Arm® Platform Security Architecture Trusted Base System Architecture for Arm®v6-M, Arm®v7-M and Arm®v8-M 2.0

Document number:	DEN 0083
Release Quality:	Beta
Release Number:	2
Confidentiality	Non-Confidential
Date of Issue:	13/12/2019

Contents

About this document

Release Information

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

Date	Version	Confidentiality	Change
December 2019	2.0 Beta 0	Non-Confidential	Updated what TBSA-M compliance means with respect to the requirements and the SDL. This is done in the Introduction (section 1.5) Updated Cryptography requirements (Chapter 7) from NIST Suite B to CNSA and referred to other national standards. Replaced R220_TBSA_DEBUG with a recommendation. Many editorial changes to the text: clarifications, typographical and stylistic. Several clarifications to the wording of requirements. Added recommendations for dealing with Armv8-M UDE support (section 5.10.9). Updated references to NIST (page ix).
February 2019	1.1 Beta	Non-Confidential	Several re-wordings and fixing of typographical errors.
October 2018	1.0 Beta 0	Non-Confidential	Updated cryptographic References Added support for Ed25519
September 2018	1.0 Alpha 0	Confidential	Aligned document context with other PSA documents Removed most threat modelling description (PSA-SM to cover this) Modified many sentences to clarify based on internal feedback Added Labelling of each chapter as either [Informative] or [normative] Added section 1.2 which describes the document organization Removed PSA readiness chapter (out of line with current PSA intentions)
June 2018	1.0 Dev 0	Confidential	Fork from TBSA Armv8-M (DEN 0062A) Adding: non-TrustZone isolation mechanisms (Chapter 4)

	Use of Hardware Anchors, for example secure elements and secure enclaves (Section 4.5) 2 new requirements (R001_TBSA_BASE and R002_TBSA_BASE) Modifying: Context referring to PSA firmware framework and security model Existing text (in requirements) so that it does not rely solely on TrustZone for Armv8-M Removed PSA Readiness chapter

Arm® Platform Security Architecture Trusted Base System Architecture for Arm®v6-M, Arm®v7-M and Arm®v8-M

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.

NOTWITHSTANING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENCE, TO THE FULLEST EXTENT PETMITTED 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 validity, construction and performance of this Licence shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585

References

This document refers to the following documents.

Ref	Document Number	Title
[1]	PRD29 GENC 009492C	Arm Security Technology - Building a Secure System using TrustZone Technology
[2]	ARM DDI 00553A.b	Armv8-M Architecture Reference Manual
[3]	ARM DEN 0006B	Arm Trusted Board Boot Requirements
[4]		Trusted Firmware-A
[5]	ARM ECM 0437502	TrustZone Technology Microcontroller System Hardware Design Concepts.
[6]	ARM DEN 0021C	Arm Trusted Base System Architecture, CLIENT
[7]	ARM DEN 0063	PSA Firmware Framework – M-profile
[8]	ARM DEN 0072	PSA Trusted Boot and Firmware Update
[9]	ARM ECM 0390891	PSA: Device Security Model
[10]	ARM DEN 0073	Network Camera Threat Model and Security Analysis (English language Protection Profile)
[11]	ARM DEN 0074	Water Meter Threat Model and Security Analysis (English language Protection Profile)
[12]	ARM DEN 0075	Asset Tracker Threat Model and Security Analysis (English language Protection Profile)
[13]		**NIST Special Publication 800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators** **NIST Special Publication 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation** **NIST Special Publication 800-90C Recommendation for Random Bit Generator (RBG) Constructions**
[14]		**NIST Special Publication 800-22rev1a: A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications**

[15]	Commercial National Security Algorithm Suite. (superseding NSA Suite B Cryptography)
[16]	SEC - Recommended Elliptic Curve Domain Parameters
[17]	GlobalPlatform TEE Protection Profile Specification v1.2
[18]	NIST Special Publication 800-108 Recommendation for Key Derivation Using Pseudorandom Functions
[19]	NIST Special Publication 800-57 Part 1 Revision 4 Recommendation for Key Management
[20]	NIST Special Publication 800-107 Recommendation for Applications Using Approved Hash Algorithms
[21]	Common Methodology for Information Technology Security Evaluation 3.1
[22]	NIST FIPS PUB 202 SHA-3 Standard
[23]	IETF RFC 8032 Edwards-Curve Digital Signature Algorithm (EdDSA) Algorithms
[24]	OG-IS Crypto Evaluation Scheme Agreed Cryptographic Mechanisms
[25]	IPA/ISEC：JCMVP：Approved Security Functions
[26]	Office of State Commercial Cryptography Administration, P.R. China

Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
ACL	Access Control List
AES	Advanced Encryption Standard
APB	Advanced Peripheral Bus
API	Application Programming Interface
AXI	Advanced eXtensible Interface
CMAC	Cipher-based Message Authentication Code
DMA	Direct Memory Access
DPM	Debug Protection Mechanism
ECC	Elliptic-Curve Cryptography

ECU	Electronic Control Unit
EIP	External Interface Peripheral
HMAC	Hash-based Message Authentication Code
HUK	Hardware Unique Key.
IDAU	Implementation Defined Attribution Unit.
IPC	Inter-process communication
JTAG	Joint Test Action Group
MCU	MicroController Unit
MPU	Memory Protection Unit
MTP	Multiple-Time Programmable
NSC	Non-Secure Callable
NSPE	Non-Secure Processing Environment
NVM	Non-volatile memory
OEM	Original-Equipment Manufacturer
OTP	One Time Programmable
OWF	One-way Function
PE	Processing Element
PSA	Platform Security Architecture
RMA	Return Material Authorization
RoT	Root of Trust
ROTPK	Root Of Trust Public Key
RSA	Rivest-Shamir-Adleman
SAU	Security attribution unit.
SHA	Secure Hash Algorithm
SM	Security Model
SoC	System-on-Chip
SPE	Secure Processing Environment
SPM	Secure Partition Manager
SWD	Single Wire Debug

TA	Trusted Application
TDES	Triple Data Encryption Standard
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TMSA	Threat Model and Security Analysis
TRNG	True Random Number Generator
TRTC	Trusted Real-Time Clock

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 Base System Architecture for Arm®v6-M, Arm®v7-M and Arm®v8-M).
- The number and release (DEN 0083 2.0 Beta 2).
- 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 [Informative]

Arm *Trusted Base System Architecture for Armv8-M* (TBSA-M) is an architecture for the design and implementation of secure devices.

The design of a secure product depends on analysis of threats to its assets through application use cases. The number and diversity of M-profile CPU-based products mean that, in practice, this approach might not scale. To address this, TBSA-M encapsulates best practice security principles when designing systems around Armv8-M *Processing Elements* (PEs).

These principles support the design and integration of the following features rooted in hardware:

- A *Root of Trust* (RoT).
- A Protected keystore.
- Isolation between Trusted and Non-Trusted software components.
- A Secure firmware update mechanism.
- A lifecycle management mechanism, for secure control of debug, test, and access to provisioned secrets.
- A high-entropy random number generator, for reliable cryptography.
- Cryptographic acceleration, so that real-time functionality can be maintained with the correct security properties.

TBSA-M is part of Arm *Platform Security Architecture* (PSA). PSA can be used to define a secure processing environment that isolates security-critical functionality and data from application software. This increases confidence in the trustworthiness of the device, even in the presence of exploitable software vulnerabilities. Arm PSA defines *Application Programming Interfaces* (APIs) for fundamental security functions. The APIs support the development of secure functionality that is more easily ported to other Arm-based platforms.

The goal of this document is to support the creation of platforms that support Trusted services. Trusted services are collections of operations and assets that require protection from the wider system, and from each other. This ensures their confidentiality, authenticity, and integrity.

This document aims to provide information that is useful to designers and implementers of secure platforms. This does not eliminate the requirement for security analysis during system design, despite the intention to reduce it. In addition, following the requirements and recommendations of this document does not guarantee that vulnerabilities will not exist in any compliant design.

TBSA-M describes a set of security requirements that assist the development of secure target systems. Among other uses, this set of requirements can support pre-silicon security design analysis within a company-specific secure development lifecycle. Some requirements have a clear architectural focus to support secure isolation partitions in a device as described in the PSA firmware framework. Other requirements are generic security requirements that represent good practice within the class of target systems.
TBSA-compliance is achieved when all requirements are either met or deemed not applicable. A Requirement may be deemed not applicable when the threats that the requirement mitigates is documented as countered in another manner or that those threats are demonstrated as not present in the specific threat model for the device. Security design reviews should compile evidence on each requirement that might be helpful in achieving PSA certification.
TBSA-M recommendations are provided as default design choices. Documenting which recommendations are followed, and which are not, is not required as part of the process of establishing TBSA-M compliance. However,

in some cases documenting which recommendations are followed may benefit the security design review. This documentation may form part of the compilation of evidence to support claims of adherence to, for example, relevant national security standards or market certification regimes.

1.1 Scope

This version of the TBSA-M specification is targeted at connected *microcontroller SoCs* (MCUs) and their product use cases, for example IoT. The MCUs have an Armv8-M, Armv7-M or Armv6-M processor as its host, and have sufficient security features to authenticate a boot process and ensure confidentiality and integrity of its data.

Implementations compliant with TBSA-M are sufficient to operate within the PSA *Security Model* (SM), which allows deployment of secure services using devices with known security properties

1.2 Organization of the document

The document is organized in eight chapters in addition to this Introduction. Chapters 2, 3 and 4 describe the context in which subsequent requirements should be understood. Chapter 5 provides a set of requirements that TBSA-M compliant devices are expected to meet, together with a brief commentary about each requirement. Chapter 6 contains recommendations for how lifecycle management may be implemented, and Chapter 7 lists approved cryptographic algorithms for which hardware support is prescribed in Chapter 5. The Appendix tabulates the requirements in Chapter 5 for ease of access and to support design reviews.

Some chapters are labelled **Normative**. These chapters are prescriptive and are expected to be followed to comply with the architecture – subject to threat-model-based exemptions which should be documented at the security design review stage.
Other chapters are labelled **Informative**. These chapters are descriptive and are intended to help the reader to understand the concepts presented in the normative chapters and to provide reasonable defaults for certain design choices.

1.3 Target Platforms

The target platforms that are addressed by this document are primarily IoT devices and automotive ECUs. These devices typically have several of the following product features:

- A long active lifespan.
- Resource constrained.
- A location that makes secure manual updates difficult.
- Potentially good physical access for untrusted third parties.
- Deployment in vast quantities.

This document concerns SoCs that are centered on Armv6-M, Armv7-M and Armv8-M PEs and embed non-volatile storage.

In general, these SoCs:

- Are closed systems so that the software running on them can be controlled.
- Have wired or wireless network connectivity.
- Support internal non-volatile bulk storage, most commonly embedded flash (eFlash).

- Integrate *One Time Programmable* (OTP) non-volatile storage for storage of assets provisioned at manufacture and later.

Given the variety of platforms and products that are covered by the scope of this document, each with a specific set of use cases, assets and threats, several aspects of this document are necessarily high-level.

However, the resulting collection of use cases, assets, threats, and necessary security measures cannot be reduced to a single, simple checklist of security requirements. Each platform requires specific analysis to determine the appropriate use of security features and will need to consider the specification and certification requirements of the target market.

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 are intended to strengthen the security guidance when compared to previous versions of this document and its predecessors. In all cases, the differences are in the degree of security that is provided, or that is demanded by other market specifications. The newer requirements described here are more resilient to certain types of attack.

