# Contents

About this document iv

Release Information iv

Arm Non-Confidential Document Licence (“Licence”) vi

References vii

Terms and abbreviations viii

Feedback x

Feedback on this book x

1 Introduction 11

1.1 Scope 12

1.2 Assumptions 12

2 PSA terminology 13

2.1 Trusted subsystems 14

2.2 Trusted memory 15

2.3 Image manifest 16

3 Requirements 16

3.1 Approved cryptographic algorithms 16

3.1.1 RSA 17

3.1.2 ECC 17

3.1.3 EdDSA 17

3.1.4 Hashing 17

3.1.5 Key derivation 17

3.1.6 Side channels 17

3.2 Chain of Trust 17

3.2.1 Single provider 18

3.2.2 Multiple dependent providers 19

3.2.3 Multiple independent providers 19

3.2.4 Certificate creation and key management 20

3.3 Image verification 20

3.3.1 Secure storage 22

3.4 Trusted Boot Flow 23
3.4.1 PRoT immutable bootloader requirements 25
3.4.2 PRoT runtime requirements 26

3.5 Update process (informational) 27

3.6 Image manifests 28

3.7 Anti-rollback protection 30

3.8 Measured boot and attestation 32

4 Architectural variants 34
4.1 Minimal boot from on-chip storage 34
4.2 Minimal boot from off-chip storage 35
4.3 Minimal boot using a passive security module 36
4.4 Minimal boot using a security processor 37

5 eFlash considerations (informational) 38

6 Delegated signing schemes (optional) 39
6.1 Key certificate scheme 40
6.1.1 Revocation workflow 40

Example manifest using the IETF SUIT draft 41

Example manifest using the X.509v3 standard 41
About this document

This manual is part of the Arm Platform Security Architecture (PSA) family of specifications. It defines the security architecture and technical requirements to create a trusted boot process. A trusted boot process involves verifying and measuring software in accordance to a chain of trust.

Release Information

The change history table lists the changes that have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 2018</td>
<td>1.0 Alpha 0</td>
<td>Confidential</td>
<td>• First development release.</td>
</tr>
<tr>
<td>June 2018</td>
<td>1.0 Alpha 1</td>
<td>Confidential</td>
<td>• Second development release.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Aligned with PSA Security Model and updated terminology.</td>
</tr>
<tr>
<td>August 2018</td>
<td>1.0 Alpha 2</td>
<td>Confidential</td>
<td>• Third development release</td>
</tr>
<tr>
<td>October 2018</td>
<td>1.0 Beta 0</td>
<td>Non-confidential</td>
<td>• IETF SUIT manifest format is now recommended for constrained devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X.509v3 certificates are now recommended for application processors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Clarification - NSPE software is permitted to be loaded into off-chip memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Update client section has been renamed to Update process, and is now informational</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Manifest requirements have been separated from the update section</td>
</tr>
<tr>
<td>February 2019</td>
<td>1.0 Beta 1</td>
<td>Non-confidential</td>
<td>• New PSA terminology section introduced as Section 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Rules no longer use on-chip/off-chip terminology. Trust boundaries are defined and used in the rules to refer to trusted/untrusted peripherals/memory/storage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Image hash of encrypted images must be of ciphertext not the plaintext.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Removed Section 6.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Removed security epoch as it is no longer defined in the PSA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Updated rollback section to allow for implementation-defined rollback schemes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 2018</td>
<td>1.0 Alpha 0</td>
<td>Confidential</td>
<td>• First development release.</td>
</tr>
<tr>
<td>June 2018</td>
<td>1.0 Alpha 1</td>
<td>Confidential</td>
<td>• Second development release.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Aligned with PSA Security Model and updated terminology.</td>
</tr>
<tr>
<td>August 2018</td>
<td>1.0 Alpha 2</td>
<td>Confidential</td>
<td>• Third development release</td>
</tr>
<tr>
<td>October 2018</td>
<td>1.0 Beta 0</td>
<td>Non-confidential</td>
<td>• IETF SUIT manifest format is now recommended for constrained devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• X.509v3 certificates are now recommended for application processors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Clarification - NSPE software is permitted to be loaded into off-chip memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Update client section has been renamed to Update process, and is now informational</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Manifest requirements have been separated from the update section</td>
</tr>
<tr>
<td>February 2019</td>
<td>1.0 Beta 1</td>
<td>Non-confidential</td>
<td>• New PSA terminology section introduced as Section 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Rules no longer use on-chip/off-chip terminology. Trust boundaries are defined and used in the rules to refer to trusted/untrusted peripherals/memory/storage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Image hash of encrypted images must be of ciphertext not the plaintext.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Removed Section 6.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Removed security epoch as it is no longer defined in the PSA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Updated rollback section to allow for implementation-defined rollback schemes</td>
</tr>
</tbody>
</table>
• Introduced `R25_TBFU_EXEC` to be explicit about system behavior during security violations.

• Example manifests are included in the appendix for the IETF SUIT and ITU X.509v3 standards.

• Introduced `R80_TBFU_MANIFEST` about disabling trusted boot for the NSPE.

• Introduced `R30_TBFU_KEYS` and `R40_TBFU_KEYS`
Arm® Platform Security Architecture Trusted Boot and Firmware Update

Copyright ©2017-2018 Arm Limited or its affiliates. All rights reserved. The copyright statement reflects the fact that some draft issues of this document have been released, to a limited circulation.

Arm Non-Confidential Document Licence (“Licence”)

This Licence is a legal agreement between you and Arm Limited (“Arm”) for the use of the document accompanying this Licence (“Document”). Arm is only willing to license the Document to you on condition that you agree to the terms of this Licence. By using or copying the Document you indicate that you agree to be bound by the terms of this Licence. If you do not agree to the terms of this Licence, Arm is unwilling to license this Document to you and you may not use or copy the Document.

This Document is NON-CONFIDENTIAL and any use by you is subject to the terms of this Licence between you and Arm.

Subject to the terms and conditions of this Licence, Arm hereby grants to you under the intellectual property in the Document owned or controlled by Arm, a non-exclusive, non-transferable, non-sub-licensable, royalty-free, worldwide licence to:

(i) use and copy the Document for the purpose of designing and having designed products that comply with the Document;

(ii) manufacture and have manufactured products which have been created under the licence granted in (i) above; and

(iii) sell, supply and distribute products which have been created under the licence granted in (i) above.

You hereby agree that the licences granted above shall not extend to any portion or function of a product that is not itself compliant with part of the Document.

Except as expressly licensed above, you acquire no right, title or interest in any Arm technology or any intellectual property embodied therein.

THE DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. Arm may make changes to the Document at any time and without notice. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENCE, TO THE FULLEST EXTENT PERMITTED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENCE (INCLUDING WITHOUT LIMITATION) (I) LICENSEE’S USE OF THE DOCUMENT; AND (II) THE IMPLEMENTATION OF THE DOCUMENT IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS LICENCE. THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.
This Licence shall remain in force until terminated by you or by Arm. Without prejudice to any of its other rights, if you are in breach of any of the terms and conditions of this Licence then Arm may terminate this Licence immediately upon giving written notice to you. You may terminate this Licence at any time. Upon termination of this Licence by you or by Arm, you shall stop using the Document and destroy all copies of the Document in your possession. Upon termination of this Licence, all terms shall survive except for the licence grants.

The Document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of the Document complies fully with any relevant export laws and regulations to assure that the Document or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

If any of the provisions contained in this Licence conflict with any of the provisions of any click-through or signed written agreement with Arm relating to the Document, then the click-through or signed written agreement prevails over and supersedes the conflicting provisions of this Licence. This Licence may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this Licence and any translation, the terms of the English version of this Licence shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm (or its subsidiaries) in the EU, US and/or elsewhere. All rights reserved. No licence, express, implied or otherwise, is granted to you under this Licence, to use the ARM trade marks in connection with the Document or any products based thereon.

The validity, construction and performance of this Licence shall be governed by English Law.

Copyright © 2018 Arm Limited (or its affiliates). All rights reserved.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585
References
This document refers to the following documents.

