It is required that each stage erase its input seed value before it begins execution of the next stage. This makes it impractical for any subsequent stage to generate key values that might be used by different versions of the code. For example, without access to IF_Seed, a malicious Protected Environment cannot compute the PE_Seed for a non-malicious version of the code.

NOTE Access to *eFuses* value is required to be disabled before the ROM code exits.

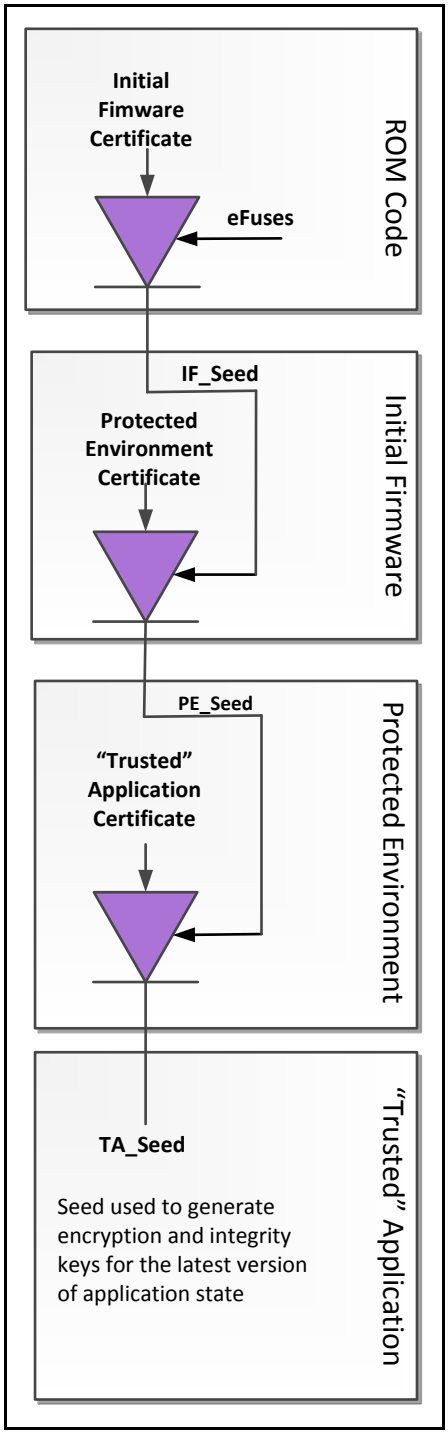


Figure 13 — Generating Protection Keys

As will be illustrated later, a ROM implementation may generate IF_Seed using a value other than *eFuses*.

D.3.1 Firmware Update

The implementation of Figure 13 does not address the second requirement for the confidentiality protection – support for firmware updates. This is because a change to a certificate will change the seed values for all

subsequent seeds. If, for example, the Initial Firmware is changed, then the seeds in the Initial firmware, Protected Environment, and all “Trusted” Applications will change. This means that stages that are not changed would no longer be able to access their protected data.

Figure 14 illustrates the process for rekeying after a firmware update. The process uses the current certificates for the code and the certificate for the code replaced during the update process. This allows computation of the seeds for the previous firmware and the current firmware. When the system has booted following a firmware update, and each code component has re-encrypted its data, the old certificate can be removed and the next boot can follow the normal sequence and generate a single seed per stage.

NOTE In the figure, the “[C]” designation indicates that the certificate for the current firmware is being used. The “[P]” designation indicates that a certificate for a previous version of the firmware is being used. “[P]” is used rather than “[C-1]” because the numeric value of the version number might not be one less than the current version number.