1.3.1 Level of protection

The types of attack that this document addresses are primarily those in which the attacker deploys unsigned software, either locally or remotely, or by injecting faults. The motivation behind this type of attack is to steal proprietary information , disrupt device functionality, or both. Although hardware protection technologies can protect against direct access to sensitive information, it is still possible to acquire sensitive information from statistical analysis attacks. Therefore, for certain applications it might be necessary to design both hardware and software so that visibility of such information outside the device is restricted.

Lightweight hardware attacks are those achieved using commonly available consumer or hobbyist equipment. Attackers obtain physical access to the device, but do not have the equipment or expertise to attack within the integrated circuit package. For example, an attacker might attempt to attack the system by:

- Probing the signals around the SoC to:
    - Read, modify or substitute external memory contents.
    - Read, modify or substitute information on communications channels.
- Tampering with how the device is clocked, powered or reset to corrupt programmable states within the system.
- Tampering with the device pins, for example debug pins, to attempt to read, modify or substitute internal states or internal memory contents.
- Tampering with manufacturing-related test pins to attempt to read, modify or substitute internal states or internal memory contents.

TBSA-M requirements focus on protecting easily accessible interfaces, discouraging the use of class keys, and supporting software countermeasures to low-cost side channel attacks.

Finally, 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.

1.4 Security Development Lifecycle

A *Security Development Lifecycle* (SDL) is a structured methodology for the creation of products which incorporate secure practices during each stage of their design lifecycle.

An SDL establishes a series of processes, policies, and activities that expand a design lifecycle to improve product security. SDLs are normally customized to fit with the particular design flow of the organization that is performing the design. Table 1 shows some of the early outputs of an SDL:

Table 1: SDL outputs

Artifact	Description
Security Product Requirements	These are application and usage-specific security requirements. These requirements include certifications that are imposed by the market, and certifications whose need comes from the application.
Threat Modeling	This identifies assets and threats at architectural level and microarchitectural design level. Relevant threats are determined using market requirements, vulnerability and risk analysis.
Security Objectives	These are high-level, descriptive countermeasures that mitigate the in-scope threats for identified vulnerabilities. Security objectives include assumptions about the operational environment.
Security Functional Requirements and Non-requirements)	These are Security Objectives mapped to specific mechanisms (low-level, prescriptive), which are to be implemented in the design.

These outputs mean that:

- Design specifications include security requirements.
- Design and verification reviews need to cover security requirements.
- Verification plans need to include security verification.
- Project reviews and stage gates need to consider progress in meeting security requirements.
- Errata need to be reviewed for security impact.

This document supports these activities by listing and describing common security requirements that need to be met in secure designs that are based around Armv6-M, Armv7-M or Armv8-M PEs.

1.5 Compliance

A claim of compliance to TBSA-M is an evidence-backed assertion that the design meets all applicable requirements that are described in this document. The assertion is normally made by the design team and takes the form of documented output of a design review of the device. Arm recommends that this assessment is made as part of the SDL.

The design team shall confirm, for each requirement, whether the requirement is fulfilled. This confirmation shall include a brief description of why the design is compliant and references to the relevant detailed specifications. In general, requirements may not be applicable if the threats that they mitigate can be shown to not form part of the threat model of the device, or that any vulnerabilities that might result from not meeting a

requirement can be demonstrated to be mitigated in another way. In some cases, it will be necessary to provide stronger security than is anticipated by these requirements. In these cases, evidence shall be documented to support this approach alongside the requirement. The Appendix of this document includes a checklist to assist in this activity.

In several areas, TBSA-M provides recommendations. Where possible, these recommendations are provided to give guidance on reasonable default design choices. The threat model and functional requirements of the device is key in determining how requirements are met and which recommendations are followed. This is beyond the scope of this document.

1.6 Scope and intended Audience

This document is primarily intended for the use of chipset manufacturers who wish to claim compliance with ARM TBSA-M architecture requirements. Architects, designers and verification engineers can use this specification to support the process of certification against PSA with independent laboratories.

2 Trustworthy networked devices [Informative]

Threat modeling and security analysis often requires detailed and expert evaluation of devices and the context in which they operate. This includes their use-cases, the assets they protect, and the threats they may face. The quantity and diversity of IoT, and other areas in which TBSA-M will be used, means that this approach does not scale well for cost-optimized devices.

A common feature of TBSA-M systems is that their context is a network of devices connected via insecure links. TBSA-M focuses on support for Trusted devices engaging in authenticated communications using cryptographic protocols, for example *Transport Layer Security* (TLS).

The complexity of software deployed on such devices requires that software is renewable to patch exploitable vulnerabilities, reliability, or performance issues and to manage the available functions of the device. Fundamental to the security model is hardware support for security by separation of software components.

2.1 Use Cases

TBSA-M requirements are driven by two classes of use cases.

2.1.1 Secure machine to machine communication

A trustworthy network must be able to establish and maintain secure communications between devices that have mutually verified the identity and state of each other over physically insecure channels. In general, this can be achieved (with hardware support) for several basic functions:

- Secure boot
    - To ensure that only authenticated software can run on the device.
- Secure debug and test
    - To ensure that only authenticated entities can debug the software and test the hardware.
- Control of device lifecycle
    - To ensure that the protection of assets and the availability of device functions follows a prescribed and constrained path from manufacture to device disposal.
- Attestation
    - To ensure that devices present reliable evidence to other parties about the software and lifecycle state it is running. These require provision of an attestation identity.
- Isolation
    - To ensure that accessibility of the most Trusted assets is in proportion to the level of trust in the software that can access it, and to provide hardware support for management of the confidentiality of assets between different device stakeholders.
- Cryptographic algorithms
    - To provide secure storage, computation and communication at a performance level appropriate for the device. Many cryptographic algorithms require support for true random number generation, provision of Trusted time and the secure provision of device credentials and certificates.
- Unique binding
    - To ensure that application level keys, credentials, and other secret or sensitive data is uniquely accessible to a specific device when in a specific state.

In addition, most devices will need to provide some protection from hardware attacks, including countermeasures to probing of accessible interfaces or other low-cost physical tampering.

2.1.2 Secure firmware update

Significant device firmware should have updateable components which encompass parts of the device RoT all the way up to application software. The location and quantity of deployed TBSA-M devices means that updates should be achievable over a network without requiring physical intervention. This requires the following hardware resources, to ensure that update is performed securely:

- Provision of firmware integrity and authenticity keys.
- Support for approved cryptographic protocols for reception, validation, and installation of new firmware, including monotonic version counters and support for trusted time.
- Provision of non-volatile memory to hold new firmware images and audit logs.
- Resources and mechanisms to remain secure in the event of a failed update, for example, a failsafe backup or a mechanism of removal from Trusted services.

2.2 Security goal

The goal of the TBSA is to protect the Trusted services on a device from attackers who would benefit from a compromise.

No security implementation is flawless. The goal is to ensure that attacks are not worth the time and money required. TBSA-M aims to protect against the most common general attacks. TBSA-M does not protect against all types of attack. Depending on its target market, a device implementing the TBSA-M architecture might be required to meet stricter security requirements.

Prevention of the Non-Trusted World from being compromised is out of scope for TBSA-M. Also, TBSA-M does not prevent malware from executing in a rich execution environment that attempts to acquire sensitive user information by acting as a trustworthy entity.

TBSA-M does not address laboratory attacks in which devices are unpackaged and probed, or power analysis attacks in which the power consumption of the device is correlated with its processing activity to extract assets.

2.3 Security analysis

A number of threat modeling methodologies are suitable for TBSA-M devices. Within PSA, the *Threat Models and Security Analyses* (TMSA) specifications provide some examples of best practice (see [10],[11] and [12]).

3 Platform Security Architecture concepts [Informative]

3.1 Security by separation

All use cases rely on the protection of the sensitive assets, for example private data, cryptographic keys, credentials, and firmware, from disclosure or modification. This is best achieved by limiting the firmware and hardware that has access to them and separating these assets from the application firmware and hardware.

PSA comprises multiple building blocks to meet security objectives. The foundation of PSA is a separation of the system into a *secure processing environment* (SPE) for the sensitive assets and the code that manages them. The SPE is isolated from the *Non-secure Processing Environment* (NSPE), in which the main application and communication firmware executes. The *secure partition manager* (SPM) is the Trusted component within the SPE that is responsible for the isolation of the SPE and providing communication between the SPE and NSPE.

Faults or malicious activity within the NSPE should not compromise the behavioral integrity of the SPE.

Figure 1 provides a high-level view of the structure defined by PSA:

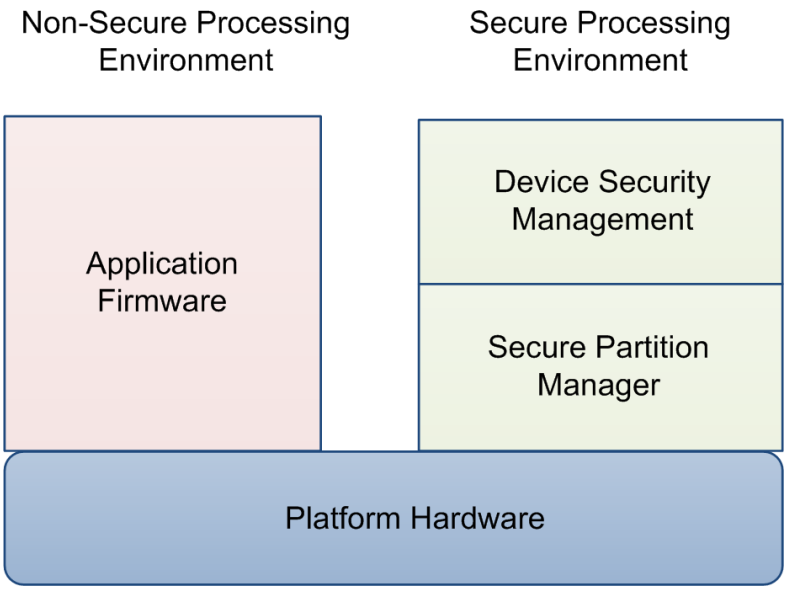

Figure 1: Separation of the Secure Processing Environment

3.2 Trusted and Non-Trusted worlds

For Armv8-M-based devices, the separation provided by TrustZone into Secure state execution and Non-secure State execution gives rise to two worlds: the Trusted world, and the Non-Trusted world, respectively. The Trusted world is used by PEs executing in their Secure state or by peripherals acting on their behalf. Non-TrustZone processors are fixed to operate in one of the worlds. TrustZone processors can use secure transitions to operate in both worlds.

This document will use the term Trusted world to refer to hardware resources whose state supports the SPE and the term Non-Trusted world to refer to those hardware resources whose state supports the NSPE.

3.3 PSA software architecture

For many devices, secure remote device management services and application network management services will be provided by different organizations. Thus, adding secure device management services to a system not only increases the firmware complexity, but also adds another vendor whose product needs to be integrated in the device. This vendor's presence will increase integration complexity and cost.

Isolation of the sensitive assets further increases integration complexity. The secure device management firmware depends on the SPE design to execute the security-critical functionality in an isolated environment, and on the communication firmware used in the NSPE by the application.

PSA addresses some of this complexity by providing a standard SPE architecture and API for developing and communicating with firmware that runs within the SPE. PSA provides the specification of SPM and SPE, but not the implementation. Arm provides an open-source reference to implement these components as a separate product.

As an architecture, PSA enables alternative implementations of the SPE that address different quality-of-implementation requirements for different parts of the IoT device market, while retaining the same programming model for securely isolated device functionality. For example, lighter-weight implementations are needed for more constrained MCUs and Partition Managers based on proven separation kernels for products that demand formal verification.