<table>
<thead>
<tr>
<th>Ref</th>
<th>Document Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>ARM DEN 0079</td>
<td>PSA Security Model</td>
</tr>
<tr>
<td>[10]</td>
<td>NIST 800-57</td>
<td>NIST Special Publication 800-57 Part 1 Revision 4 Recommendation for Key Management. NIST.</td>
</tr>
<tr>
<td>[12]</td>
<td>IETF SUIT</td>
<td>Software Update for Internet of Things (SUIT) manifest format. IETF.</td>
</tr>
<tr>
<td>[13]</td>
<td>TCG Client Profile</td>
<td>TCG PC Client Platform Firmware Profile Specification (Family “2.0” Level 00 Revision 1.03 v51)</td>
</tr>
</tbody>
</table>

Terms and abbreviations
This document uses the following terms and abbreviations.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard, a symmetric-key encryption standard</td>
</tr>
<tr>
<td>ARoT</td>
<td>Application Root of Trust (a PSA term)</td>
</tr>
<tr>
<td>CIA</td>
<td>Confidentiality Integrity Availability</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial of Service</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read-Only Memory</td>
</tr>
<tr>
<td>eFlash</td>
<td>See Internal flash</td>
</tr>
<tr>
<td>eFuse</td>
<td>OTP memory, available in very limited quantity</td>
</tr>
<tr>
<td>eMMC</td>
<td>Embedded multi media card. Low cost flash memory with a built-in controller</td>
</tr>
<tr>
<td>HMAC</td>
<td>Hashed Message Authentication Code</td>
</tr>
<tr>
<td>HUK</td>
<td>Hardware Unique Key</td>
</tr>
<tr>
<td>Internal flash</td>
<td>On-chip embedded flash</td>
</tr>
<tr>
<td>KDF</td>
<td>Key Derivation Function</td>
</tr>
<tr>
<td>Manifest</td>
<td>Signed metadata for a firmware image</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro-controller unit</td>
</tr>
<tr>
<td>MPU</td>
<td>Memory Protection Unit</td>
</tr>
<tr>
<td>MTP</td>
<td>Multi-Time Programmable. A characteristic of some type of NVM</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology (<a href="http://www.nist.gov">http://www.nist.gov</a>)</td>
</tr>
<tr>
<td>NSPE</td>
<td>Non-Secure Processing Environment (a PSA term)</td>
</tr>
<tr>
<td>NSPE-PK</td>
<td>Public Key of the Non-Secure Processing Environment</td>
</tr>
<tr>
<td>NVM</td>
<td>Non-volatile memory</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer</td>
</tr>
<tr>
<td>OTA</td>
<td>Over-The-Air</td>
</tr>
<tr>
<td>OTP</td>
<td>One Time Programmable. A characteristic of some types of NVM</td>
</tr>
<tr>
<td>PKI</td>
<td>Public Key Infrastructure</td>
</tr>
<tr>
<td>PPC</td>
<td>Peripheral Protection Controller</td>
</tr>
<tr>
<td>PRoT</td>
<td>PSA Root of Trust (a PSA term)</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-only memory</td>
</tr>
<tr>
<td>ROTPK</td>
<td>Root of Trust Public Key (for firmware verification)</td>
</tr>
<tr>
<td>RPMB</td>
<td>Replay-protected Memory Block</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest, Shamir and Adleman. An algorithm for public-key cryptography.</td>
</tr>
<tr>
<td>RSA-PSS</td>
<td>RSA Probabilistic Signature Scheme</td>
</tr>
<tr>
<td>Runtime firmware</td>
<td>Generic term to describe the firmware that executes after boot has completed</td>
</tr>
<tr>
<td>SE</td>
<td>Secure Element. An example of a secure element is a smart card.</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SPE</td>
<td>Secure Processing Environment. Contains the PRoT and the ARoT.</td>
</tr>
<tr>
<td>SPE-PK</td>
<td>Public Key of the Secure Processing Environment</td>
</tr>
<tr>
<td>SPM</td>
<td>Secure Partition Manager</td>
</tr>
<tr>
<td>Trusted subsystem</td>
<td>A self-contained subsystem providing security functionality e.g. a secure element</td>
</tr>
<tr>
<td>UC</td>
<td>Update client</td>
</tr>
<tr>
<td>Update Client</td>
<td>Code responsible for fetching, verifying and storing updates</td>
</tr>
<tr>
<td>XIP</td>
<td>eXecute-In-Place</td>
</tr>
</tbody>
</table>

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to arm.psa-feedback@arm.com. Give:

- The title (Arm® Platform Security Architecture Trusted Boot and Firmware Update).
- The number and release (DEN0072 1.0 Beta 1).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.
1 Introduction

Arm’s Platform Security Architecture (PSA) is a holistic set of:

- Threat models.
- Security analyses.
- Hardware and firmware architecture specifications.
- Open source firmware reference implementations.

PSA provides a recipe, based on industry best practice, that allows security to be consistently designed in, at both a hardware and firmware level.

Platforms that wish to provide secure services have a fundamental requirement to be trustworthy. A key component of their design is a trusted boot process, which ensures the integrity of the code executed on the System-on-Chip (SoC) from the very first instruction through to the loading of the operating system and the main applications.

This document, PSA Trusted Boot and Firmware Update (PSA-TBFU), describes the security requirements to ensure that from the point of reset of a SoC, only the correct and intended firmware, operating system, and Root of Trust services, are authenticated, loaded and initialized on the SoC. Authentication and authorization of firmware updates is done using standard cryptographic certificates and system keys.

The design of a secure product depends on the analysis of the threats to its assets in the context of the application use cases. PSA-TBFU presents the best security principles for developing a trusted boot process across a range of different systems.

SoCs vary in complexity and capability to meet different cost targets. Optional recommendations are given for SoCs that have sufficient resources to provide more robust protection. System developers are encouraged to test the given recommendations to increase the resiliency and longevity of their products.

In this document, trusted boot refers specifically to the concept of validating that an image is authorized before booting it and providing an audit log of the boot process. Since this occurs recursively it creates a “chain of trust” as shown in Figure 1. Firmware update refers to the verification of the update before storing it to flash. These
two concepts are complimentary to each other.

![Figure 1: The Chain of Trust approach with N defined stages and optional decryption](image)

### 1.1 Scope

Trusted boot and update mechanisms must be used to bootstrap and protect critical code and data of the system from unauthorized modification. This ensures a solid foundation for providing secure services. Loaded components are also measured to prove the system state to a remote party.

Protection against unauthorized physical access depends on the hardware capabilities and the threat model for a particular product. Therefore, a detailed threat model and security review for a particular product is recommended, which is not within the scope of this document. Robustness rules and informational statements are included where appropriate.

Attacks on systems continuously evolve, with the effect that old security defenses must be strengthened, and new security defenses must be implemented to maintain the required level of security. The requirements described in this document represent best practice at the time of writing. Some requirements provide implementation flexibility and improvements when compared to older documentation provided by Arm. In all cases, the differences are in the degree of security that is provided, or that is demanded by other market specifications.

Operating system package managers or equivalent are out of the scope of this document.

Production or diagnostic test modes used in repair and rework, such as the manufacturing and key provisioning test modes are outside the scope of this document. The initial setting of an on-board real time clock for trusted time is also out of scope.

This document makes no assumptions about the network connectivity of the system. To provide a trusted boot process all mandatory requirements described in this document must be implemented fully.

### 1.2 Assumptions

This document assumes the reader is familiar with standard techniques in cryptography, such as authentication, hashing, encryption, Public Key Cryptography Standards (PKCS), NSA suite B and digital certificates. It does not describe these concepts in detail.

This document assumes the reader is familiar with the **Trusted Base System Architecture (TBSA)**, which includes the necessary hardware features expected for a PSA platform. At the time of writing there are two variants of TBSA:
• **TBSA for M** is the Arm specification for building a microcontroller with best practice security properties
• **TBSA-Client** is the Arm specification for building an application processor with best practice security properties

This document, PSA-TBFU, aims to support a class of system which:

• can perform public key cryptography
• have a boot ROM or a lockable sector of on-chip flash to emulate a boot ROM
• supports a level of memory isolation that can protect the code and data of security-critical functionality

This document assumes that the reader wants to create firmware that has a trusted boot process and a secure update process.

If no masked boot ROM or security processor is present, the internal flash must have an immutable part and a mutable part. The immutable part may be implemented by lock fuses and is thus immune from malware. The mutable part is, in general, updatable by the immutable part and thus is able to be returned to a known good state. The immutable part acts a boot ROM and must meet the same requirements. The words “boot ROM” and “immutable bootloader” are used interchangeably in this specification.

2 PSA terminology

The trusted boot process has multiple stages and may use multiple firmware images. Each stage of the boot process uses standard cryptography to validate the next stage of boot, ensuring that unauthorized software is never loaded. This is known as a chain of trust. Additionally, each stage is measured and logged in order for Root of Trust services to securely report the software state for auditing purposes (for more information see Section 3.8). This specific aspect is referred to as a Measured Boot.

If the hardware supports isolation of software, then the software can be split into two security domains:

• **Secure Processing Environment (SPE),** which contains the:
  
  - **PSA Root of Trust (PRoT)** for providing roots of trust. These roots of trust are used for securing sensitive data such as secrets, platform state, and cryptographic key material. This is typically the Trusted Computing Base (TCB) for the SPE. This may be comprised of multi-stage bootloaders, a Trusted OS or a Secure Partition Manager (SPM), with the possible addition of trusted subsystems.
  
  - **Application Root of Trust (ARoT)** for providing application-specific security services while remaining isolated from the non-secure applications. It provides security functionality to the non-secure applications, and typically depends on the primitives provided by the PRoT.