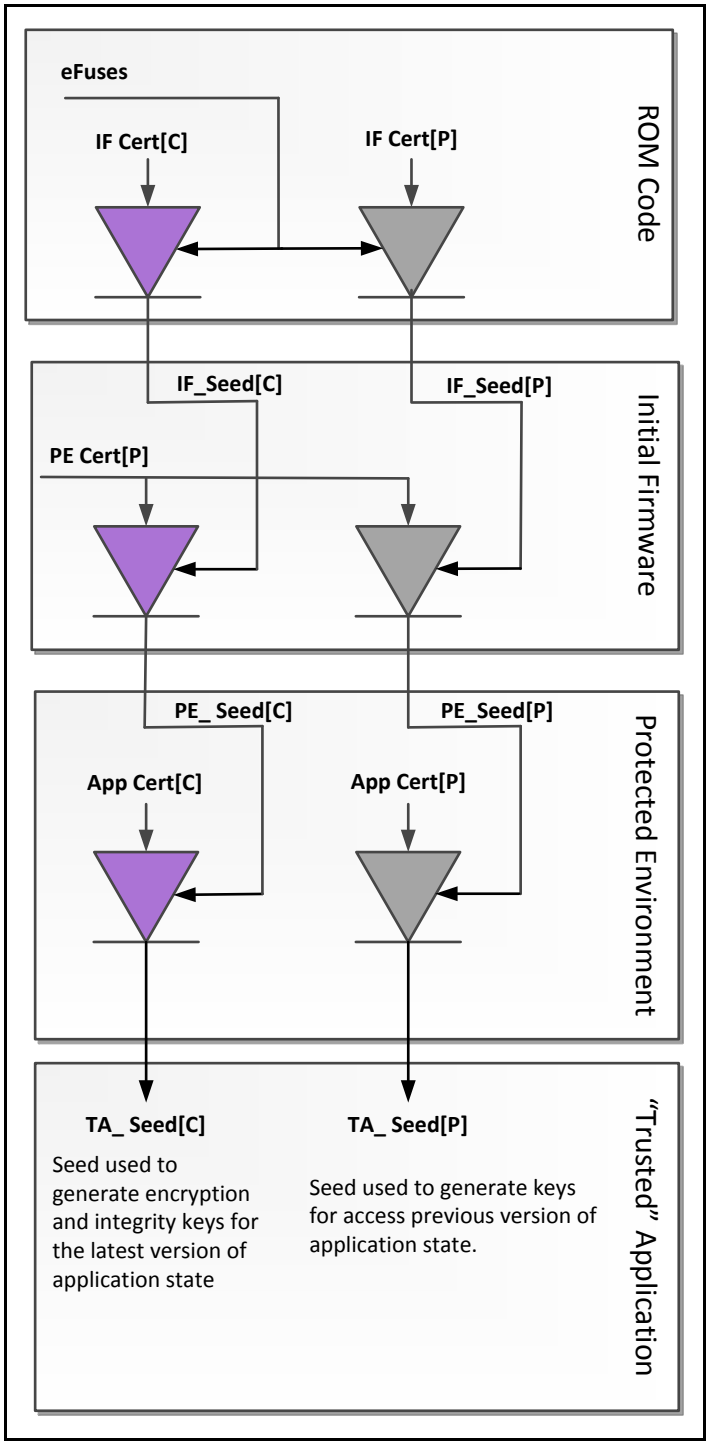


Figure 14 — Rekeying After Firmware Update

In order to make this process secure, the verification code at each stage needs to:

1) Verify that the new code and its certificate are valid (no change from normal processing);

2) Verify that the same entity signed both the old and new certificates;

3) Verify that the signature of the old certificate is valid (see note below);

4) Verify that the names of the code are the same; and

5) Verify that the version number for the new code is greater than the version number for the old code.

NOTE It may not be necessary to validate the signature of the old certificate if the entire certificate is always included as input to the key generation rather than just the signature block. If only the signature block is used in key generation, then the verifier needs to ensure that the signature block is properly associated with the other data (name, version number, etc.). If all of the verified data is included in the key computation, then no substitution is possible and the signature does not have to be checked.

Figure 14 shows the worst case for rekeying where all stages have an update. .Figure 16 illustrates a partial rekeying when the Initial Firmware has change and a “Trusted” Application has changed but the Protected Environment code remains the same. If a stage has only one input seed and the next stage has not changed, then the stage will only have one output seed. However, if the stage has two input seeds, then there will be two output seeds regardless of whether there is a code change in the subsequent stage.