Figure 2 illustrates the mapping of the Non-Secure and Secure processing environments to the Non-Trusted and Trusted Worlds.

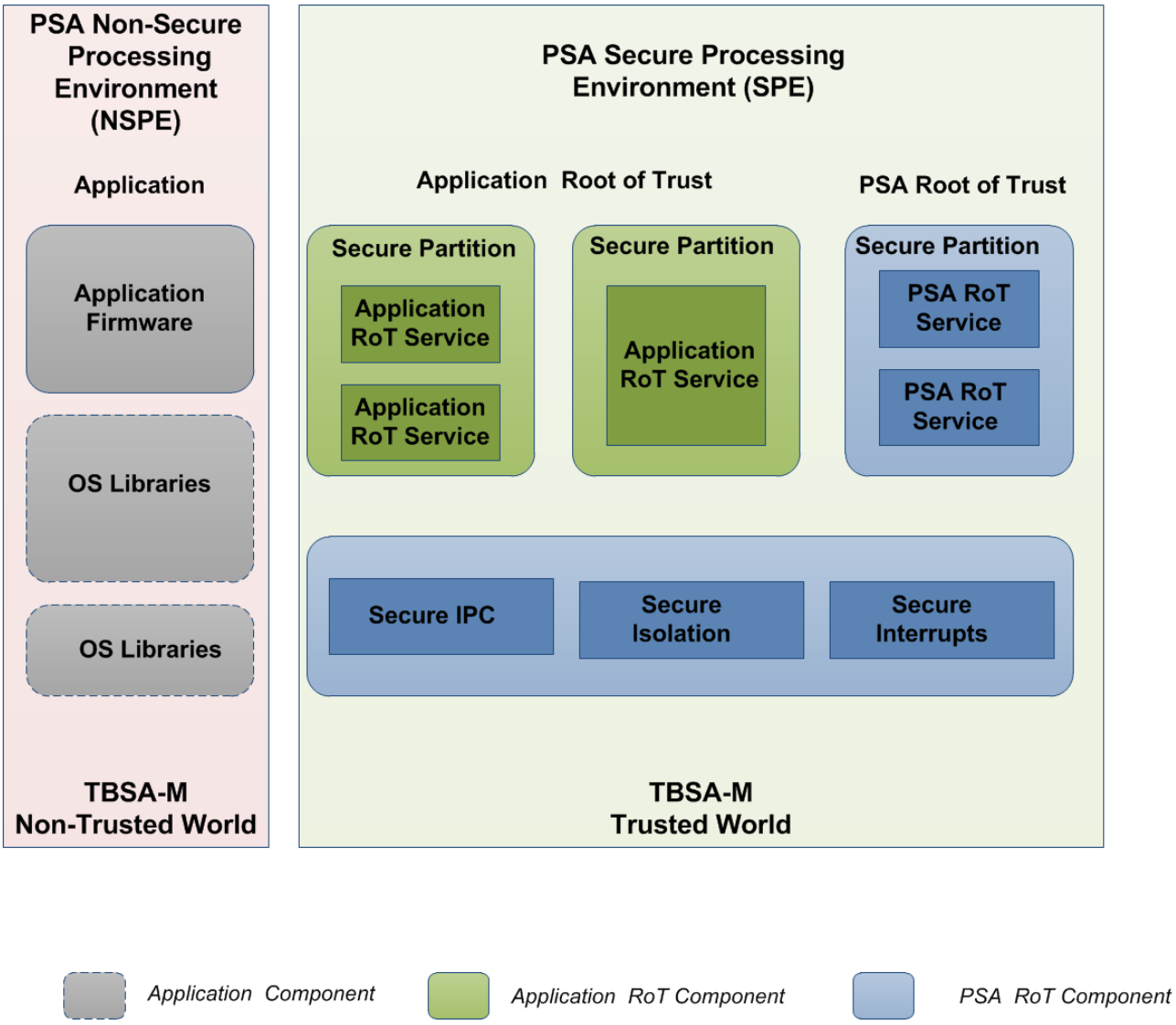

Figure 2: PSA Secure Processing Environment mapped to TBSA-M Trusted world

As described in Section 3.1, the PSA Security Model separates system processing into two domains: the NSPE and the SPE. The NSPE is typically much larger and includes the application firmware and OS kernel and libraries, and usually controls most I/O peripherals. The SPE includes the security firmware and those hardware resources that need to be isolated from NSPE firmware and hardware resources. A fundamental requirement of the Security Model is that NSPE firmware or hardware cannot inspect or modify any SPE hardware, code or data.

The PSA Security Model sub-divides the SPE into two sub-domains: the PSA *RoT* and the *Application RoT* (ARoT). The PSA RoT provides the fundamental secure services to the system and manages the isolated execution environment for the Application RoT secure services.

The PSA RoT comprises:

- A Security Lifecycle, which identifies the current phase of the device and controls the availability of device secrets and invasive capabilities, for example secure debug.

- An Immutable RoT, which is the combination of hardware and non-modifiable firmware and data installed during manufacturing.
- A Trusted Boot and Firmware Update, which ensures the integrity and authenticity of all secure firmware that runs on the device.
- An SPM, which implements the required isolation of the secure services, the *Inter-Process Communication* (IPC) mechanism that allows software in one partition to make requests of another, and scheduling logic to ensure that Partitions with requests to service are given execution time.
- A set of RoT services, which provides essential cryptographic functionality and manages access to the immutable RoTs for Application RoT services.

The PSA firmware framework [7] specifies the SPM, defines the runtime environment for RoT services and defines the standard interfaces for PSA RoT services.

4 Hardware supported isolation [Informative]

TBSA-compliant devices implement hardware to support the PSA isolation model.

Supporting the PSA isolation model gives rise to many architectural choices. For informative purposes, some of these are outlined in the following sections. Arm recommends that, where possible, a TrustZone-based system is implemented. TrustZone is known to provide a robust hardware framework when supported by a small amount of Trusted firmware. In addition, Arm provides significant design support for TrustZone-based systems that may not be available for other architectural choices.

4.1 TrustZone-based isolation

Processing Elements based on certain Arm architectures may support intrinsic isolation between secure and non-secure environments.

TrustZone for Armv8-M is a hardware-based security infrastructure that includes:

- An additional secure execution state of the processor including an MPU dedicated to this secure state.
- Secure interrupts.
- Secure debug.
- Infrastructure for propagating the security state of the processor to memory and peripherals, so that memory and peripherals are effectively isolated into secure and non-secure partitions.

TrustZone for Armv8-M recognizes software as running in either the Secure or Non-secure state. These Security states are orthogonal to the existing Thread and Handler modes, and there is both a Thread mode and a Handler mode in both Secure and Non-secure state. Thread mode can also be either privileged or unprivileged. Figure 3 illustrates these states.

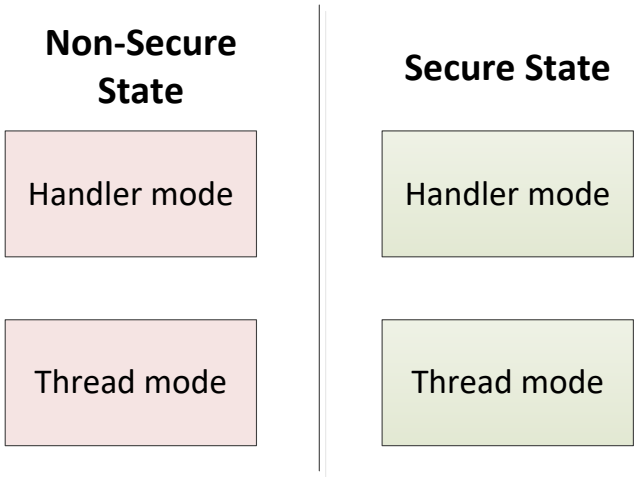

Figure 3: TrustZone for Armv8-M adds Secure and Non-secure states to the operation of a PE

Similar to TrustZone in Cortex-A processors, code running in Secure state can access both Secure and Non-secure information, but Non-secure programs can only access Non-secure information.

TrustZone for Armv8-M is an optional architecture extension. By default, the system starts up in Secure state if the TrustZone Security Extension is implemented, as is required for TBSA-M.

TrustZone for Armv8-M is designed with small energy-efficient systems in mind. Unlike TrustZone in Cortex-A processors, the division of Secure and Non-secure states is memory-map based, and the transitions take place automatically, without the requirement for a Secure Monitor exception handler. This eliminates switching overhead.

The designer of a microcontroller or SoC device must partition the memory spaces into Secure and Non-secure regions. Some regions are defined by software using a new unit defined by the Armv8-M architecture called the *Security Attribution Unit* (SAU), or by a device-specific controller logic connected to a special *Implementation Defined Attribution Unit* (IDAU) interface on the PE. The relationship between the attribution units is shown in Figure 4.

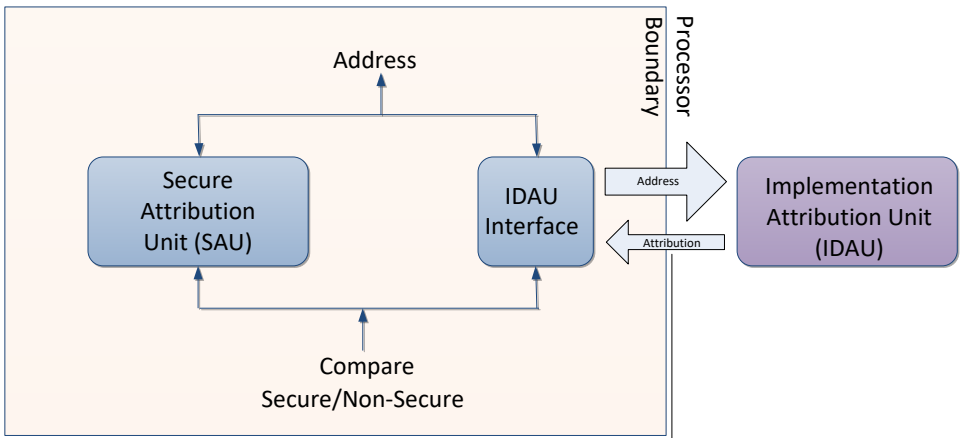

Figure 4: Security attribution defined by SAU and IDAU

The SAU is programmable in Secure state and has a programmers' model that is similar to the MPU. The SAU implementation is configurable by chip designers. The SAU should always be present, but the number of regions is defined by the chip designer. Alternatively, chip designers can use an IDAU to define a fixed memory map, and use an SAU optionally to override the security attributes for some parts of the memory. This is the recommended approach for TBSA-M devices.

The *Processing Element* (PE) state is dependent on the memory space definition. When the PE is running code in a Secure region, it is in the Secure state. Otherwise, it is in the Non-secure state. Application code can branch to, and call, code in the other Security domain, and the PE detects these switches automatically. Because an application can access functions in the other domain directly, TrustZone for Armv8-M is both flexible and simple to use.

The Secure memory space is further divided into two types:

- Secure: contains secure program code or data. This includes Secure stack, heap and any other Secure data.
- *Non-Secure Callable* (NSC): contains entry functions, for example, entry point for APIs, for Non-secure programs to access Secure functions.

Typically, NSC memory regions contain tables of small branch veneers. These are entry points. To prevent Non-secure applications from branching into invalid entry points, a new instruction called *Secure Gateway* (SG) has been introduced. When a Non-secure program calls a function in the Secure side:

- The first instruction in the API must be an SG instruction.
- The SG instruction must be in a region attributed as secure and NSC by the SAU or IDAU.

There are many other security checking mechanisms within the Armv8-M architecture. Arm recommends the *Arm Architecture Reference Manual, Armv8, for Armv8-M architecture profile* for details.