• **Non-Secure Processing Environment (NSPE)** for general purpose functionality that is not security-critical. This includes communication stacks, device drivers, task management and application software. It is unable to undermine the integrity and confidentiality of the SPE. The NSPE may be isolated on a separate processor from the SPE. On an Arm processor with TrustZone security extensions the NSPE is known as the “Normal World”. On other systems, the NSPE may be a separate processor that is unable to access SPE resources.
Each of these subsystems must be securely loaded, starting with the PRoT.

A platform may have several images depending on the system design. The code in the SPE should be small in size and limited in purpose such that the software can be better validated for programming errors or design mistakes.

### 2.1 Trusted subsystems

Trusted subsystems are blocks of security IP that sit within the trust boundary of the PRoT. They provide RoT security services to the system. The PRoT attests their implementation and configuration. They may be integrated, or external and bound to the SoC. Examples include:

- Trusted peripherals which support cryptographic operations
- Secure elements and enclaves, each of which contains its own local RoT and its own local security life cycle
- DRAM protection systems
- Trusted real-time clock

Secure Elements are independent subsystems which provide a set of RoT services for the system. There are many variations that might be on-chip or off-chip, have market-specific features and degrees of programmability. These are all collectively referred to as trusted subsystems.
If a trusted subsystem is off-chip, it is expected that the communication channel between the SoC and the trusted subsystem are cryptographically paired. Cryptographic pairing enables secure communications between components and prevents unauthorized replacement of a trusted subsystem.

### 2.2 Trusted memory

Each system has physical trust boundaries. Some trust boundaries may include external RAM depending on the target market.

Trusted memory refers to connected RAM and NVM that are trusted to be sufficiently secure against a common set of threats and adversaries. NVM includes all types of persistent storage.

The SPE uses trusted memory to initialize and load SPE components. SPE resources must be protected from the NSPE.

If RAM is shared between SPE and NSPE then they must be logically partitioned such that only the SPE can access SPE memory. This may involve an on-chip mechanism such as an address space controller or a security attribution unit.

If the SPE and NSPE do not share RAM then the SPE memory must be physically isolated from the NSPE such that only the SPE can access SPE memory.

<table>
<thead>
<tr>
<th>System class</th>
<th>Trusted memory</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>On-chip* memory</td>
<td>Off-chip memory, such as external flash, may be easily accessible to a locally present attacker, and may be accessible during manufacturing or deployment.</td>
</tr>
<tr>
<td>Endpoint</td>
<td>On-chip* memory + trusted off-chip RAM</td>
<td>Off-chip memory, such as external flash, may be easily accessible to a locally present attacker, and may be accessible during manufacturing or deployment.</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>On-chip* memory + trusted off-chip RAM</td>
<td>Off-chip storage, such as flash, may be accessible to a locally present attacker or an external system, such as a Baseboard Management Controller (BMC), during manufacturing or deployment.</td>
</tr>
</tbody>
</table>

[*] Arm recommends to use on-chip RAM. However, SRAM can be used on a separate die, if it is within the same package as the main SoC.

PSA-TBFU requirements focus on protecting easily-accessible interfaces and supporting software countermeasures against timing issues.

Advanced hardware invasive attacks, in which the attacker has access to laboratory equipment that probes on to silicon metal layers, infers fuse settings, or performs differential power analysis, are out of scope for this architecture.

A certain class of attacker may be able to read the contents of off-chip RAM given physical access to the system and appropriate tools. This may involve an attacker that can freeze the memory to manually read the contents or by replacing the memory with a more advanced component that persists data after the system has been turned off. If a security objective needs to address this then all off-chip memory must be considered untrusted.

For information about the requirements for trusted peripherals see Section 2.1.
2.3 Image manifest

This document requires digitally signed image metadata for firmware images, which are referred to as manifests. If a system is sufficiently capable of processing certificate chains using the X.509 standard, such as an A-class processor, then all instances of manifest in this document may use X.509 content certificates as a manifest. Moreover, the X.509 standard is well documented, and free tools and source code are available. Parsing of encoded formats can introduce complexity that may outweigh the need for interoperability.

3 Requirements

To be compliant with this specification, the rules specified in this section must be met. Rules are denoted with rule IDs and highlighted in blue. The surrounding text provides examples and rationale. Subsections titled with “optional” contain optional functionality that may increase robustness of implementations.

This specification refers to the SPE and NSPE when describing roles and responsibilities of components in the firmware update process. It is possible for the SPE and NSPE to be executing on separate CPUs within the same SoC, providing that the NSPE cannot interfere with SPE resources and execution.

The boot and firmware update cycle can be summarized in five steps:

1. The first trusted boot code is a bootloader that is embedded in a boot ROM, a write-protectable equivalent, or a security processor. This is referred to as the immutable bootloader in this specification. The immutable bootloader is hardware Root of Trust that executes from reset, containing the minimal functionality required to check the authenticity of the PRoT software. It may also load and verify the ARoT and NSPE software. The entire trusted boot sequence may consist of multiple stages, each of which must be authenticated before execution. It ensures that any unauthorized firmware modification can be securely identified and prevent persistent malware from residing on the system.

2. The SPE initializes different Roots of Trust for the system as well as generic security services, known as Application Root of Trust. It may also provide bootloader functionality, such as backup and test features.

3. The NSPE receives firmware updates Over The Air (OTA) or from a local peripheral.

4. A runtime service or a bootloader within the SPE will check the update authenticity against a local public key and against a policy to see if it should be installed. If permitted to be updated, then the SPE will provision the update to storage, which is typically done by a bootloader on reset.

Depending on system resources and supply chain factors, the SPE may consist of multiple images, such as a mutable bootloader, runtime services and recovery software. Multiple bootloaders may also be used to separate board-specific code from chip-specific code.

3.1 Approved cryptographic algorithms

This section describes the approved cryptographic algorithms and practices. The specific purpose of the cryptography, such as a Root of Trust Public Key (ROTPK) are described in Section 3.2. Cryptographic best practice demands that the same key is not used for multiple purposes.

This document requires firmware validation to use one of the approved public key cryptography algorithms. The choice between using RSA-PSS, ECDSA or EdDSA for cryptographic signatures may be made for patent related reasons, hardware cost, signature size and time required for authentication. This specification permits the use of any of these. The use of ECC for asymmetric cryptography is often beneficial because its smaller key sizes lessens storage and transmission requirements. System architects should also review the comparative performance of implementations in terms of throughput for each of the relevant key use cases. It should be noted that the speed of a signing operation may significantly differ from the speed of a signature verification operation. For this specification, only the latter is required for trusted boot and update. Since the image signing
process is likely to be done in an offline environment and on a more capable machine, implementers may favor speed of signature verification since this can significantly affect boot time.

3.1.1 RSA
The RSA-PSS signature scheme signs a hash of the message instead of the message directly. This technique is often used with RSA because the amount of data that can be directly signed is proportional to the size of the keys, which is usually much smaller than the original message.
A digital signature produced using RSA-PSS is the same length as the RSA key modulus, and so Arm considers signatures to be the same length as keys.
For RSA signatures, it is recommended that the RSA-PSS signature scheme is used.
In the absence of a compliance profile, it is recommended to use the key size defined in the TBSA documents [2].

3.1.2 ECC
Any use of ECDSA may want to follow the FIPS 186-4 standard [FIPS PUB 186-4]. The Commercial National Security Algorithm (CNSA) Suite recommends the P-384 curve as a parameter. Alternatively, the Standards for Efficient Cryptography Group (SECG) also provide recommendations for curves and parameters.
In the absence of a compliance profile, it is recommended to use the key size defined in the TBSA documents [2].

3.1.3 EdDSA
Any use of EdDSA should conform to IETF RFC 8032 [RFC 8032].
In the absence of a compliance profile, it is recommended to use the key size defined in the TBSA documents [2].

3.1.4 Hashing
In the absence of a compliance profile, all cryptographic hashes in this document must use either:
- Secure Hash Algorithm 2 (at least 256 bits). Systems that are expected to be in the field for a long time are recommended to use the stronger algorithm SHA-384, which is suggested by the CNSA suite.
- SHA-3 (at least 256 bits)

3.1.5 Key derivation
For Key Derivation Functions (KDF) it is recommended to follow NIST’s recommendations as specified in NIST 800-108 [8].

3.1.6 Side channels
Shared secrets that are used for encryption and decryption can be vulnerable to side-channel key-recovery attacks. Public key authentication, as required by this specification, is unaffected because no secret is required to authenticate firmware. However, ensuring confidentiality of assets remains an issue.
Cryptographic algorithms can be implemented with hardware dedicated engines or completely in software. Both might be protected against non-invasive side-channel attacks. The software implementation of the cryptographic primitives may want to ensure that their execution have the same processing time and cache footprint for every code path in the cryptographic algorithm.