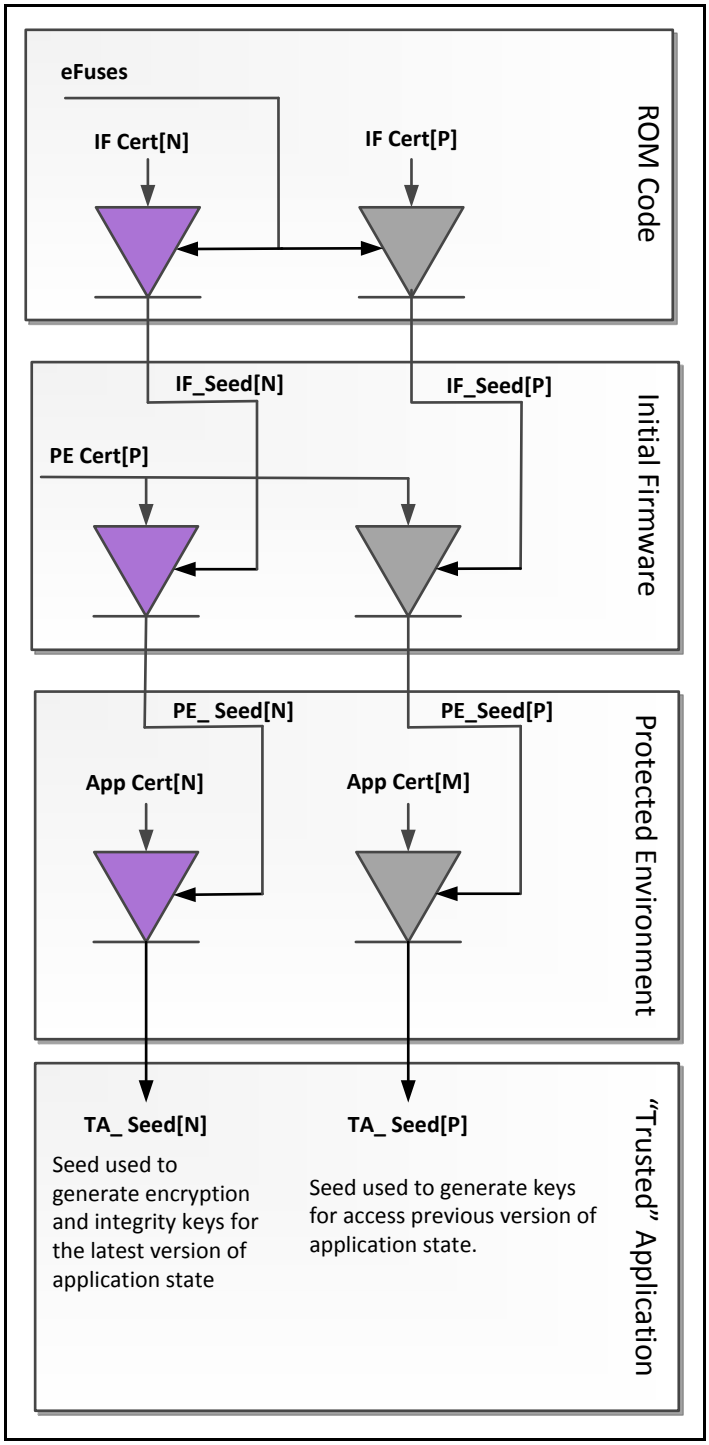


Figure 15 — Partial Rekeying

Figure 16 illustrates how the system would boot if a “Trusted” Application were the only code updated.