In a typical TBSA-M compliant device, the host PE is only one part of the security system. Additional hardware is required to meet security requirements at a system level, to allow memory blocks to be partitioned into Secure memory regions and Non-secure memory regions. Similarly, access permission control logic is required to manage access permission of peripherals. Legacy peripherals and legacy bus masters are reused with appropriate bus wrapper logic.

Figure 5 shows a typical SoC architecture based on TrustZone technology. The processor cluster is supported by several security hardware IPs that utilize TrustZone technology, for example the NS-bit, to work within the Trusted world.

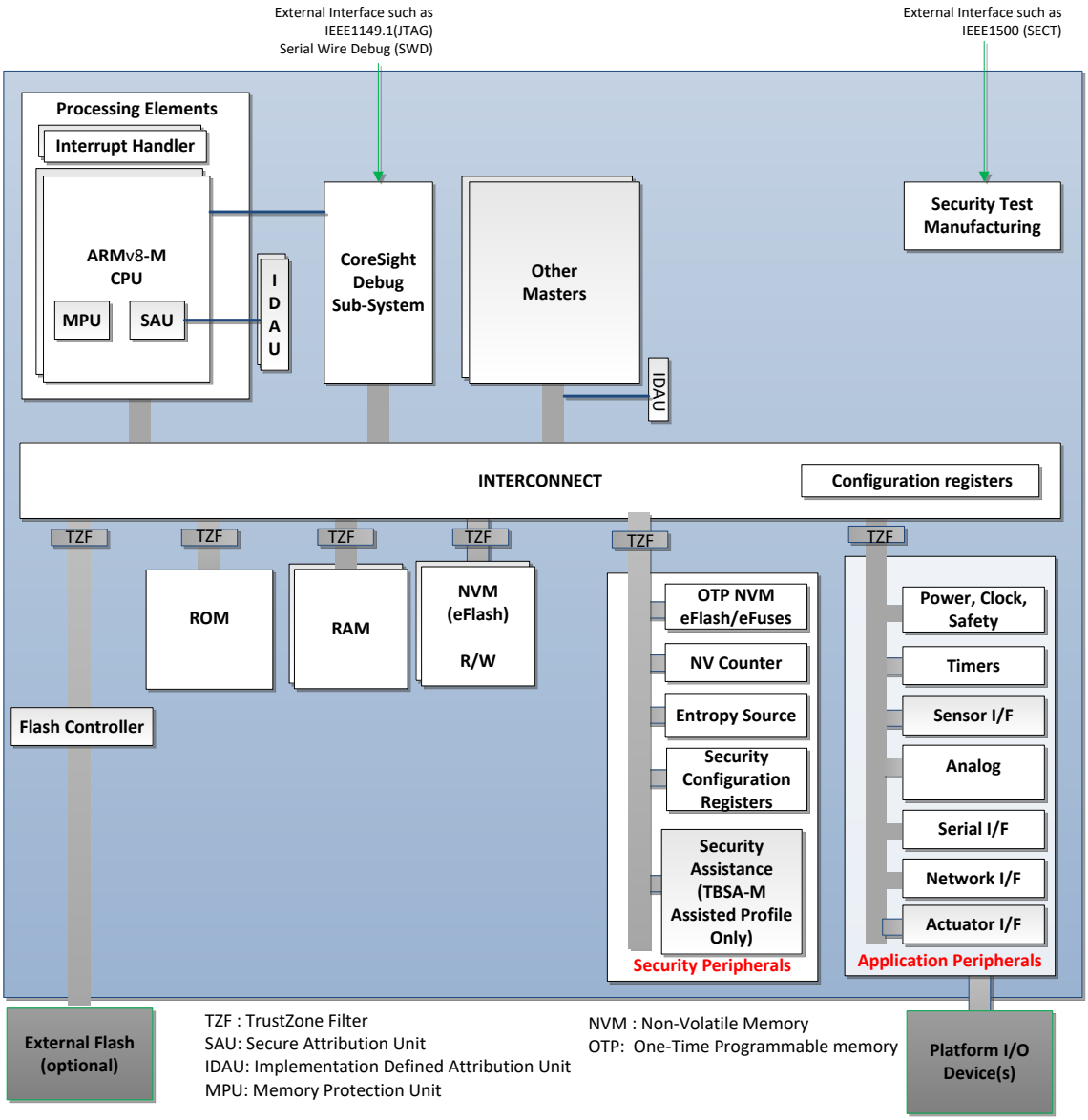

Figure 5: Example TBSA-M system using TrustZone for Armv8-M

In a TBSA-M system using TrustZone for Armv8-M, TrustZone isolates the Trusted world from the non-Trusted world and provides the level-1 isolation (see Section 4.6).

Arm recommends deploying TrustZone for Armv8-M based solutions early in the design process, because PSA implementations using TrustZone for Armv8-M commonly provide robust and performant low-cost solutions with wide applicability.

4.2 MPU-based isolation

In a SoC that relies on MPU-based isolation, control of the *Memory Protection Unit* (MPU), is the primary means of isolation on an SoC, together with support for privileged execution on the processor. The separation of the Non-Trusted world and the Trusted world is provided by appropriate configuration of the MPU.

On this sort of platform, the PSA firmware framework uses a self-contained *Secure Partition Manager* (SPM) that creates independent security domains and provides hardware-enforced sandboxes, called partitions, for individual code blocks by limiting access to memories and peripherals. The SPM:

- Is initialized right after secure boot.
- Runs in the privileged mode of the processor.
- Sets up a protected environment using an MPU, for example the Arm Cortex-M MPU or a vendor-specific alternative. In particular:
    - Its own memories and the security-critical peripherals are protected from the unprivileged code.
    - *Access Control Lists* (ACLs) limit unprivileged access to selected hardware peripherals and memories.
- Allows interaction from the unprivileged code by exposing Supervisor Call-based APIs.
- Forwards and deprivileges interrupts to the unprivileged handler that has been registered for them.
- Prevents register leakage when switching execution between privileged and unprivileged code, and between mutually untrusted unprivileged modules.
- Forces access to some security-critical peripherals, for example *Direct Memory Access* (DMA), through Supervisor call-based APIs.

When using MPU-based isolation, the application and other parts of the NSPE run in the unprivileged processor execution mode and:

- Have direct memory access to unrestricted unprivileged peripherals.
- Can require exclusive access to memories and peripherals.
- Can register for unprivileged interrupts.
- Cannot access privileged memories and peripherals.

It is often difficult to run an application entirely in unprivileged state, because of restrictions on programs executing in this state. This means that significant parts of an application, with their accompanying vulnerabilities, might execute in a privileged state. Therefore, devices which rely solely on MPU and privileged execution for isolation afford the least robust implementations of PSA.

4.3 Dual PE-based isolation

It is sometimes convenient to dedicate one processing element to execute Trusted world firmware and to dedicate a second, different PE to execute Non-Trusted world firmware. This kind of design can provide sufficient isolation for a robust PSA implementation, together with suitable separation of requests, separate routing of interrupts and separation of debug. **Figure 6** illustrates this approach:

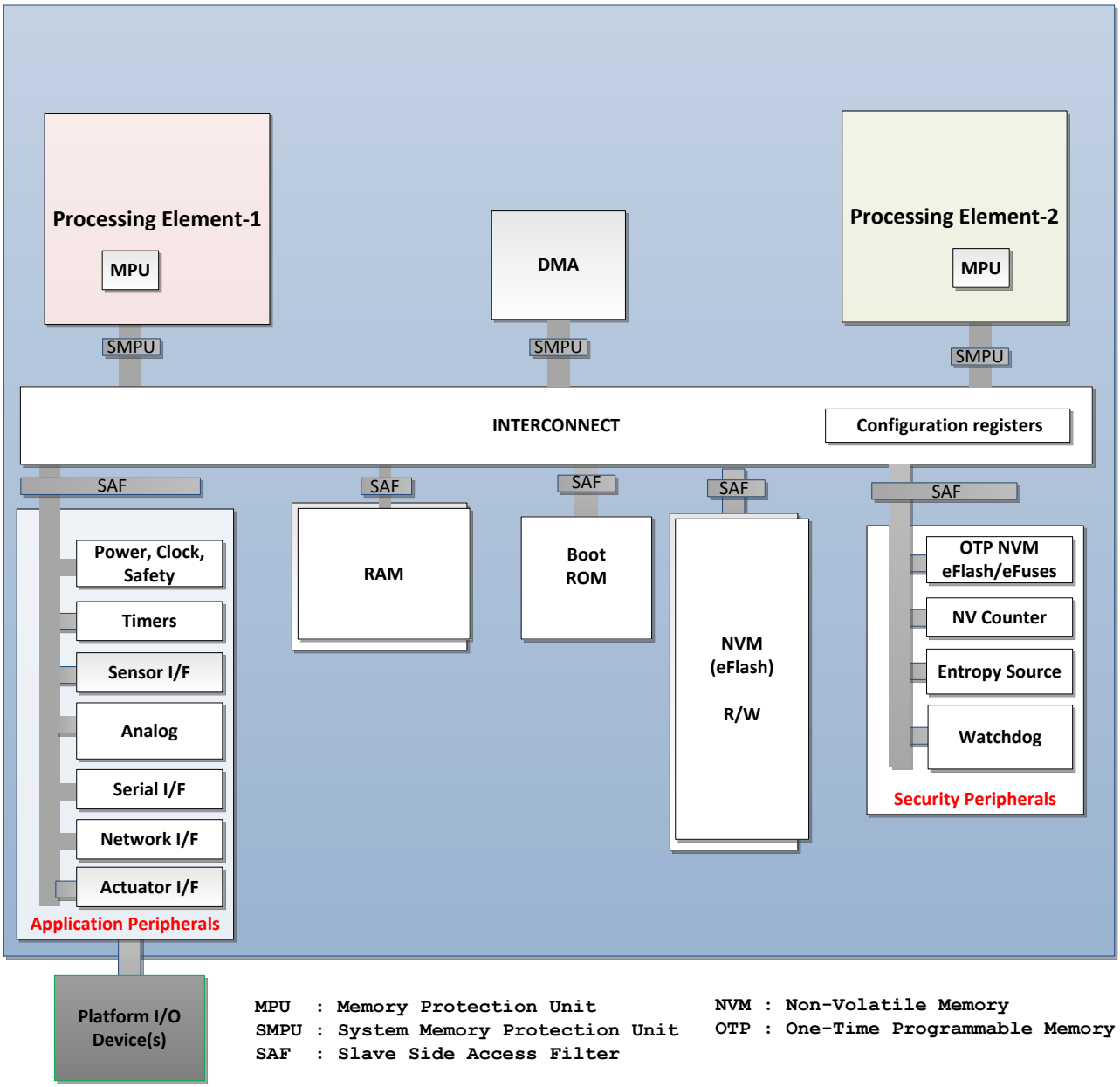

Figure 6: Example of a dual-CPU TBSA-M architecture

The separation between the worlds using dual PEs is supported by hardware in several ways. For example, a TrustZone-based system is functionally emulated by tagging bus requests and filtering them to ensure isolation between the Trusted and Non-Trusted worlds. Interrupts are statically hardwired to the core of the appropriate world. If more flexibility is required, then use a structure that routes some interrupts through a trusted interrupt controller first.

This arrangement allows more flexibility in deploying CPU performance. However, it typically uses more area and power than an equivalent single TrustZone-based CPU approach. Also, calls between firmware in the Trusted and Non-Trusted worlds will typically suffer higher latency in a dual-CPU approach.