3.2 Chain of Trust
This section describes the use of cryptography for boot and for firmware update.
This specification assumes that two immutable keys exist, which must be in accordance to the rules described in Section 4.2:

- a Root of Trust Public Key (ROTPK) that is responsible for securely authenticating the first stage of code using public key cryptography. It may also be used to verify certificates of delegated signing keys.
- a Hardware Unique Key (HUK) that is used for symmetric cryptography (AES, MAC, etc.) and is unique to the system. It can be used for secure storage and for local authenticated encryption. Since this key is immutable it must only be accessible to trusted software.

These keys may be provided by a trusted subsystem.

Either the ROTPK hash or the ROTPK itself must be in an immutable part of the SoC NVM. A hash is often preferable because it requires less space in immutable storage.

Multiple ROTPKs may be included for different vendors in the supply chain. For instance, the SoC vendor may be able to securely boot their own code using an in-built ROTPK, and then verify OEM firmware using a separate OEM ROTPK (see Section 3.2.3). The OEM ROTPK may be provisioned at a different point in the supply chain, where there may be operational processes in place to reduce risks in the provisioning process. The specific ways of provisioning ROTPKs are outside the scope of this specification.

**R10_TBFU_KEYS:** The system must include at least one firmware verification public key known as a Root of Trust Public Key (ROTPK) and a Hardware Unique Key (HUK)

The private key associated with the immutable ROTPK cannot be revoked. This private key may be used to sign an OEM generic second stage boot loader or SPM, which will perform board-specific configuration and continue the trusted boot process. As such the ROTPK private key is hardly used (a few times per OEM) and can be kept in a highly secure Hardware Security Module (HSM).

A product analysis must determine the number of potential actors that will maintain the components of the product. Examples include:

- A Silicon Provider (SP) that could customize and add SoC-specific code before the boot process begins
- A manufacturer who adds board-specific customizations based on product requirements (for example, OEM firmware and management software)
- An operator (or owner) may configure an update policy and device identity on the system before deployment

**R20_TBFU_KEYS:** Each ROTPK must be immutable. These can be stored using a locked on-chip flash sector, a secure element, or on-chip OTP memory. It is permitted to store an immutable hash of each ROTPK to check the integrity of ROTPKs in untrusted storage.

### 3.2.1 Single provider

Figure 3 shows the simplest case, such as a constrained IoT device, where a manufacturer controls the entire software stack and signing process. The manufacturer provides the ROTPK. No revocation is possible in this scenario.
If the system has multiple software providers and has sufficient computational ability to validate a certificate chain, then it is recommended that separate signing keys are created to mitigate any loss of a private image signing key. This is discussed in the following sections.

### 3.2.2 Multiple dependent providers

The first link in the chain of trust is the management of the Root of Trust public key, which is used to verify all subsequent certificates and images. An example scenario is the following:

- The OEM provides their own ROTPK.
- The OEM signs a certificate belonging to a software vendor’s credentials. There can be multiple software vendors.
- The software vendor uses their certified credentials to sign:
  - Production image signing credentials (unique for a particular model line)
  - Debug image signing credentials (unique for a particular model line)
- The SoC ships with the OEM ROTPK provisioned into OTP memory

If SPE and NSPE are separate systems with separate images, then Arm recommends signing these images with separate signing keys. If SPE and NSPE come from the same vendor, then the signing key for the SPE should have more levels of operational protection. Since the SPE must provide secure services to the NSPE, it must be more rigorously tested and should not require frequent updates.

### 3.2.3 Multiple independent providers

The system may include enough on-chip storage to hold multiple discrete ROTPKs. Each ROTPK provides an independent chain of trust, which allows for different manufacturers to authorize and revoke firmware signing keys independently of each other. For instance, one ROTPK can correspond to the SoC vendor whereas another may correspond to the OEM. Alternatively, an OEM owns one ROTPK while allowing a customer to install a separate ROTPK for NSPE firmware. Figure 5 shows a simplified chain of trust for independent providers.
The NSPE ROTPK (or the hash used to identify it) can be stored in one of the following permitted implementations:

- On-chip OTP memory
- A secure element or security processor
- Secure storage that is only writeable by the SPE. Any update of that key after provisioning must either:
  - Be rejected by the SPE (one time provisioning)
  - Be replaced with a newer key only if it is signed by the currently stored NSPE ROTPK

**Implementation note**

Sufficient OTP space may allow for two truncated ROTPK cryptographic hashes to be stored. If this is desired, then it must conform to NIST’s recommendations as specified in NIST 800-107 [9]. For example, it is common for the most significant 128 bits from a SHA-256 digest to be used when truncated.

### 3.2.4 Certificate creation and key management

It is recommended that for the creation of certificates for use in production and development devices, users should consider appropriate processes for the handling, and management of keys used in the signing process, that should consider the physical security and storage of keys, controlled access to those keys, auditing of access and so on. A description of such a process is outside of the scope of this document but is a significant security issue that must be considered as part of the implementation of this document.

### 3.3 Image verification

The trusted boot process works by authenticating a series of cryptographically signed binary images each containing a different stage or element in the system to be loaded and executed. Each image has either a lightweight manifest or a certificate, which is authenticated by a public key. This public key must be traced back to the ROTPK.

At the point where a new image is to be installed the associated signature must first be verified against a public key. Signature verification may be computationally expensive to perform on each boot. A large number of components may have a significant effect on the boot time. An implementation can optimize the trusted boot
process at the expense of simplicity: once an image manifest has been successfully verified against a public key, it is permitted for an implementation to use one of the following mechanisms to speed up subsequent boots:

- The image manifest can be locally authenticated with a Message Authentication Code (MAC). In this specification this process is referred to as “rekeying”. This avoids having to perform an asymmetric signature check again on subsequent boots of the same image and therefore can speed up the boot process. It is recommended that the key of the MAC be derived from the HUK using a KDF. It is then safe for the manifest and the MAC to be placed in untrusted storage.
- The manifest of the verified image can be placed in on-chip storage that is write protected from the next stage of components. On subsequent reboots the calculated hash of the image can then be directly compared with the expected hash within the stored manifest without re-authenticating the manifest.

R10_TBFU_EXEC: Any use of a MAC to re-key and authenticate a firmware image manifest must be in the form of a HMAC, CMAC, or GMAC signature. The key may either be a key in trusted memory or in a trusted subsystem.

R20_TBFU_EXEC: Any caching of authenticated manifests must be held in trusted memory and write protected from untrusted components.

Memory may be shared between components at different stages of the trusted boot process. Secrets that are resident in memory may need to be removed from memory.

R30_TBFU_EXEC: Secrets used by a trusted component X must be scrubbed from volatile memory and registers before ownership of the memory is transferred to a component not trusted by X.

Scrubbing a memory location can mean any of the following:

- Overwritten with a pre-defined constant value (for example, zero)
- Overwritten with a random value
- Indirectly changed to a random value, for example by changing a key which is used to decrypt the memory contents

To prevent interference from code running on other processors, and to eliminate time-of-check-time-of-use (TOCTOU) exploits, concurrent execution must be disabled when validating images. Therefore:

R40_TBFU_EXEC: Each loaded SPE image must be verified in trusted memory before execution. It is permitted for NSPE images to be loaded into untrusted memory.

By loading components from untrusted storage to trusted memory prior to cryptographic validation, it is not possible to bypass verification with an authorized copy of the firmware and then substitute an unauthorized version at runtime.

R50_TBFU_EXEC: The boot process must be uninterruptible during signature verification to prevent race conditions.

When signatures fail to verify or a roll back attack is detected, then this event is considered a security violation. The component that causes the security violation must not be executed. If the component is critical to the system functionality then a recovery mode might be entered, or the system might log diagnostics and reboot depending on the severity of the violation.
R60_TBFU_EXEC: The boot process must deny execution of a component if a security violation occurs.

Updating an image in persistent storage can be a complex process. Care must be taken to avoid partial or undefined system states.

R70_TBFU_EXEC: The update process must be an atomic operation. If interrupted, then the update process must either revert to the prior state or enter a recovery mode.

Recovery modes are implementation defined. As an example, the mode may restore a stored backup of the image or listen on a wired interface awaiting a new image. The restored image must still be subject to all the rules defined in this specification, particularly anti-rollback detection.

R80_TBFU_EXEC: If the NSPE is to be executed on a secondary processor, the secondary processor must be kept in reset or halted until the NSPE firmware has been verified by the SPE.

This rule includes implementations with a secondary processor that has minimal ROM code waiting for an SPE notification.

If the SPE and the NSPE share compute resources, such as CPU or memory, then isolation mechanisms must be configured before the NSPE executes. For instance, an isolation mechanism may be one of or a combination of Address Space Controllers, Security Attribution Units, Peripheral Protection Units, Memory Protection Units, etc.

R85_TBFU_EXEC: Isolation mechanisms must be correctly configured before the NSPE begins execution.

Upon successful verification of a manifest, the SPE might create a Message Authentication Code (either a HMAC, GMAC or CMAC) for that image using a key derived from the HUK. This avoids having to perform asymmetric cryptography again on subsequent boots for the same image.