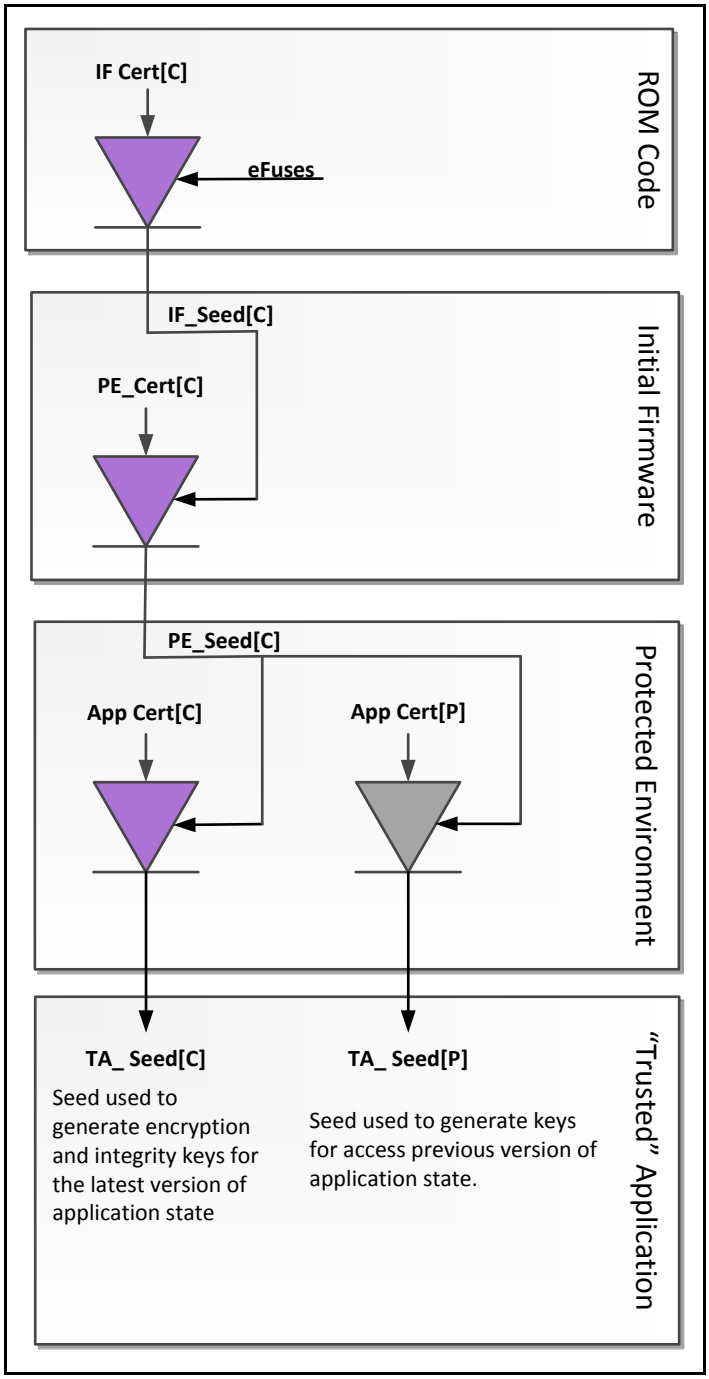


Figure 16 — Rekeying After Application Update

D.3.2 Firmware Identity

As explained in the introduction, being able to perform firmware updates is only part of the problem. The other is proving that the update has been applied. Depending on the severity of the flaw being addressed by the update, there might be no existing means of proving that the update has been done.

Figure 16 shows a process for producing a verifiable firmware identity. The process for generating the firmware identity is similar to the process for generating the data protection seed values. The $eFuses_B$ value used in this process is not the same as the *eFuses* value used to seed the protection keys but, otherwise,

the process is identical. In addition, as with the protection seeds, the input to each stage is erased from memory before the next stage begins execution.

NOTE Access to $eFuses_B$ is required to be disabled before the ROM code exits.

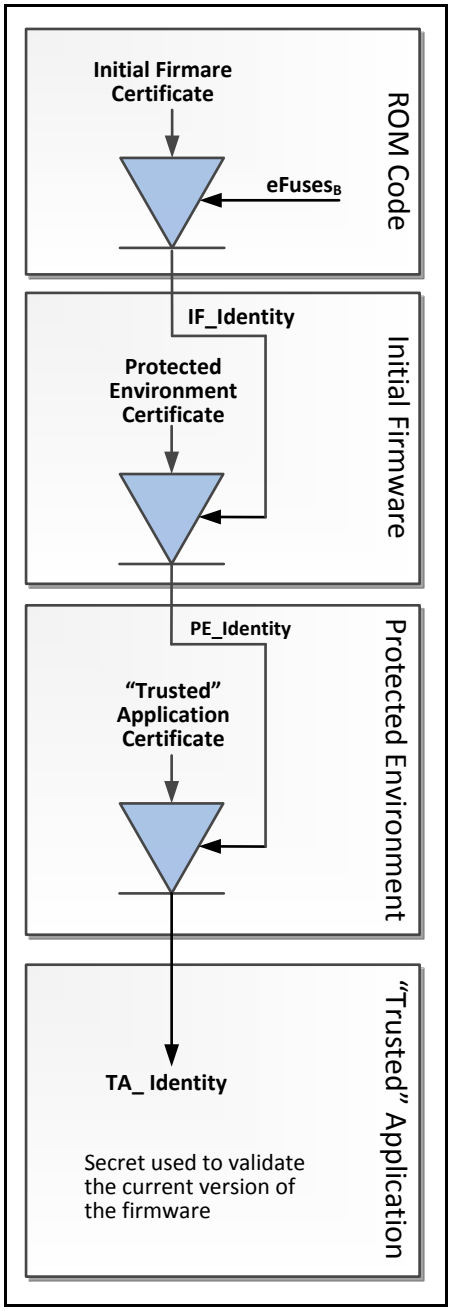


Figure 17 — Generation of Identity Secret

The value generated through this process should remain secret. It is intended to be used in a protocol that allows the device manufacturer (or their designee) to verify the firmware that is running on a device. As with the encryption key values, the identity will change with any change to firmware in the boot path. As long as the ROM erases the root values and disables access to *eFuses$_B$*, it is impractical to generate an identify value that is valid for a collection of firmware other than the one on the device.

NOTE A malicious Initial Firmware could use a false value for the Protected Environment Certificate and generate a set of keys that are not accurate. However, the malicious Initial Firmware will not have the same certificate value as non-malicious code. So the keys and identity values generated in the system with malicious firmware cannot match the keys and identity values generated in a non-malicious system.