4.4 Custom-logic based isolation

Custom isolation logic is used to create generalized protection zones for SoC resources. In a protection zone, no access is granted to address regions, interrupts and debug for CPU execution contexts outside the zone. This mechanism can be used to separate the Non-Trusted world from the Trusted world and to provide a hardware foundation for PSA.

For example, custom logic is used to restrict memory access dependent on the PC of the instruction that made the access. Such logic is supplemented to isolate interrupts and debug access using master or slave-side filters that are configured by the Boot ROM prior to loading of the application firmware.

This arrangement allows PSA isolation to be built on top of the mechanism, provided the mechanism meets the security requirements outlined in Section 4.6. Figure 7 illustrates this SoC design:

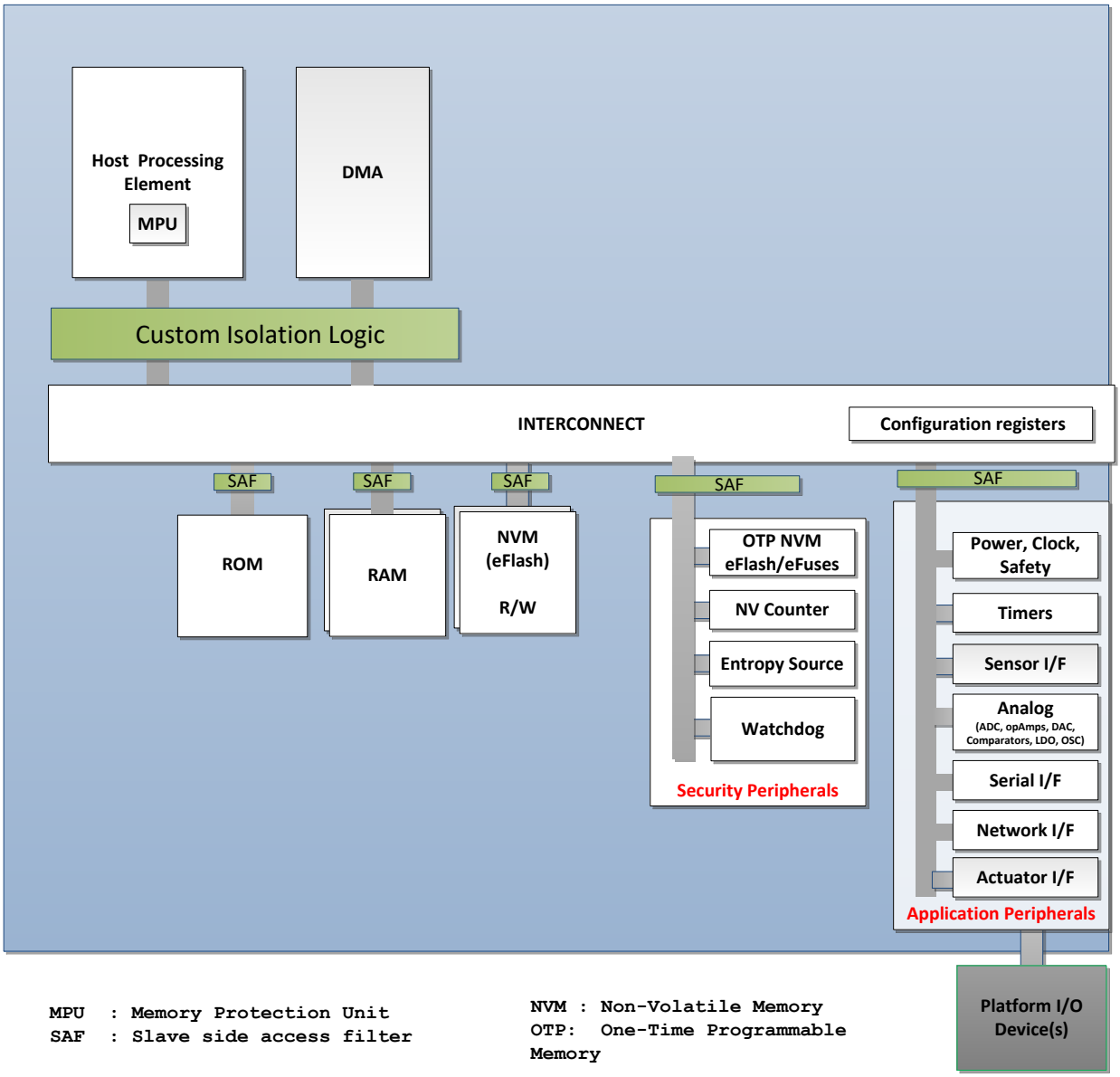

Figure 7: Example of a TBSA-M architecture using custom isolation logic

Well-designed custom isolation logic can be very robust in protecting read-out of firmware IP by application software that is outside the firmware's protected zone. However, in general, care must be taken in the integration and security verification of such schemes within the firmware ecosystem and the extent to which such designs are re-used.

4.5 Trusted subsystems

Trusted subsystems are re-usable blocks of security IP that sit within the trust boundary of the PSA RoT. They provide RoT security services to the device. The PSA RoT attests their implementation and configuration. They may be integrated, or external and bound to the SoC. Examples include IP, for example DRAM protection systems, or Trusted Peripherals that support cryptographic operations, and Secure elements and Security Enclaves. In the case of Secure Elements and Security Enclaves, the subsystem implements its own local RoT and its own local security life cycle.

Security Elements are independent subsystems that provide a large set of RoT services for the device. Security Enclaves additionally integrate a processor on which it is possible to run application-specific firmware.

Figure 8 illustrates a TBSA system in which a Trusted subsystem, in the form of a security enclave, combines hardware accelerators, RoT control hardware with a layer of security middleware, and software tools for the IC and device production process.

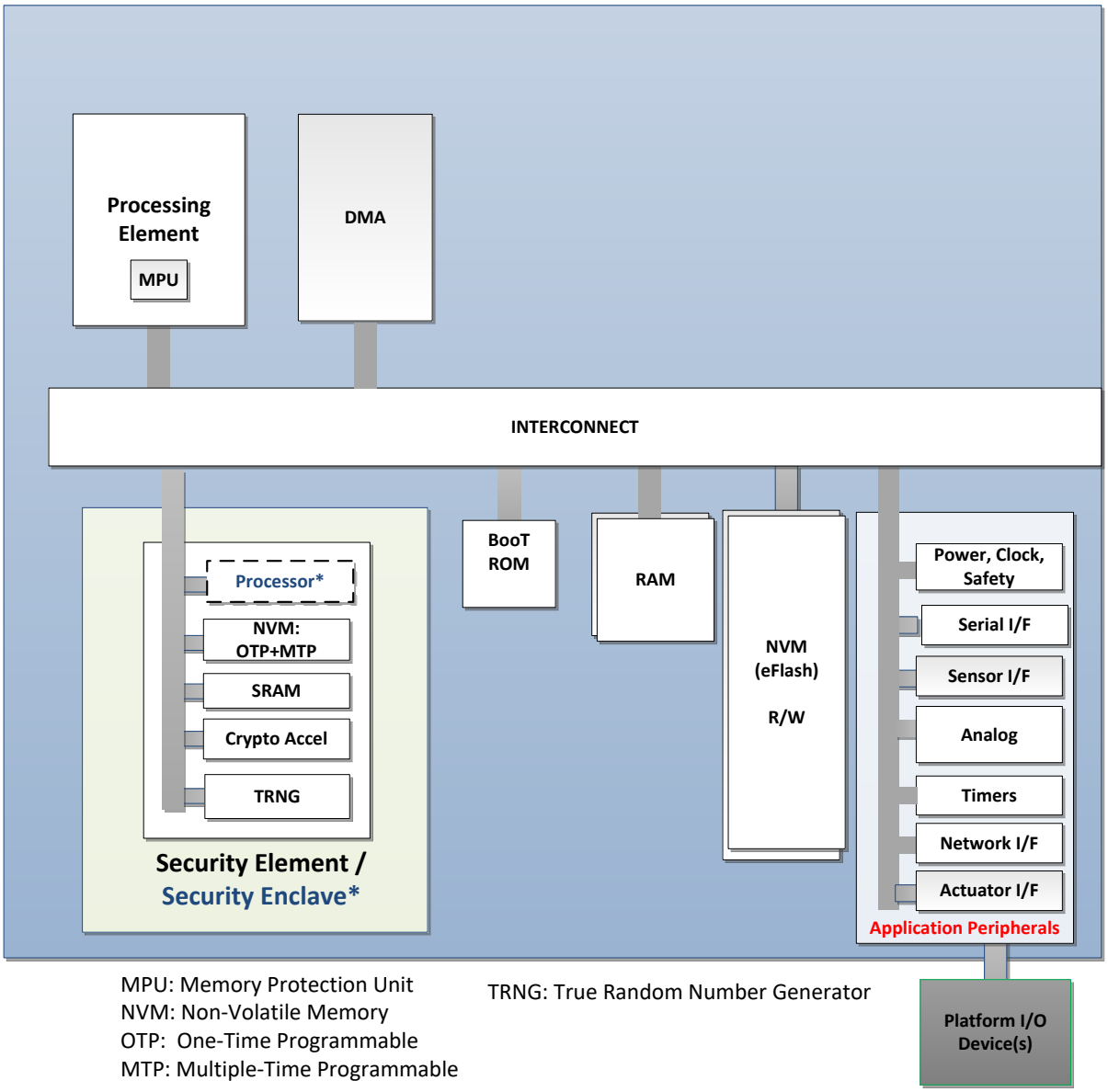

Figure 8: Example of a TBSA SoC with an integrated security element/secure enclave

The security enclave provides:

- Cryptographic acceleration for the protection of data-in-transit and data-at-rest.
- Protection of various assets belonging to different, optional, stakeholders, for example IC vendors, device manufacturers, service operator or users). These asset protection features include:
    - Image verification at boot/during runtime.
    - Authenticated debug.
    - Random number generation.
    - Lifecycle management.
    - Provisioning of assets.

Trusted subsystems combine with one of the isolation mechanisms described in 4.1, 4.3 and 4.4 to make the divide more robust. See Section 4.8 for details.

TBSA systems can also pair an SoC with an external secure element. Figure 9 illustrates this type of system. Depending on the application, a secure element may be used to:

- Support signature verification (for secure boot and firmware upgrade).
- Key storage and wrapping and unwrapping of local or remote keys.
- Generate on-chip key pairs.
- Establish a secure channel with a remote host including *Transport Layer Security* (TLS) handshake.
- Application usage monitoring with secure counters.
- Authenticate peripherals.
- Attestation.

It is essential to TBSA compliance that external secure elements robustly pair and create a secure channel with the host SoC. The secure element should reside within the Trusted world. See Section 4.6 for requirements.