R90_TBFU_EXEC: Any MAC operation used for re-keying firmware manifests must only be performed within the SPE.

Direct Memory Access (DMA) is a common way for external peripherals to transfer data to and from SoC memory. It is possible for an external peripheral to be replaced or exploited to interfere with the trusted boot process. Some SoCs include special functionality for restricting DMA transactions to specific memory regions, ensuring that trusted components are integrity protected during the boot process.

R95_TBFU_EXEC: Any available I/O protection mechanisms must be enabled to integrity protect loaded images from untrusted peripherals.

An example of an I/O protection mechanism is a System Memory Management Unit (SMMU) or a Peripheral Protection Unit (PPU).

3.3.1 Secure storage

The following requirements address problems when handling images and authenticated data in untrusted memory.

SPE firmware must not be executed directly from external flash memory. Instead it must either be copied into trusted RAM and verified and executed from there or executed in-place in secure internal flash memory.

R10_TBFU_STORAGE: An image in untrusted memory must be copied to trusted memory before authentication.
R20_TBFU_STORAGE: When an image in untrusted memory is copied into trusted memory, it must be integrity checked after the copy has completed. The integrity check must match the authentication data for the image.

R30_TBFU_STORAGE: Authentication data used to verify SPE images must be in trusted memory before use.

R40_TBFU_STORAGE: Encrypted images in untrusted memory must be decrypted into trusted memory and authenticated in trusted memory.

R50_TBFU_STORAGE: Images and data in trusted memory must be signed using a key derived from the HUK before being copied to untrusted memory. The key must never leave trusted memory.

Arm recommends that SPE images be protected from NSPE access. This can be achieved by providing the SPE with exclusive access to some on-chip or off-chip non-volatile storage.

### 3.4 Trusted Boot Flow

When the device is powered on, the SoC automatically executes from the start address (such as the reset vector). This start address may point to the first location of internal flash’s address space or to masked ROM. The code at this start address must be considered immutable once provisioned in the factory.

The verification and execution of the firmware needs to be performed in a specific ordered sequence during boot. At a high level, the immutable bootloader code must verify the mutable secure software. This ensures that low-level software is not compromised.

The immutable bootloader must be implicitly trusted. Its purpose is to load and validate the first PROT image using the ROTPK:

- In a single stage boot process, the immutable bootloader can choose to simultaneously validate and load both the Application Root of Trust (ARoT) and the NSPE application.
- In a multi stage boot process, there may be numerous loaders, each of which uses keys to verify and load other SPE images or NSPE images.

An implementation must decide how many boot stages will be needed based on product requirements. Regardless of the number of stages, each boot stage must authenticate all the software it loads.
The most constrained system may only contain a single stage bootloader within locked flash. Figure 6 shows the boot flow for this type of constrained device. The boot state is described in Section 3.8.

It is common for a bootloader to be divided into several stages, the first of which is the immutable bootloader. The latter stages might be loaded from non-volatile storage into Secure RAM and executed there or executed directly from eFlash. Splitting the bootloader into two stages, immutable and mutable, has some advantages:

- It minimizes the risk of problems in the immutable code, as it allows for updates or errata to be handled by the mutable stage at device provisioning time or later. As an example, image backup and recovery functionality can be non-trivial to implement in code for certain storage types and is likely to require bug fixes after product release.
- It separates concerns between silicon vendor and board maker, who may be different parties. For instance, the SoC manufacturer may provide code for basic initialization and protection of certain assets whereas the Original Equipment Manufacturer (OEM) may customize board specific features, such as external flash and advanced recovery capabilities. In this scenario, the SoC manufacturer’s code would authenticate the OEM’s initial code.

At each step in the boot chain, each stage must verify the next, and verification of an image is based on a combination of hashing and asymmetric cryptography. Since asymmetric cryptographic algorithms are CPU-intensive, hardware acceleration can be employed. For instance, the RSA cryptosystem may be accelerated using a Montgomery Multiplier hardware unit. Where possible, image authentication should be done using any available secure elements or accelerators to reduce memory footprint and accelerate the boot process.

Images can be encrypted to hinder reverse-engineering efforts from attackers, however this is not a requirement for the trusted boot process.

The immutable bootloader is the beginning of the “chain of trust” and must be considered immutable and irreplaceable. All other images must be updatable.
It is possible for the boot sequence to fail at any stage due to a faulty component or restricted functionality. If the signature of a component fails verification, then the platform must not execute that component. The platform may want to perform one or more of the following actions:

- log the event
- recover the component
- reset the system
- load other components

### 3.4.1 PRoT immutable bootloader requirements

The first code of the trusted boot process is an immutable bootloader placed in a boot ROM or a locked eFlash sector that can emulate a boot ROM. An example of an emulated boot ROM would be an area of flash with an OTP lock that permanently disables write and erase accesses to the flash area, while also disabling debug access (e.g. JTAG/SWD).

The purpose of the immutable bootloader should be solely to validate the next stage against the ROTPK. The next stage is expected to be a second stage bootloader or a SPM, which provide richer functionality. As a minimum, the hash of the ROTPK is either included embedded in the bootloader or provisioned into OTP NVM. The hash must be used to verify the integrity of the full public key, which might be included in the mutable next stage image.

Code that performs more complex functions will naturally have a greater attack surface or risk of bugs. Therefore, the immutable bootloader should only contain the flash and cryptographic primitives necessary to read and validate software. Additional functionality should be a part of the SPE software, either another bootloader or a SPM.

The immutable bootloader can be considered a part of the initial Root of Trust (iRoT) according to the GlobalPlatform definition [11].

The following rules are defined for the immutable bootloader:

**R10_TBFU_BOOT:** The immutable bootloader must verify the PRoT before booting it. If image validation does not succeed, then the image must not be executed.

**R20_TBFU_BOOT:** The immutable bootloader must verify loaded images using the ROTPK itself or a delegated key. It is permitted for the immutable bootloader to only verify a single firmware image containing all remaining verification functionality.

**Implementation note:**

It is advised that most ROM functionality is placed into a loadable module to ensure that there is never a need for the boot ROM itself to be updated. This module can be verified using RSA/ECC authentication or using a hash in on-chip OTP memory. A silicon provider may use this to deliver errata to the SoC and to defer certain ROM functionality to a later stage of manufacturing.

**R30_TBFU_BOOT:** The immutable bootloader must compare the computed hash of the ROTPK with the immutable ROTPK hash.

Booting must initially be handled by an immutable bootloader that is held in on-chip boot ROM (or eFlash securely emulating a boot ROM, which must be locked after being provisioned during manufacture). A trusted
on-chip security module might be used to ensure the integrity of the immutable bootloader, either by acting as a boot ROM or by checking the integrity of the bootloader in trusted memory during power-on-reset.

When using locked eFlash instead of ROM, the immutable bootloader must be placed at the reset vector to ensure that this stage always boots first. This may be at the beginning of the address space on some platforms. The regions must be configured in such a way that they are protected from program and erase operations.

The immutable bootloader must be able to read an immutable hash of the ROTPK (or the ROTPK itself) in on-chip memory. Typically, there are a number of ways to achieve this:

- The ROTPK or its hash might be embedded in the immutable bootloader itself. This is the least flexible option because the ROTPK owner may be a different party to the owner of the immutable bootloader.
- The full ROTPK might be provided with an SPE image. In this case, the immutable bootloader must calculate the hash of it and compare it against an embedded provisioned hash in immutable memory before using it.
- The ROTPK or its hash might be stored in separate on-chip OTP NVM (i.e. eFuse) or a flash sector (that was locked during manufacturing).
- The ROTPK or its hash might be provided by a Trusted Subsystem.

The immutable bootloader itself may execute within a Trusted Subsystem provided that the Trusted Subsystem is on-chip and has sufficient capability to verify the SPE software while the main cores of the processor are held in reset.

It is recommended that the bootloader stages are only readable during the trusted boot process to prevent reverse engineering and code re-use attacks.

Manufacturers may want confidentiality of bootloader secrets after initialization. Hiding the bootloader after the boot process requires a non-reversible mechanism, for example a sticky register bit that is activated by the boot software. This is not required in this specification.

If a warm reset is supported, then the immutable bootloader must be able to determine whether the system performed a warm reset or a cold reset. This may be achieved by checking a variety of platform specific registers. It is recommended that the immutable bootloader enables any available watchdog timers as soon as possible before the next stage to reduce the risk of tampered memory from a physical attack.

**R40_TBFU_BOOT:** The immutable bootloader must disable debugging functionality, such as JTAG or SWD, subject to the PROT device lifecycle state.

An advanced attacker with access to the device and specialized equipment may attempt to analyze the boot process before performing a non-invasive timing attack. Such a non-invasive timing attack might introduce intentional faults using clock, power or thermal means, which could result in instruction skipping, decoding errors, or malformed data accesses. Attackers may use these side-effects to target specific areas of code execution to bypass authentication checks during the trusted boot process. Since these attacks rely on reliable and precise timing, early stage firmware may employ an on-chip hardware random number generator to randomize the execution time of boot, making timing attacks infeasible for adversaries.