In a TPM, the TPM's *TA_Identity* (*TPM_Identity*) would be associated with one of the TPM_RH_AUTH_xx values (TPM_RH_AUTH_00 in the reference design). This handle would be used as the *Bind* value in TPM2_StartAuthSession(). This will cause the TPM to compute a *sessionSecret* value that is based on the value of *TPM_Identity* and the session nonces. This secret value will be unique to a device and firmware combination. A device manufacture would know the $eFuses_B$ value for a device and, with a list of firmware versions, be able to compute the expected value of TA_Identity for the TPM. The device manufacturer could, by getting the correct HMAC from the TPM, know that the TPM is running the "advertised" firmware.

To perform the necessary computation, the device manufacturer needs to be able to know the *eFusesB* value used by a device. This means that there must be some unique identifier on the device that can be read allowing the correct $eFuses_B$ value to be accessed. One way of doing this is to use the $eFuses_B$ value in a one-way function to produce a value that can be accessed by anyone. Suggested implementations are shown in Figure 18. For either of these, the manufacturer, could retain the value of $eFuses_B$ and be able to compute the *DevicePublicID* and the associated firmware identities values. For option A, the manufacturer would have the option of storing *DevicePublicID* and *IdentityRoot* as an alternative to storing the $eFuses_B$ value.

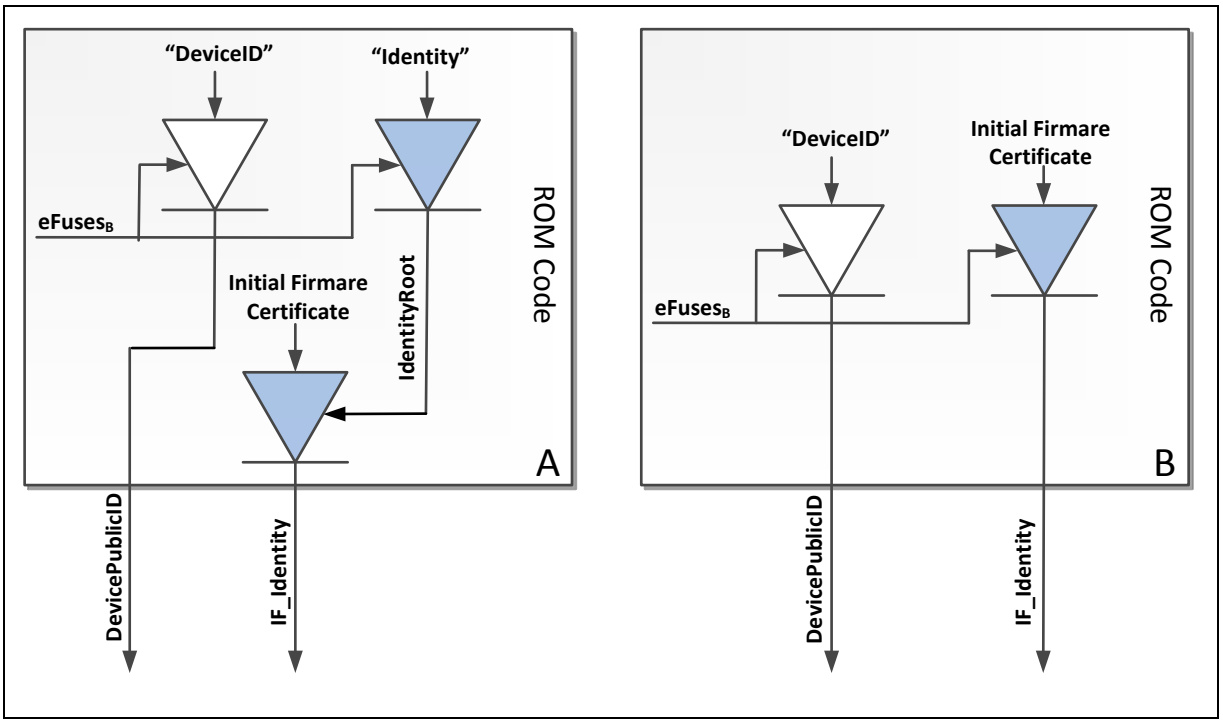


Figure 18 — Identity *eFuses* Sharing

NOTE The quoted values ("DeviceID" and "Identity") could be, but are not required to be, literal strings. They represent values chosen by the device manufacturer to differentiate the generated values.

D.4 Supporting Multiple "Trusted" Applications

A trusted application receives an encryption seed (or seeds) and the identity value for the application. The keys and identity are unique to a device and the firmware in the boot path of the application. The Protected Environment may support any number of "Trusted" Applications.