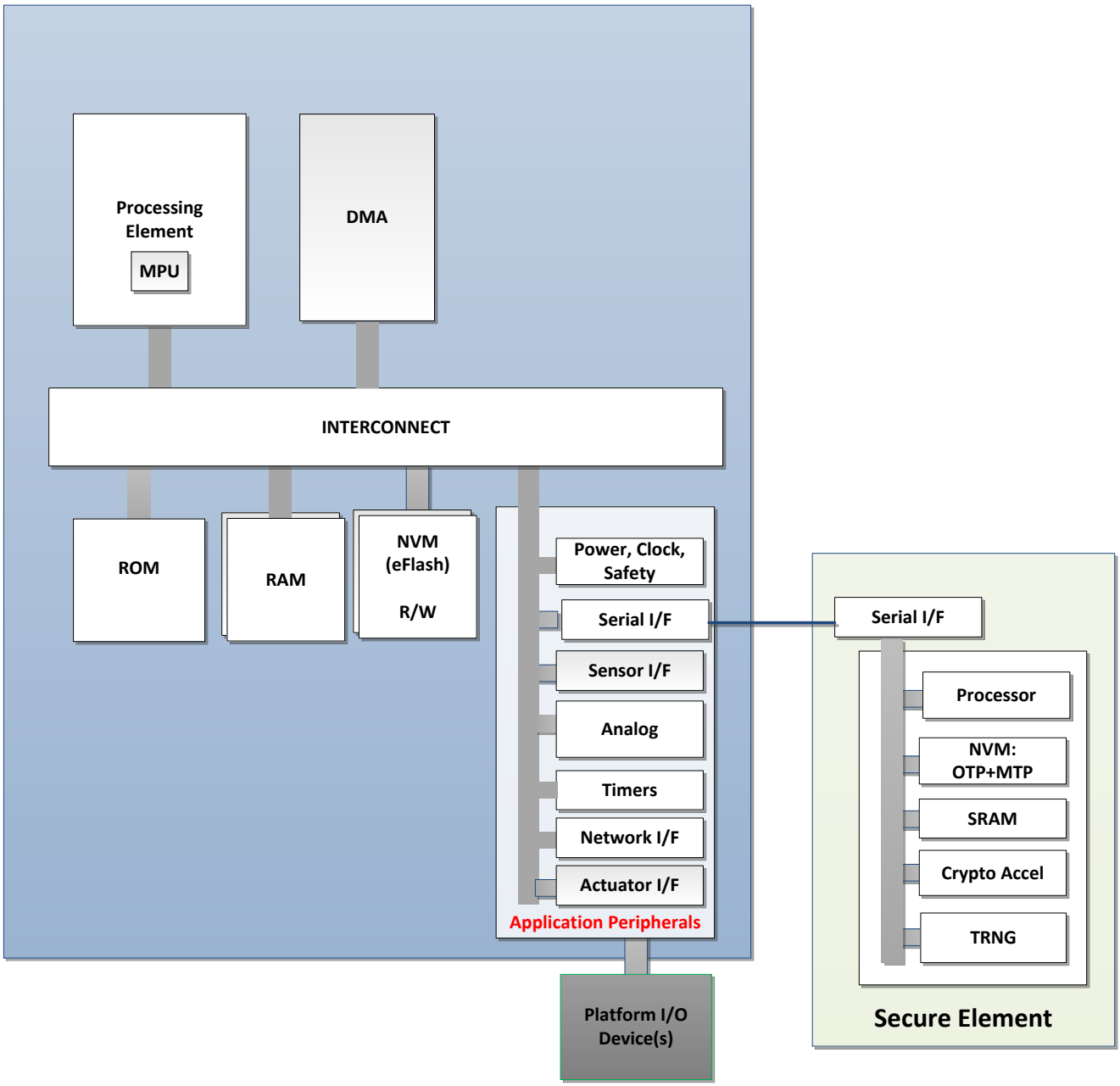

Figure 9: Example of a simple TBSA-M SoC with an external security element

4.6 PSA isolation levels

TBSA-M requires hardware support for isolation of software in accordance with the PSA firmware framework. Figure 10 illustrates the isolation levels:

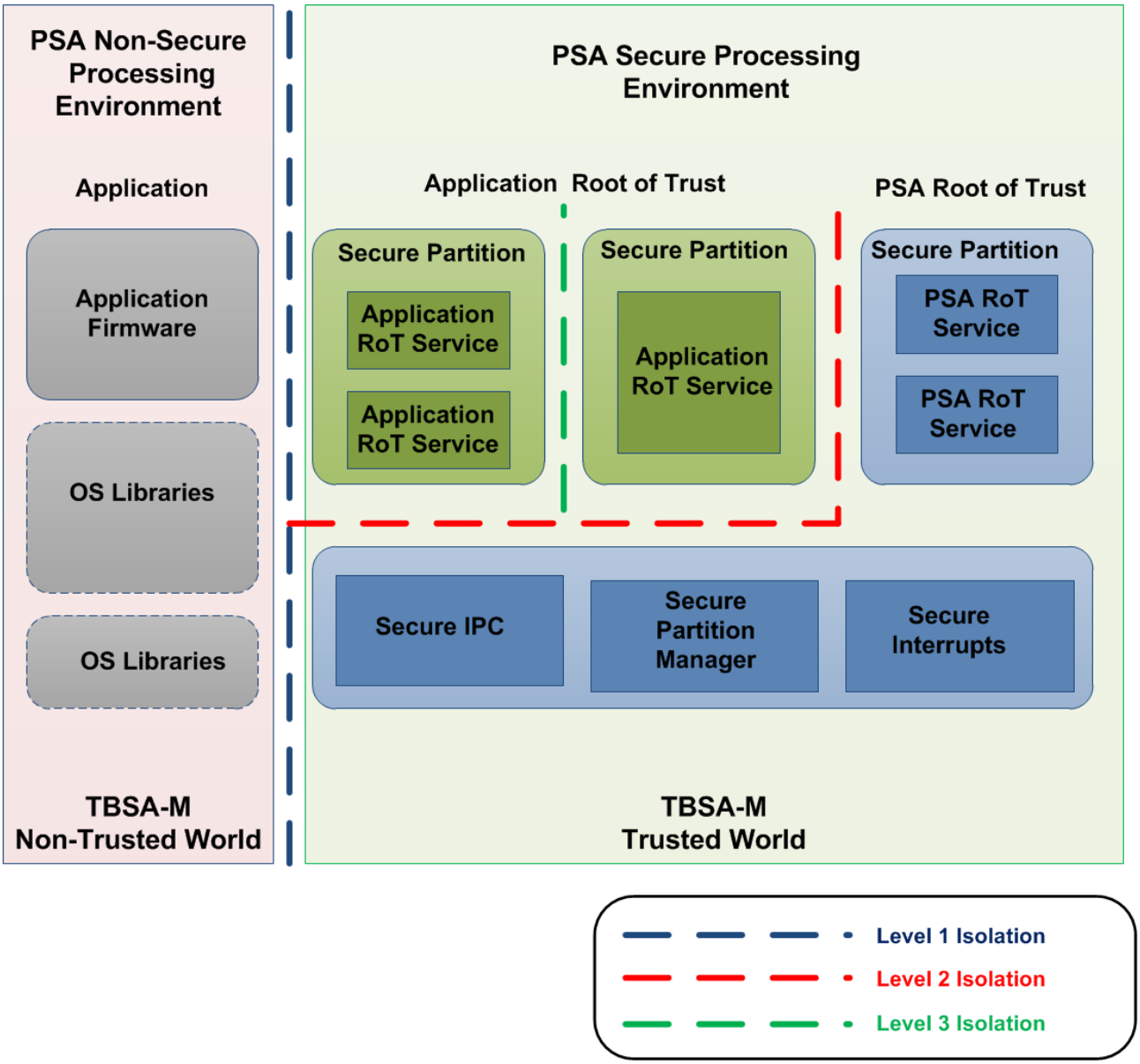

Figure 10: PSA isolation boundaries

In PSA, the SPM is responsible for the isolation of the SPE from the NSPE, the PSA RoT from the Application RoT, and secure partitions within the SPE.

The isolation must be enforced by platform hardware throughout the system using master-side or slave-side filters, for example, the SAU and MPU in an Armv8-M CPU. This enforcement prevents other bus masters from

bypassing the isolation, and also mitigates software errors that are manifest in the NSPE from having the same effect.

The hardware filters that enforce isolation must be configured by the SPM or the secure boot firmware before the SPM runs. The filters must not be accessible to the NSPE or any Secure Partitions that are isolated from the SPM.

A PSA implementation fully isolates every secure partition, so that each partition only accesses its own data and peripherals, and only the SPM accesses the whole system.

Increased isolation improves the security and robustness of the system, by reducing its vulnerability to software defects. However, these benefits come at the expense of additional hardware, memory, performance or energy. The PSA Security Model specifies multiple levels of isolation, in order to support implementations with different security, performance and cost trade-offs. Table 2 below provides a summary of the three supported isolation levels:

Table 2: PSA isolation security levels

Isolation level	Description
Level 1	SPE isolation Two security domains SPE is protected from access by Non-secure application firmware and hardware
Level 2	PSA RoT isolation Three security domains In addition to Level 1, the PSA RoT is also protected from access by the Application RoT
Level 3	Maximum firmware isolation Three or more security domains In addition to Level 2, each Secure Partition is sandboxed and only permitted to access its own resources. This protects each Secure Partition from access by other Secure Partitions and protects the SPM from access by any Secure Partition.

Arm recommends that RoT Service software that is to be placed in a secure partition is designed to run with Level 3 isolation and does not assume that data is shared with another Secure Partition or the NSPE. This design increases the portability of firmware to run on multiple PSA implementations and reduces the risk of introducing vulnerabilities related to the sharing of data.

Although the PSA firmware isolation levels are a useful indicator of the platform's security capability, it does not include all forms of isolation that the platform provides. Many platforms also use temporal isolation, in which resources are only available within a specific time window, for example, during boot. The additional security provided by Trusted Subsystems does not form part of the specification of TBSA-M, but may be used as countermeasure for a threat model posted by a particular application (see Section 4.5).

4.7 Basic architecture

The Basic architecture performs most of the security functions within Trusted world software on the host processor. It is supported by a minimum set of required security hardware, for example:

- Trusted Boot ROM.
- Trusted RAM, Trusted External Memory Partitioning, or both.

- Trusted peripherals:
    - OTP Fuses, for secrets, counters, lifecycle states, etc.
    - Entropy source.
    - Timer.
    - Watchdog.

The Basic architecture ensures that the Trusted world software has access to all the assets it requires, and has the underlying mechanisms to protect the integrity, confidentiality, and authenticity of the Trusted world. The Trusted world software exports cryptographic services to the Non-Trusted world, and supports the execution of Trusted services by, for example, implementing an environment that can run Trusted applications.

4.8 Assisted architecture

An Assisted architecture is a basic system that is supplemented with one or more Trusted subsystems. Assisted architectures may provide countermeasures which extend beyond the scope of TBSA-M.

An Assisted architecture builds on the Basic architecture by adding hardware to accelerate and offload some of the cryptographic operations from the Trusted world software, and to provide increased protection to high value assets, for example root keys.

The cryptographic accelerators support the most commonly used algorithms for encryption, decryption, and authentication, for example AES, TDES, SHA, RSA, and ECC.

Arm recommends increasing protection for the keys in the system by implementing a hardware Key Store. The Key Store enables use of the keys by cryptographic accelerators but prevents the keys from being read by both Non-Trusted and Trusted software.

Assisted architectures can also contain hardware for governing life-cycle state transitions and enforcing lifecycle state policies.

An assisted architecture may provide a hardware-initiated response to detect tamper events, as well as hardware countermeasures for:

- Invasive attacks, for example probing.
- Side channel attacks, for example power and electromagnetic emission analysis.
- Perturbation attacks, for example clock or voltage manipulation.

5 TBSA-M security requirements [Normative]

TBSA-M requirements form a convenient checklist that can be used to support a pre-silicon security review of a design that is based on an M-profile processor. The requirements are based on principles of good security practice as applied to target platforms Section 1.3. Section 1.5 describes the significance of requirements and recommendations and what constitutes a claim of compliance for TBSA-M.

TBSA-M devices require support for PSA isolation levels.

R001_TBSA_BASE *The SoC must provide a hardware-based mechanism for separating the Trusted world from the Non-Trusted world.*

This requirement supports PSA level 1 isolation (see section 4.6).