### 3.4.2 PROT runtime requirements

The implementation of the PROT runtime software may include one or more of the following:

- Another bootloader that is capable of initializing more complex devices such as USB or SPI.
- An SPM or Trusted OS that provide secure services to application firmware
R50_TBFU_BOOT: All mutable SPE code must be updatable. The update may be performed by the SPE code itself or by a prior boot stage.

It is recommended to provide a firmware update image staging area (for download and verification) which does not overlay the current image. The staging area may be trusted or untrusted memory.

Some platforms provide a register for programming an address for the processor to jump to on reset. This must be protected from untrusted code.

R60_TBFU_BOOT: If the platform has a programmable reset address, then the PRoT must protect this from the ARoT and the NSPE. This may be achieved using a locking or memory protection mechanism.

Firmware must only be executed after it has been successfully authenticated.
Images in untrusted memory might be decrypted into trusted memory using a key derived from the HUK.
The SPE might use public keys available in a secure element.
The SPE might need to include the ROTPK with its image if the bootloader only includes the hash of the ROTPK.

An SoC may contain multiple isolated execution environments. For instance, Arm TrustZone security extensions may be available for propagating the current security state to each SoC component. Alternatively, there may be multiple systems within an SoC, where one isolated core contains security critical firmware while another core executes a feature rich application. These are respectively referred to as the SPE and NSPE. In this case, it is recommended that these two systems should be signed with different public keys (PKs), namely an SPE-PK and a NSPE-PK.

Implementation note:

It is permitted for the SPE to re-use available bootloader code due to storage constraints. If the only bootloader is the immutable bootloader, then this is not recommended because the immutable bootloader code cannot be remotely updated in the event of a bug or security vulnerability.

3.5 Update process (informational)

Some devices require protection against failure of a new image by retention of a known good image, normally the current image. This implies sufficient NVM to store two images. The simplest case is when both images might be stored on the device in eFlash, in which case the eFlash has to be dimensioned for two image slots, a primary slot and a secondary slot. The same principle can be applied for external flash. Following the download and processing of a new image the update client of the device is responsible for programming the new image into the secondary slot and arranging for the device to be rebooted. Images that have been provisioned to storage are known as candidate images.

Firmware updates are composed of a manifest and an image. A manifest contains meta-data about the firmware image and is protected against modification. The firmware image is a binary that may contain the complete software of a device or a subset of it.

A firmware update may consist of multiple images for certain types of systems. A firmware image may consist of a differential update for performance and bandwidth reasons.
The update process might fetch images from an external interface such as USB, UART, SD-MMC, NAND, NOR, Ethernet to SoC NVM memories such as NAND Flash, LPPDR2-NVM or any memory as indicated by SoC inputs pins.

The update procedure may consist of the following stages:

1. Fetching signed manifests and their corresponding firmware images
2. Authenticating the firmware images using the manifests to check their provenance and integrity
3. Authorizing updates against a device security policy
4. Installing the images into persistent storage

A high availability use case may require candidate SPE images to be protected from the NSPE at all times. This ensures that certain images cannot be overwritten. This can be achieved if the SPE configures the appropriate hardware isolation mechanisms correctly.

A device security policy is used by an updater to authorize the firmware update. It may include information about:

- which public keys should be used to verify the image
- whether the update is targeted for that specific device (based on device identity)
- whether the device has enough power to perform all the required steps
- rate limiting the frequency of updates in order to prevent premature write exhaustion of flash

There may be restrictions set by any installed device management software.

An update server holding firmware images should not be trusted for providing any security. The only data the updater needs to trust is manifest information signed by a trusted authority. Therefore, image updates can be propagated in different fashions. For instance, in a mesh topology of IoT or edge devices it may be convenient to distribute and host image updates via a local available gateway or through a decentralized distribution mechanism. Similarly, a network service provider may have an elaborate and dynamic content distribution network for provisioning updates to different regions resulting in fast updates and less network pressure.

### 3.6 Image manifests

A signed manifest must be used to convey information about an image from the developer. This helps devices to inter-operate with different boot images and does not require any alteration of image formats. Firmware images are always integrity protected but may also be encrypted.

**R10_TBFU_MANIFEST:** Each image must be associated with a signed manifest. It is permitted for the manifest to be appended to the image itself. It is permitted for the image and manifest to be concatenated and then signed together.

Implementations may wish to transfer manifests in encrypted form to provide privacy for system updates.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manifest format version</td>
<td>Version of the manifest format used</td>
</tr>
<tr>
<td>Signer ID</td>
<td>Identifies the signing key for the image. This must be the hash of the signer’s public key. This field is not required if there is only one signer for the device software.</td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image hash algorithm</td>
<td>The cryptographic algorithm used to calculate the image hash value. This field is not required if only one algorithm is supported.</td>
</tr>
<tr>
<td>Image hash value</td>
<td>The expected hash value of the image. If the image is encrypted, then it is the hash value of the ciphertext.</td>
</tr>
<tr>
<td>Image size</td>
<td>The image size in bytes</td>
</tr>
<tr>
<td>Image type</td>
<td>Specifically identifies the boot stage or peripheral. Also known as a purpose identifier. This field is not required if only one type of image is supported.</td>
</tr>
<tr>
<td>Product class</td>
<td>This can refer to a vendor ID, product ID, or a more specific identifier.</td>
</tr>
<tr>
<td>Image version</td>
<td>The image version, which is used to prevent rollback attacks. The specific versioning scheme is implementation defined. An implementation may separate image versioning from security versioning for deployment flexibility.</td>
</tr>
<tr>
<td>Manifest signature algorithm</td>
<td>Identifies the digital signature algorithm used for the “Manifest signature” field. This field is not required if only one algorithm is supported.</td>
</tr>
<tr>
<td>Manifest signature</td>
<td>Digital signature that signs all the manifest fields. The manifest may be signed using an industry-standard container format, Cryptographic Message Syntax (CMS), also known as PKCS-7. It is recommended that a the manifest format supports multiple signatures to support counter signing.</td>
</tr>
</tbody>
</table>

**Table 2: Required manifest attributes**

More attributes can be added to the manifest to fit different use cases. The following contains a non-exhaustive list of additional attributes:

- The full public key must also be included (instead of the Signer ID) if the full key is not provisioned on the device.
- If a firmware image has multiple dependencies, then a manifest can specify multiple images. For instance, a device may be a collection of different processing units, each of which requires specific firmware versions.
- A set of conditions may also be included, such as a pre-condition or a post-condition.
- A value (in seconds) specifying the maximum time the firmware update process should take to flash
- Firmware storage location in terms of absolute or relative addressing
- A device ID for special images that may be created for a specific platform instance. This may be necessary to provide specialized builds for specific customers, which should not be transferable to other platforms not owned by the customer.
- A nonce may be included to make the manifest unique.

**R20_TBFU_MANIFEST**: Each image manifest must contain the properties specified in Table 2. It is permitted for additional information to be included.

The serialization format of a manifest may benefit from Concise Binary Object Representation (CBOR) or Abstract Syntax Notation One (ASN.1) encoding, depending on processing or interoperability requirements.
For constrained devices, such as microcontrollers, Arm recommends using the IETF SUIT manifest format for signed manifests [12]. An example manifest is provided in the Appendix of this specification.

For capable devices, such as application processors, Arm recommends the X.509v3 content certificate format for signed manifests. This enables interoperability with existing tools and software stacks. Additional fields may be specified using X.509v3 extensions. An example manifest is provided in the Appendix of this specification. Parsing of encoded formats can introduce complexity that outweighs the need for interoperability with popular formats.

R40_TBFU_MANIFEST: For delta updates, the image manifest must also contain the hash of the final expected image state.

A system may have many modules that require updating individually. It may also need to trust several different actors to authorize an update. For example, a firmware author may not have the authority to install firmware on critical infrastructure without the authorization of an operator. In this case, the firmware should reject firmware updates unless they are signed both by the firmware author and by the operator.

R50_TBFU_MANIFEST: All data in the image manifest must be digitally signed using asymmetric cryptography.

When encryption is desired there are three options for decryption keys:

- A full pre-shared symmetric key, which must be provisioned securely
- A symmetric key derived using a key derivation function (KDF), which is based on a small pre-shared secret
- An asymmetrically encrypted symmetric key, provisioned dynamically with the encrypted payload

R70_TBFU_MANIFEST: Image updates that include security enhancements or vulnerability fixes must increase the software version when signing the manifest.

The image signer of the NSPE may decide to disable trusted boot of the NSPE either at factory provisioning stage or at a later point in the product lifecycle.

R80_TBFU_MANIFEST: If the NSPE-PK or NSPE image hash are equal to zero, it means that the NSPE can be freely replaced on the SoC.