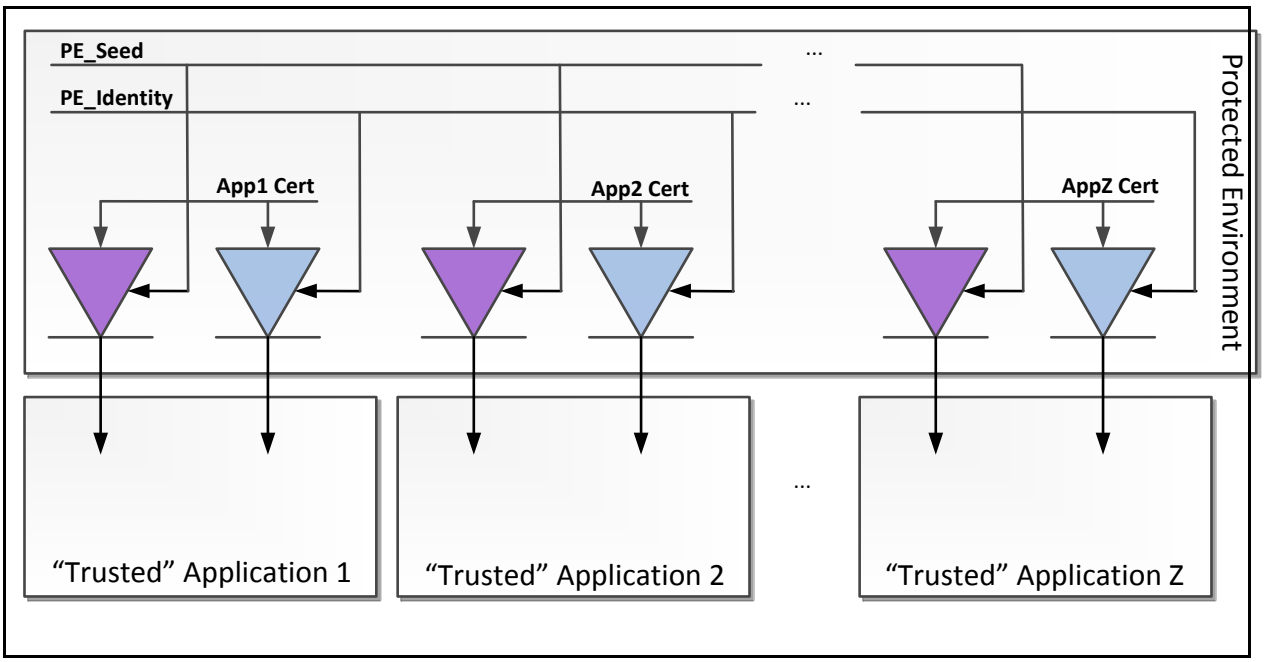


Figure 19 — Key Distribution for Multiple "Trusted" Applications

D.5 Deferred Processing of an Update

The certificates for "Trusted" Applications may be maintained by the Protected Environment and not be part of the update process for the Initial Firmware or Protected Environment update. In the normal course of an update, when the Protected Environment and all the "Trusted" Applications have re-encrypted their protected data, the certificate database would be updated indicating that the next boot can use the single seed path. The Protected Environment may, instead, keep a database of all of the certificates for its applications as part of its protected state. When a "Trusted" application runs, the Protected Environment can check its database to see if the application has been run since being updated. If not, the Protected Environment may provide the pairs of protection keys that the application needs in order to update its database.

D.6 Miscellaneous

This section contains variations for processing and *eFuse*s handling by the ROM

NOTE Some of the figures show how an RPMB HMAC key (*RpmbAccessKey*) can be generated from the *eFuses* values. These are illustrative only.