R002_TBSA_BASE *The SoC must provide a hardware-based mechanism which is able to separate partitions within the Trusted world.*

This requirement enables firmware to support PSA isolation level 2 and also enables firmware to support PSA level 3 isolation (see Section 4.6).

Arm recommends that SoCs provide a hardware-based mechanism so that secure partitions within the Application RoT are isolated from each other. This recommendation addresses PSA level 3 isolation.

Examples of how R001_TBSA_BASE and R002_TBSA_BASE may be met are given in Chapter 4.

5.1 System view

At an abstract level, the TBSA-M is a system that comprises a collection of assets, together with operations that act on those assets. In this context, an asset is a data set that has an owner and a particular intrinsic value, for example a monetary value. All data sets are assets associated with a value, even if that value is notionally zero. A data set is any stored or processed information, including executable code as well as the data on which it operates.

High value assets that require protection should belong to the Trusted world, while lower value assets that do not require protection may belong to the Non-Trusted world. The actual classification, ranking, and mapping of assets to worlds depends on the target specifications, and is outside the scope of this document.

Similarly, an operation belongs to a world and is therefore classified as either Trusted or Non-Trusted.

R010_TBSA_BASE *A Non-Trusted world operation must only access Non-Trusted world assets.*

R020_TBSA_BASE *A Trusted world operation may access both Trusted and Non-Trusted world assets.*

As described in Chapter 4, some TBSA-M architectures will be built around TrustZone for Armv8-M. In this case, code executing on an Armv8-M PE with the security extension exists in one of two Security states, Secure state or Non-secure state. Secure state corresponds to Trusted world operations, and the Non-secure state corresponds to Non-Trusted world operations. When in the Secure state, an Armv8-M PE with TrustZone can access state attributed as both Secure and Non-secure.

R030_TBSA_BASE *A SoC using TrustZone Isolation must be based on an Armv8-M architecture PE with the Security Extension and MPUs implemented.*

In order to support common embedded OSs in a convenient manner, Arm recommends implementing both secure and non-secure MPUs, with each supporting a minimum of eight regions. Arm also recommends that the SAU is implemented, with a minimum of four regions. This is to give sufficient flexibility to implement common Armv8-M security attribution address map schemes.

Arm recognizes that the security features of a TBSA-M device will not be entirely implemented in hardware, and that the hardware might be configurable by software.

R040_TBSA_BASE *The hardware and software of a TBSA-M device must work together to ensure that all the security requirements are met.*

5.2 Infrastructure

The TBSA-M is underpinned by a hardware infrastructure that provides strong isolation between the operations and assets of the Trusted and Non-Trusted worlds.

The processor is not the only key component of a larger SoC design that performs operations on stored assets within the wider system. In such a system, storage comprises registers, random access memory, and non-volatile memory. To provide the required protection for assets, the storage is divided, either physically or virtually, into two types: Secure and Non-secure. These types correspond to the Trusted and Non-Trusted worlds, respectively.

5.2.1 Memory system

Operations and assets are connected by transactions, in which a transaction represents read or write access to storage containing the asset. Each transaction originates from either the Trusted world or Non-Trusted world.

As described in Chapter 4, the processor sees the memory map as two spaces, Secure and Non-secure storage, in which Trusted world assets are held in Secure storage and Non-Trusted world assets are held in Non-secure storage.

To build a useful system, it is necessary to facilitate communication between the two worlds through shared memory. In the TBSA-M, this permits a Trusted operation to issue both Secure and Non-secure transactions. The opposite, however, is not true. A Non-Trusted operation can only issue Non-secure transactions.

R010_TBSA_INFRA *A Trusted operation can issue Secure or Non-secure transactions.*

R020_TBSA_INFRA *A Non-Trusted operation must only issue Non-secure transactions.*

Note: There are special cases in TrustZone for Armv8-M systems which do not comply with the above requirements:

Non-secure code calling a Secure API requires secure instruction fetches

If cache memory is present, Non-secure operations might lead to cache eviction of Secure data. The data will not be accessible from Non-trusted world.

As described in Chapter 4, Arm recommends adopting a consistent system-wide approach, so that Secure transactions only access Secure storage, and Non-secure transactions only access Non-secure storage. A recommended memory system design is described in [5] on page ix.

R030_TBSA_INFRA *A Non-secure transaction must only access Non-secure storage.*

The following requirements summarize the link between operations, transactions and storage:

- A Non-Trusted operation operates in a Non-secure state and only issues Non-secure transactions targeting Non-secure storage locations. It must not issue Secure transactions, and therefore cannot access Trusted assets.
- A Trusted operation operates in a Secure state and can issue either Secure or Non-secure transactions. As such, it can access both Secure and Non-secure storage. However, Arm recommends that a Secure transaction only access Trusted assets and that a Non-secure transaction only access Non-Trusted assets.

In practice, memory modules (RAM or DRAM) are often split into two types of regions, Secure and Non-secure. To map the regions correctly into the larger physical address map, remapping logic is implemented. In simple implementations, this is fixed logic, but it can also be programmable logic, which offers greater flexibility if

software is updated. In the latter case, the relevant configuration registers must only be accessible to Secure transactions and belong to the Trusted World.

R040_TBSA_INFRA *If programmable address remapping logic is implemented in the interconnect, then its configuration must be possible only from the Trusted world.*

Arm recommends that any programmable address remapping logic be programmable in secure privileged state.

In TrustZone for Armv8-M systems, the mapping of resources into Secure or Non-secure memory is achieved using either fixed or programmable logic. However, a more optimal solution uses a target-based filter. Such a filter divides the memory into many regions and enables each memory region to be defined by software configuration as either Secure or Non-secure. Access is only permitted to a region if:

- The region is Secure and ADDRESS.NS = 0.
- The region is Non-secure and ADDRESS.NS = 1.

The physical address space after the filter, which does not consider ADDRESS.NS, is consequently halved in size.

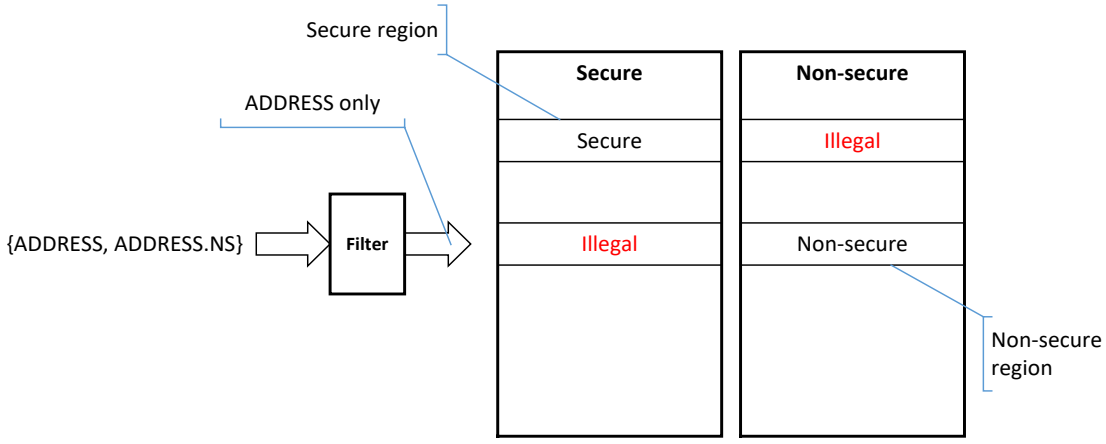

Figure 11 shows the resulting address map:

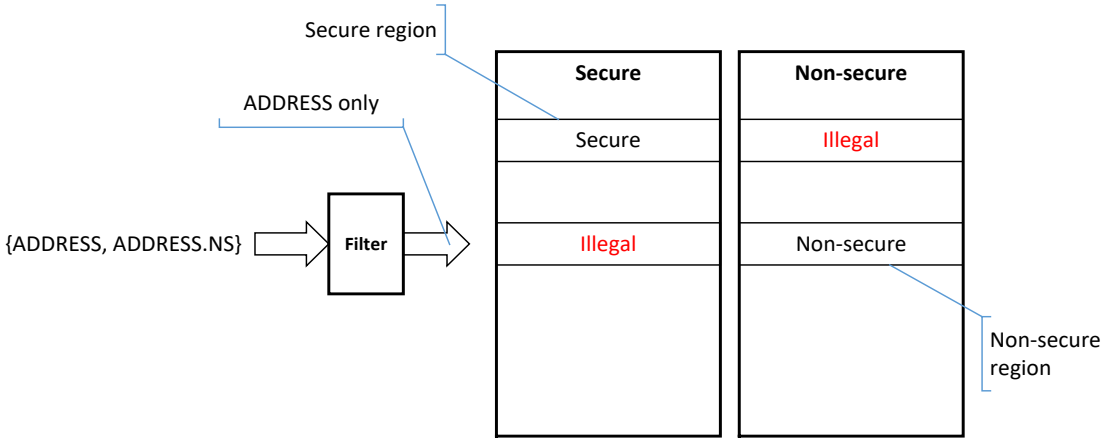

Figure 11: Filter aliasing

The aliasing in the address map after filtering constrains the memory layout from the point of view of a bus master, for example an Arm processor.

R050_TBSA_INFRA *A unified address map that uses target side filtering to disambiguate Non-secure and Secure transactions must only permit all Secure or all Non-secure transactions to any one region. Secure and Non-secure aliased accesses to the same address region are not permitted.*

R060_TBSA_INFRA *The target transaction filters configuration space must only be accessed from the Trusted world.*

At the interconnect level, before filtering, ADDRESS.NS forms an additional address bit. Each memory transaction must transport this bit, together with all other address bits, to the point at which the filter constraints are applied.

Note: When using legacy interconnects for example the *Advanced Peripheral Bus* (APB) v3 or earlier, the peripheral bus does not support an ADDRESS.NS bit. In this case, it is necessary to perform filtering before a transaction reaches the bus. An example is a bus bridge joining *Advanced eXtensible Interface* (AXI) and APB.

The Arm TrustZone protection controller components of Arm CoreLink SIE-200 contain implementations of highly flexible target-based filters.

Arm TrustZone protection controllers are configured to silently block illegal transactions, or to block and signal a security exception through a bus error or interrupt. If an interrupt is generated, it is classified as a Trusted interrupt, as described in Section 5.2.3. In any event, illegal transactions must be prevented from reading or writing to memory.

R070_TBSA_INFRA *Security exception interrupts must be wired or configured as Secure interrupt sources.*

For an Arm processor with TrustZone for Armv8-M, the Security state of the transaction is available at the boundary of the processor so that it is propagated through the on-chip interconnect. For example, in an AXI bus implementation, the Security state of the transaction, ADDRESS.NS, is mapped to the ARPROT[1] and AWPROT[1] signals so that:

- ARPROT[1] indicates a Trusted read when low.
- AWPROT[1] indicates a Trusted write when low.
- An AXI bus master will generate the same signals to indicate the Security state of each transaction.

In some interconnect designs, it is possible to re-configure the routing of packets to arrive at a different interface. Although the access address remains unchanged, this is dangerous and can lead to an exploitation mechanism. Such configuration a is only possible from the Trusted World using Secure transactions.

R080_TBSA_INFRA *Configuration of the on-chip interconnect that modifies routing or the memory map must only be possible from the Trusted world, unless it is not possible for such modifications to affect secure transactions.*

These techniques for address remapping and filtering are methods of constraint that bind storage locations to worlds.

It is possible to have world-aware peripherals, in which the peripheral is visible in both Trusted World and Non-Trusted World address aliases at the same time. It is also possible for that peripheral to use security attribute signals to determine if the access is from Trusted World, using a Trusted address alias, or from Non-Trusted world, using a Non-Trusted address alias. This arrangement does not use filters, but the Secure aliases of the peripheral address space must be in an XN (execute never) region.