3.7 Anti-rollback protection

If a firmware image is updated to fix security vulnerabilities, and the device permits the firmware image to be “rolled-back” to a previous, insecure version, then a security risk exists. Therefore, firmware must use non-volatile (NV) version counters to protect against rollback.

For validating untrusted software, each trusted boot stage must use one of the following mechanisms for providing anti-rollback protection for images:

- **Secure on-chip NVM**: Counters can be implemented in on-chip storage only accessible to the SPE.
- **On-chip OTP memory**: Counters can be implemented using on-chip OTP memory if enough individual fuses are available. Care should be taken to ensure that enough updates can be supported for the lifetime of the product.
- **Trusted subsystem**: Counters may be provided by a trusted subsystem. If the trusted subsystem is off-chip then the trusted subsystem must support an MTP counter that is cryptographically paired with the
SoC. The pairing is required in case an attacker replaces the trusted subsystem with one that contains lower counter values.

- **Authenticated off-chip memory**: Implementations may use a single on-chip counter to secure off-chip version counters. This can be achieved by storing an authenticated table of counters in untrusted storage. The table must be authenticated using a key derived from the HUK (to prevent cloning) and combined with the on-chip counter (for replay protection). The on-chip counter must be incremented on every update to the table.

Counters must never be updatable to a value less than their current value. A counter must never overflow. If the maximum value is reached it must remain at that value.

**R10_TBFU_ROLLBACK**: Only images of a higher version or the same version can be installed.

After a firmware image is verified, the image version number taken from the signed manifest and is compared with the corresponding stored counter. If the value is:

- Less than the NV counter then the authentication fails.
- Identical to the NV counter then the authentication is successful.
- More than the NV counter then the authentication is successful, and the NV counter is updated with the higher value.

It is recommended to implement as many version counters as there are images, where each image can use a separate counter without affecting other images. However, the number of rollback counters that can practically be supported is implementation dependent.

**R20_TBFU_ROLLBACK**: Each software stage must not be able to decrease their corresponding rollback counter. It is permitted for the PRoT to be exempt from this rule if it is responsible for preserving the integrity of counters.

**R30_TBFU_ROLLBACK**: Rollback counters must be implemented either with on-chip OTP memory, a trusted subsystem, or a private on-chip NVM region which is only write-accessible to the SPE.

Rollback counters might also be required to support version control of other software. A suitable implementation might employ one counter per software instance, or group together a list of version numbers inside a database file, which is itself versioned using a single counter.

**R40_TBFU_ROLLBACK**: Rollback counters must never overflow. If the maximum value is reached it must remain at that value.

If a rollback counter is implemented using on-chip OTP memory, such as eFuses, then a lower bound on the number of supported updates must be specified.

**R50_TBFU_ROLLBACK**: Each rollback counter used to validate SPE software must support at least 64 values. If the SPE consists of multiple boot stages, then it is recommended that each stage has a dedicated counter for each verification step.

A rollback counter is increased when newer software is loaded. In some markets it can be desirable to perform a boot test of the image before increasing the version counter. One simple example of a boot test mechanism is a watchdog that tests whether an update is unresponsive. A more complex example may involve testing the network stack or update system to ensure that network connectivity has not been broken.
R60_TBFU_ROLLBACK: If the counter value in the manifest is greater than the rollback counter, and if the manifest is authentic, then the rollback counter must eventually be increased to match the counter value in the manifest.

It is implementation defined when the rollback counter is increased:

- An implementation can decide to perform a boot test of a new image before incrementing the rollback counter. When implementing staged update with failure rollback, counters must only be updated when the installed image has been successfully tested.
- An implementation can decide to support a mechanism to control anti-rollback protection by remote messaging. However, it must not be possible to use any remote management feature to increase an anti-rollback counter on a device to a value beyond the highest version of any images currently loaded on the device. Such a mechanism requires:
  - Authentication of the command issuer with at least the same cryptographic properties as that used for image signing.
  - Replay protection, ensuring that any issued command instance can only be acted on only once by a device.

It is implementation defined when a rollback counter is reset:

- An implementation can decide to reset the anti-rollback mechanism following factory reset. This allows devices to be recovered if the anti-rollback mechanism becomes desynchronized with the signer.
- An implementation can decide to use a remote messaging protocol as described above.

R70_TBFU_ROLLBACK: Any implemented mechanism to reset rollback protection must be at least as secure as the image signing mechanism

3.8 Measured boot and attestation

An SoC may need to prove the integrity of its software to a remote party or to local systems on the same board. This can be achieved if the SoC has a unique cryptographic identity and is able to measure itself in a way that it cannot spoof measurements. The identity must be strong enough so that it cannot be reasonably forged.

A prerequisite for providing attestation is to preserve all the calculated measurements from all loading stages. This is known as a measured boot. Any measured boot mechanism must assure the integrity of such firmware and make it part of an overall chain of trust. Each stage of the chain of trust must accurately and robustly measure all code that will be loaded. This includes:

- Loadable modules (including dynamic patches, OptionROMs)
- Parameters that influence boot behavior (For example, flags or variables that may change the control flow of the loaded program)

Each stage of the chain of trust must store the measurements in a secure manner. The measurements may be held in a security module or in protected PRoT memory. A remote party can use the list of measurements to help validate the specific software identity of the platform.

After successful validation, the immutable bootloader must store boot state for use by later PRoT runtime services, such as attestation and binding. Part of the boot state include a freshly generated number called a boot seed. The boot seed may be used by later services, for example to allow a validating entity to ensure that attestations for different attestation end points were generated in the same boot session. It must be large enough to make global collisions statistically improbable.
R10_TBFU_ATTEST: On each reset, the immutable bootloader must create a boot state.

R20_TBFU_ATTEST: The boot state must include calculated cryptographic measurements of loaded boot image components prior to execution. The immutable bootloader may also measure the installed NSPE image, if applicable.

A platform may have one or more trusted subsystems, such as a Secure Element, Trusted Platform Module, or a cryptographic accelerator, for example. They may include updateable firmware, which must be measured.

R30_TBFU_ATTEST: Any updatable components of trusted subsystems must be measured and verified at boot. The measurements must be included in the boot state.

The boot state must include a randomly generated number called a boot seed.

R40_TBFU_ATTEST: The boot state must include a randomly generated boot seed. It is permitted for the boot seed to be generated by the PRoT runtime software and added to the boot state once boot has completed.

It should be noted that the boot seed is separate and unrelated to any challenge or nonce provided by a remote party during an attestation protocol. It must be statistically improbable, which can be met with 256 random bits.

R50_TBFU_ATTEST: The boot seed must be 256-bits in size.

The boot state must be accessible to the PRoT software. This may be stored in on-chip NVM or on-chip RAM. The specific format and way to pass information is considered implementation-defined.

R60_TBFU_ATTEST: The boot state must be stored in an on-chip memory area, which is only accessible to the PRoT. It is also permitted for the boot state to be stored within a trusted subsystem.

Implementation note:
A trusted subsystem, such as a Trusted Platform Module (TPM), can store measurements made by the boot software. With a TPM this is achieved by programming the TPM’s Platform Configuration Registers (PCRs) using the “PCR extend” TPM operation. The convention for using PCRs is specific to the device class and being standardized by the Trusted Computing Group (TCG).

For implementations based on the Unified Extensible Firmware Interface (UEFI) the convention for using TPM PCRs is described in the TCG Client profile [13].

It is possible for systems to provide the PRoT, ARoT, and NSPE in the form of a signed firmware image package or a single image. A firmware image package allows for packing bootloader images (and potentially other payloads) into a single archive that can be loaded. Nevertheless, each component must be measured independently. This is necessary for accurately attesting the boot state.

R70_TBFU_ATTEST: All images must be measured separately, even if they are in one firmware image package.
The state of the hardware may have a direct effect on the security of the system. As an example, a bootloader stage that supports an “unlocked” state may permit third party images to be loaded after an appropriate authorization procedure. This is considered a change of state because it affects and may be recorded in the boot state for remote attestation purposes.

*R80_TBFU_ATTEST: In addition to measuring the next stage, each stage must parameters that may influence the behavior of software.*

Examples of sensitive configuration data are boot parameters and configuration data, which might be stored separately from the images.

### 4 Architectural variants

The implementation of a secure boot process largely depends on the security properties of the non-volatile memory. This section presents the common variants. The figures:

- are not to scale
- omit redundant boots stages (for example, the ARoT could be an additional boot stage)
- do not show multiple images per stage. It is expected that more complex systems have multiple stages that contain multiple images per stage.

#### 4.1 Minimal boot from on-chip storage

This variant does not depend on external NVM. The internal flash images are XIP.
4.2 Minimal boot from off-chip storage

This variant has no on-chip NVM. Since no on-chip NVM exists for secure variables, both SPE and NSPE must be authenticated using embedded public keys on boot.