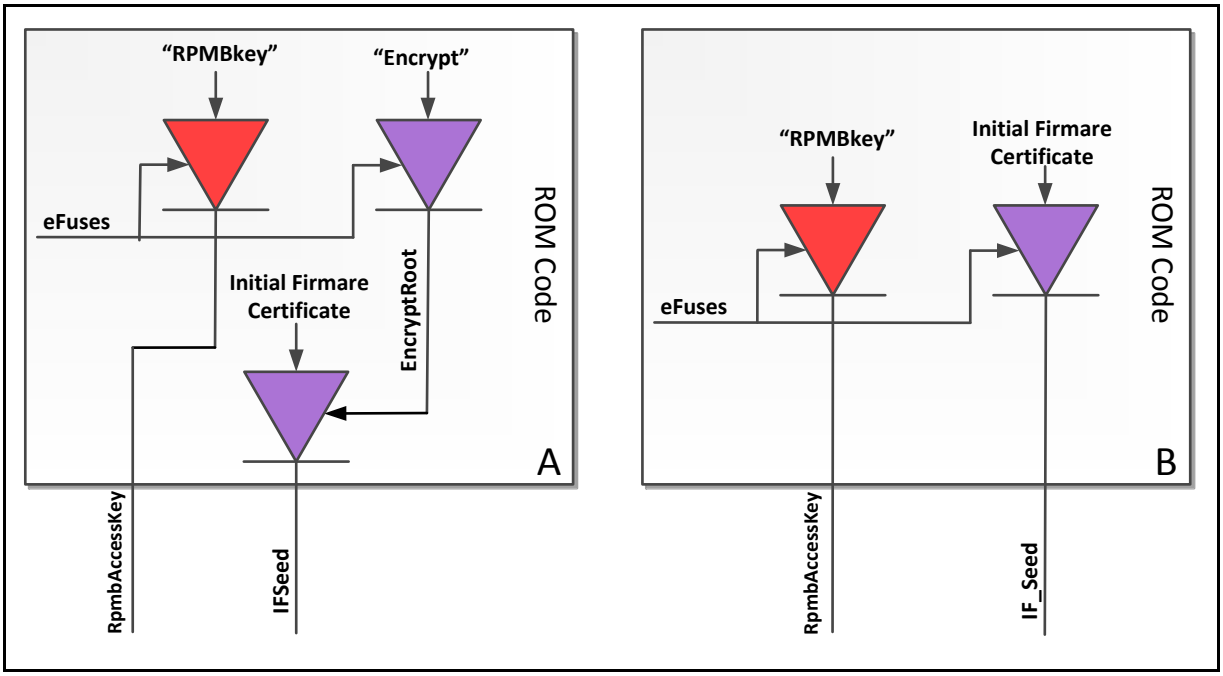


Figure 20 — Integrity/Confidentiality Fuse Sharing

Both of these implementations use the *eFuses* value as a key value input into a one-way function. This means that knowing the *RpbmAccessKey* does not allow computation of IF_Seed.

The "B" implementation seems to be the obviously better choice because of implementation complexity but there are other factors that might make the "A" implementation the better choice (see Figure 21).

The preferred implementation is shown in Figure 21 where the protection values and the identity values are derived from different *eFuses* values. The assumption is that the device manufacturer will retain knowledge of the value of $eFuses_B$ for identification and certification purposes but the value of $eFuses_A$ would not be retained, so the manufacturer would not know the encryption keys derived from those fuses.

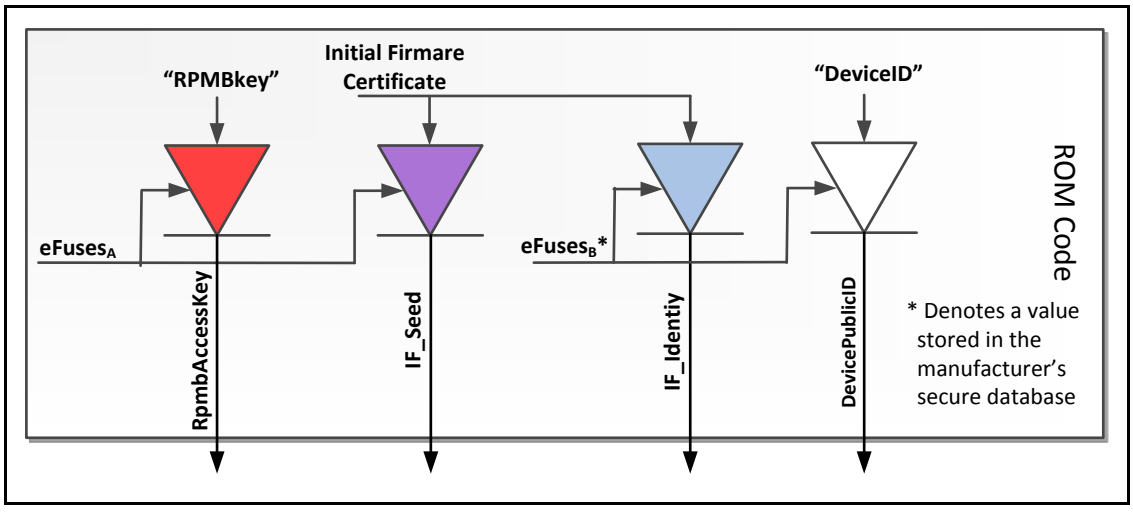


Figure 21 — Preferred Use of *eFuses*

If the device does not have enough *eFuses* to allocate to two different seed values, then the implementation in Figure 22 is a possibility.

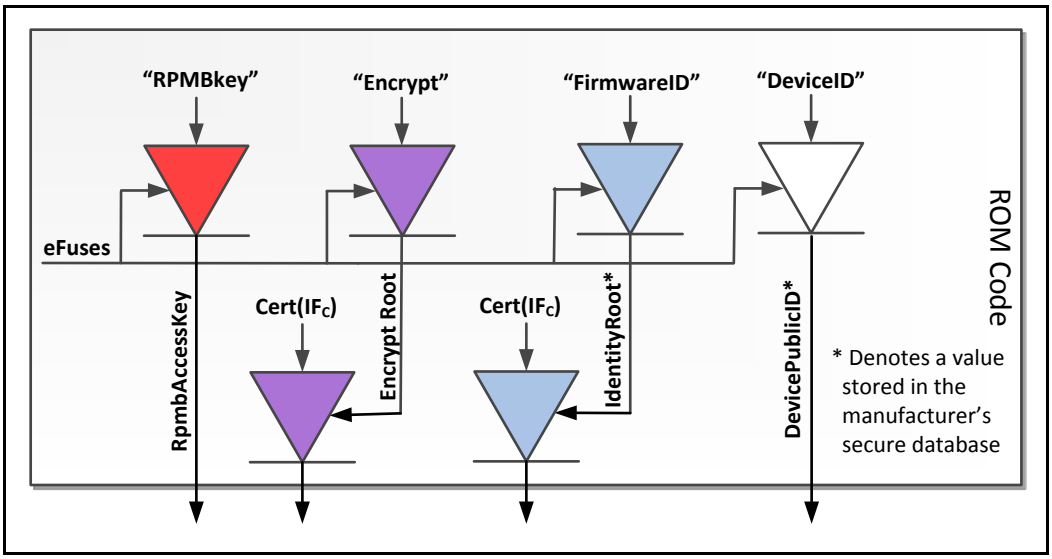


Figure 22 — Alternative Fuse Use

If the arrangement in Figure 22 is used, then it is not permitted for the manufacturer to retain the *eFuses* value as this would give the device manufacturer the ability to access user secret data. Instead, the manufacturing process would have to produce an *IdentityRoot*, pair that with the *DevicePublicID* value, and somehow communicate these values to a secure database. With the arrangement in Figure 21, the manufacturer could specify the desired value for $eFuses_B$ and allow the device to generate a random value for $eFuses_A$ that is never known outside of the ROM.

D.7 Immutability of ROM

If the amount of code needed for full verification of Initial Firmware is too large to fit into the available ROM space on an SoC, then the code has to be split with some of the code remaining in ROM and the remainder in external flash (or equivalent). Any data stored in external flash is subject to modification but it is still possible to have it be considered as immutable if any change to the code would result in a boot failure.

The code in the ROM on the SoC could simply be enough code to load and hash check the code in external flash. During the manufacturing process, the flash version of ROM for the platform could be installed and *eFuses* blown to indicate the required hash for the flash ROM. This would allow the same SoC to have any number of different ROM versions without having to change the ROM on the SoC itself.

If the code in flash can be changed after manufacturing and have the boot process continue without remediation, then the code is not ROM and has to be validated against a certified value.

D.8 Remediation

The generation of the protection seeds and identity values uses certificates that need to be verified. When the certificate associated with the current version of the firmware is not valid then it is expected that the device will enter remediation code where it is possible to re-flash the firmware and put it back in a valid state. Before starting remediation, all secrets (including *eFuses* values) are required to be erased from memory.

NOTE It may possible to recover the firmware in a device without losing the secrets used by the Protected Environment or the “Trusted” Applications. If the firmware can be put into a state that allows the proper protection seed values to be created, then the secrets need not be lost.

A second failure case occurs when the certificate for the current version of some stage is not compatible with the certificate for the previous version of the stage. This failure could occur because different signing keys are used, the names for the code do not match, or the version number of the previous version is not less than the current version. For any of these failures, the boot may continue but the seed values from a stage may only use the current certificate for the stage that is changed. Referring to Figure 15, the Initial Firmware would use the current certificate value whether there was no change to the Protected Environment or the previous certificate did not validate against the current certificate.

This handling of mismatched certificates allows recovery of a system after a very severe security problem – loss or disclosure of a vendor’s signing key. Previous versions of the firmware are no longer trusted because the signing key cannot be trusted. However, new firmware can be issued using a new signing key.. Secrets based on the compromised firmware are lost but the device can be re-provisioned; and, if the identity chain is implemented, the device can be recertified.

NOTE The reason that the old secrets are discarded is that an attacker could have issued a malicious version of software signed with a stolen signing key. This firmware may have disclosed all of the secrets in a device so those secrets need to be discarded.