The SPE may be composed of multiple images. The SPE and NSPE may be a combined single image. Anti-rollback counters must be implemented in on-chip memory that is only accessible to the SPE.
4.3 Minimal boot using a passive security module

The ROTPK(s) and other hardware secrets are contained within the security module.

A security module may be also known as a secure element or a secure enclave. For example, a Trusted Platform Module (TPM), smart card, or a generic security processor. The security module may be on-chip or off-chip.
4.4 Minimal boot using a security processor

The security processor may be:

- A system control processor
- A secure enclave or advanced version of a secure element

The ROTPK(s) and other hardware secrets are contained within the security processor. The security module may be a secure enclave or a separate processor that controls the boot process. It has the capability to verify the initial SPE image in the internal flash. It is possible for a security module to verify multiple images. If the contents are successfully verified, then the main CPU is released and execution begins.

The non-volatile counters used for rollback protection of the SPE are stored within the security module. The main CPU then boots the SPE software.

The immutable bootloader is held within the security module.
5 eFlash considerations (informational)

SoCs are used in different market segments and have varying security requirements based on their usage models. Usage models can be part of open or closed software ecosystems. System designers must consider which assets they need to protect, and which threats they want to protect against. This specification describes the required rules to reduce the impact of software-based attacks. It also provides recommendations, where appropriate, to prevent scalable low-cost physical attacks.

The requirements for this document are derived from best security practices. To understand threats and general security objectives, refer to the Threat Models and Security Analyses (TMSA) provided by Arm.

Offline modification of off-chip flash memory is a likely risk if an attacker has physical access to the system. It is common for external off-chip storage to be present, such as an SD card or QSPI NOR flash.

Many consumer systems include a method of linking with a PC, for example a USB connection. This is an example of simple equipment that any attacker who is local to the system would have. Beyond this, an attacker might utilize more specialized equipment that can be easily acquired and that is relatively inexpensive. Examples of this equipment are JTAG interface controllers, soldering irons, and oscilloscopes. To perform the most sophisticated...
attacks, an attacker might require expensive laboratory-like equipment or software that must be specifically developed.

Internal on-chip flash that can easily be reprogrammed or erased using programmers is also considered as untrusted storage. Microcontrollers can offer different levels of protection for internal flash. It is important to understand the implications of these protections for both development and production scenarios. Special configuration registers may provide the following modes for the internal flash:

- **Internal protection**: This is typically used to protect a bootloader or any parts of the system to be overwritten either by the debugger or by the application itself. This protection can be reversed either by a mass erase capability or reprogramming a special configuration register. If the registers cannot be protected from application software, then this does not provide security against software-based attacks.

- **External protection**: Prohibits external tools or debuggers from accessing the flash. Designed to prevent reverse engineering of software. To recover the system a mass erase has to be applied. If the register cannot be protected from application software, then this does not provide security against software-based attacks.

- **Disable mass erase**: Prevents triggering of a mass erase. Combined with **Internal protection** and **External protection**, the system can never be manually reprogrammed. If mass erase is enabled and triggered then all system code, secrets and data are erased. If mass erase is enabled, this marks the end of the security lifecycle.

If the registers for these states cannot be protected from untrusted application software, then this does not provide security against software-based attacks. The configuration registers for these modes must be protected from application software.

### 6 Delegated signing schemes (optional)

The security of the trusted boot process is primarily dependent on the secrecy of the private portion of the ROTPK. If the ROTPK private key is leaked or lost, then no recovery is possible. To mitigate this risk, the private portion of the ROTPK should have limited exposure, while the attack surface is moved to a delegated signing key of lesser authority. By limiting the exposure of the ROTPK’s private key, strict operational processes can be put in place around its usage, which reduces the possibilities for attackers and enables a recovery method for defenders.

Implementations with higher robustness requirements should have a scheme in place to limit the exposure of the private key. Signing of key certificates should take place in a secure offline environment. Operational processes are very dependent on business processes and are not within the scope of this specification. However, this section will describe a couple of schemes that implementations may wish to consider.

Figures are provided to help illustrate the dependencies in such schemes.

The provided examples only consider one level of delegation. However, the examples can easily be extended to include intermediate certificate authorities.

**R01_TBFU_DELEGATION**: On systems that support subsidiary keys, a compromised image signing key must be revoked as long as the private key of the ROTPK is not compromised, by signing a new key certificate with an incremented key version value. The use of subsidiary keys is not mandatory.

**R02_TBFU_DELEGATION**: On systems that support subsidiary keys, the key version must be compared to an on-chip non-volatile counter to detect rollback of old keys.
6.1 Key certificate scheme

In this scheme a key certificate is used. A key certificate is signed metadata that explicitly contains the identities of the delegated signing authorities. The identities are represented by the hashes of each signer’s public key. The ROTPK private key is used to sign the key certificate. It is expected that this event is very rare.

Figure 12 shows an example of this scheme that includes a root authority with a delegated image signing key for the PRoT. The public key associated with the PRoT is called the PRoT-PK. It should be noted that the example could have more intermediary signers to create a longer chain.

Figure 12: Key certificate example with a PRoT image signer

The PRoT-PK does not need to be embedded in the PRoT image manifest. It may be separate without any protection.

6.1.1 Revocation workflow

If the PRoT image signer loses control of the private signing key, then the revocation process is as follows:

1. The PRoT image signer must generate a new key pair, keeping the private key secret.
2. The public key of the new key pair is given to the holder of the ROTPK using a secure process. The ROTPK owner validates the image signer using multiple alternative factors of authentication.
3. The holder of the ROTPK creates and signs a new key certificate, which includes:
   - an incremented Key Version number
   - the hashes of all delegated signer identities (their public keys), including the public key hash of the new keypair. The old public key hash associated to the PRoT image signer is removed and not included.
   - the policy of each signer (who can sign which type of image)
4. All signers (for example, the PRoT, ARoT and NSPE signers) must be notified by the ROTPK owner of the new changes
5. The new key certificate is sent to all image signers and must be included with the latest firmware update.

The process is the same for the ARoT signer and the NSPE signer if they require revocation.

**Example manifest using the IETF SUIT draft**

The IETF SUIT working group is working on a standard firmware manifest specifically for constrained devices. Arm recommends this standard for microcontrollers and embedded processors. This is based on the [draft-moran-suit-manifest-03](https://tools.ietf.org/html/draft-moran-suit-manifest-03) document available from the IETF.

<table>
<thead>
<tr>
<th>PSA-TBFU field</th>
<th>IETF-SUIT field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manifest version</td>
<td>manifestVersion</td>
<td></td>
</tr>
<tr>
<td>Image version</td>
<td>sequence</td>
<td>Used for anti-rollback protection</td>
</tr>
<tr>
<td>Image size</td>
<td>payloads</td>
<td>A list of payload components. There must be one PayloadComponent specified.</td>
</tr>
<tr>
<td>Image hash</td>
<td></td>
<td>Each payload must contain:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• payloadSize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• payloadDigest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• payloadComponentIdentifier</td>
</tr>
<tr>
<td>(optional)</td>
<td>dependencies</td>
<td>A list of dependencies.</td>
</tr>
</tbody>
</table>

**Example manifest using the X.509v3 standard**

Application processors, such as the Cortex-A series, have sufficient resources to handle industry standard X.509 certificates within a reasonable computation budget. This section provides an example interpretation of the PSA-TBFU manifest requirements using the X.509 standard.

The fields are part of the X.509 content certificate. Some fields defined in this specification are hence represented as X.509v3 certificate extensions.

DER encoding is recommended.

<table>
<thead>
<tr>
<th>PSA-TBFU field</th>
<th>X.509 field</th>
<th>Extension name</th>
<th>Extension criticality</th>
<th>Extension size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image hash value</td>
<td>Extension</td>
<td>FirmwareHash</td>
<td>1</td>
<td>&gt;=32</td>
<td>SHA-256 is 32 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SHA-512 is 64 bytes etc</td>
</tr>
<tr>
<td>Image size</td>
<td>Extension</td>
<td>FirmwareSize</td>
<td>0</td>
<td>&gt;=32</td>
<td>Images are unlikely to exceed 4GB in size. Bigger sizes may be provided.</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>--------------</td>
<td>---</td>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>Image type</td>
<td>Extension</td>
<td>FirmwareType</td>
<td>1</td>
<td>1</td>
<td>Identifies the component type. The value is IMPDEF.</td>
</tr>
<tr>
<td>Image version</td>
<td>Extension</td>
<td>NVCounter</td>
<td>1</td>
<td>4</td>
<td>Used for anti-rollback protection. Legacy implementations may use: “TrustedFirmwareNVCounter” or “NonTrustedFirmwareNVCounter”</td>
</tr>
<tr>
<td>Vendor/class ID</td>
<td>Certificate Issuer name</td>
<td>n/a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subject name</td>
<td>n/a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manifest version</td>
<td>Version of the certificate</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Used to make sure that an update is not intended for another device.</td>
</tr>
</tbody>
</table>