Whatever the method of constraint, a memory transaction must not be able to bypass it.

One example is in TBSA-M systems which implement multiple caches that are upstream from a target filter and are synchronized through a coherency mechanism. If such a mechanism, for example bus snooping, is implemented, then the mechanism must force a coherency transaction to pass through the target filter.

R090_TBSA_INFRA *All transactions must be constrained; it must not be possible for a transaction to bypass a constraining mechanism.*

5.2.2 Shared volatile storage

Sometimes assets from different worlds occupy the same physical volatile storage location. In this case, the underlying storage, for example internal RAM, external RAM, or peripheral space, is called shared volatile storage. Because of the requirement to mitigate the leakage of assets, Arm recommends avoiding shared volatile storage whenever possible.

A shared volatile storage implementation enables a storage location or region, which previously held a Trusted asset, to hold a Non-Trusted asset. Before such a storage location or region is reallocated from Trusted to Non-Trusted, the Trusted asset must be securely removed. This is achieved using scrubbing.

Scrubbing is the atomic process of overwriting a Trusted asset with an unrelated value, which is either a constant, a Non-Trusted asset value, or a randomly generated number of the same size. In this situation, the term atomic indicates that the process must not be interrupted by the Non-Trusted world.

R100_TBSA_INFRA *If shared volatile storage is implemented, then the associated location or region must be scrubbed, before it can be reallocated from Trusted to Non-Trusted.*

Conversely, a shared volatile storage implementation enables a storage location or region. which previously held a Non-Trusted asset, to hold a Trusted asset. Take care to mitigate the threat of escalation of privilege. Before such a storage location or region is reallocated from Non-Trusted to Trusted, the storage must not be marked as executable by a PE in the Secure state. If an Armv8-M processor is used, this is achieved by ensuring that the storage is not attributed as NSC, and that it is marked as execute-never (XN) by the secure MPU when it is reallocated. If the storage is subsequently scrubbed, it might be made executable and attributed NSC.

R110_TBSA_INFRA *If shared volatile storage is implemented in a TrustZone for Armv8-M based system, then the associated location must not be executable or NSC immediately after it is reallocated from Non-Trusted to Trusted.*

Note: When a copy of Trusted data is held in a cache, the implementation must not permit any mechanism that provides the Non-Trusted world with access to that data. If a hardware engine is used for scrubbing, pay careful attention to the sequencing of operations, to ensure that the relevant cached data is flushed and invalidated before the scrubbing operation. This situation also applies to all hardware registers of any shared device. Even if there are no shared registers, the hardware must ensure there are no leaks between worlds. If the changing of some state of one world caused a changing in the state of another, secrets might be inferred.

5.2.3 Interrupts

In most cases, a Trusted interrupt must not be visible to a Non-Trusted operation, in order to prevent information leaks that might be useful to an attacker. Consequently, the on-chip interrupt network must be able to route any interrupt to any world. However, the routing of Trusted interrupts must only be configured from the Trusted world.

R120_TBSA_INFRA *An interrupt originating from a Trusted operation must by default be mapped only to a Trusted target. By default, this must be the case following a system reset.*

R130_TBSA_INFRA *Any configuration to mask or route a Trusted interrupt must only be carried out from the Trusted world.*

R140_TBSA_INFRA *The interrupt network may be configured to route an interrupt originating from a Trusted operation to a Non-Trusted target.*

R150_TBSA_INFRA *Any status flags recording Trusted interrupt events must only be readable from the Trusted world, unless specifically configured by the Trusted world to be readable by the Non-Trusted world.*

These requirements permit a Non-Trusted world request to a Trusted operation to deliver a Trusted Interrupt to a Non-Trusted target, which signals the end of the operation.

This Configuration of the interrupt is done by the Trusted world before or during the Trusted operation.

Handle these operations carefully. Arm recommends that designs compliant with TBSA-M ensure that, if a requirement allows the Non-Trusted world to trigger Secure interrupts, the hardware arrangement only allows the dedicated Secure interrupt to be triggered from the Non-Trusted side. The Secure interrupt handler must be written carefully, in order to avoid denial of service attacks.

In the Arm architecture, these requirements are supported using the NVIC interrupt controller block.

5.2.4 Secure RAM

In this document, Secure RAM refers to one or more regions of memory that are dedicated to the Trusted world and that are mapped onto one or more physical RAMs. In a TBSA-M system, Trusted code is expected to execute from Secure RAM and may also store high value assets within the Secure RAM. When a physical RAM block is not entirely dedicated to Secure RAM, that physical RAM block is said to be shared between worlds. The manner of this sharing needs to be done carefully to avoid security vulnerabilities.

A flexible implementation of Secure RAM consists of blocks of RAM that all default to being mapped into the Trusted world at boot, until Trusted software partitions it between secure and non-secure use.

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

Example Secure RAM use cases are:

- Secure boot code and data.
- A Secure OS.
- Cryptographic services.
- Trusted services, execution environments which conform to particular security standards and *Trusted Applications* (TAs).

The Secure RAM size depends on the target requirements and is not specified in this document. Many TBSA-M systems will meet all their RAM requirements on-chip, using one or more independent banks.

R160_TBSA_INFRA *A TBSA-M system must integrate a Secure RAM.*

R170_TBSA_INFRA *Secure RAM must be mapped into the Trusted world only.*

R180_TBSA_INFRA *If the mapping of Secure RAM into regions is programmable, then configuration of the regions must only be possible from the Trusted world.*

Note: If Secure RAM is re-mapped from the Trusted world to the Non-Trusted world, it is classified as shared volatile storage, and it must meet the requirements of shared volatile storage.

Many TBSA-M devices use embedded resources for both RAM and *non-volatile memory* (NVM). However, some devices might use DRAM, and some might use external NVM, for example serial flash.

5.2.5 Power and clock management

Modern battery-powered platforms have a high degree of power control and might integrate an advanced power management subsystem using dedicated hardware and execute a small software stack from local RAM. In such cases, the management subsystem has control over a number of Trusted assets, for example:

- Clock generation and selection. Examples include:
  - *Phase-locked loops* (PLLs).
  - Clock dividers.
  - Glitch-less clock switching.
  - High-level clock gating.
- Reset generation. Examples include:
  - Registers to enable or disable clocks.
  - State machines to sequence the assertion and de-assertion of resets in relation to clocks and power states.
  - Re-synchronization of resets.
- Power control. Examples include:
  - Access to an off-chip power controller, switch, or regulator.
  - State machines for sequencing when changing power states.
  - Logic or processing to intelligently apply power states either on request, or dynamically.
- State saving and restoration. To dynamically apply power states, some subsystems can also perform saving and restoration of system states without the involvement of the main application processor.

Unrestricted access to this functionality is dangerous, because it could be used by an attacker to induce a fault that targets a Trusted service by, for example, perturbing a system clock. To mitigate this threat, the advanced power, clock and reset mechanisms belong in the Trusted world. The system must also integrate a Trusted management function, to perform policy checks on any requests from the Non-Trusted World, before they are applied.

This approach still permits execution of most Non-Trusted complex peripheral wake up code from the Non-Trusted world.

R190_TBSA_INFRA *The advanced power mechanism must integrate a Trusted management function to control clocks and power. It must not be possible to directly access clock and power functionality from the Non-Trusted world.*

The power and clock status are made available to the Non-Trusted world through APIs that exist in the Trusted world.

Note: All system clocks are classified as Trusted because they can only be configured using the Trusted manager.

Non-secure peripherals can have their own local clock and power control accessible to the Non-Trusted world, if such control is independent from the system clocks and power.

5.2.6 Peripherals

A peripheral is a hardware block with an operation supervised by a processor. It does not execute modifiable firmware. A peripheral implements one or more operations that act on assets. It has an interface to receive commands and data from one or more processors. Some peripherals are capable of direct memory access.

Depending on the role of a simple peripheral in a particular use case, the wider system can map the operations of that peripheral into one world or the other.

R210_TBSA_INFRA *If access to a peripheral, or a subset of its operations, is dynamically switched between Trusted world and Non-Trusted world, then this must only be done under the control of the Trusted world.*

A Non-Trusted peripheral acts only on Non-Trusted assets. A Trusted peripheral can act on assets in both worlds. Complex peripherals act in both worlds, supporting both Trusted and Non-Trusted operations, as illustrated in Figure 12:

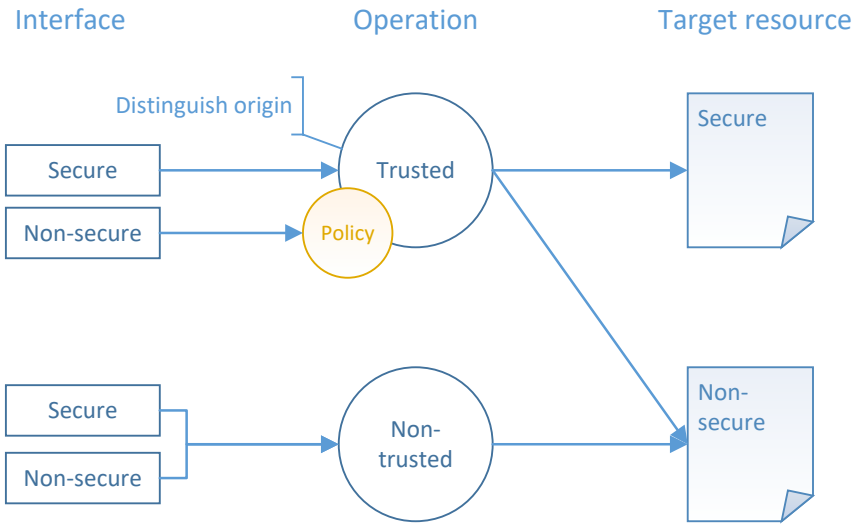

Figure 12: Peripheral operations

A Trusted peripheral is a hardware block that implements at least one Trusted operation. Each operation has an interface that is mapped into the Trusted or Non-Trusted world, or into both worlds.

A Trusted peripheral identifies the world in which a request originates. This identification ensures compliance with the general requirement that operations originating from the Non-Trusted world cannot access Trusted world resources (see Section 5.1).

Non-Trusted world software must not bypass Trusted world policies by using peripherals to access Trusted world assets on its behalf. Some designs are subject to threat models in which particular hardware IP blocks may have unknown or undesirable behaviors. In these cases, use additional master side filters under sole control of the Trusted world to ensure that such IP cannot access Trusted world assets beyond that authorized by a Trusted world policy.

The implementation of the operations is a design choice. The operations are built using fully separate hardware or using the multiplexing of shared functions and resources.

A Trusted peripheral must meet the following requirements, which are framed in terms of its operations:

R220_TBSA_INFRA *If the peripheral stores assets in local embedded storage, a Non-Trusted operation must not be able to access the local assets of a Trusted operation.*

R230_TBSA_INFRA *A Trusted operation must be able to distinguish whether commands and data were received at an interface accessible to the Trusted world only, or at an interface accessible to the Non-Trusted world.*

R240_TBSA_INFRA *A Trusted operation that exposes a Non-secure interface must apply a policy check to the Non-Trusted commands and data before acting on them. The policy check must be atomic and, following the check, it must not be possible to modify the checked commands or data.*

An example policy for a cryptographic accelerator peripheral would cover at least:

- The world the input data can be read from.
- The world the output data can be written to.
- Whether encryption is permitted.
- Whether decryption is permitted.

A specific example is a DMA engine that is shared between worlds. When configured from the Trusted world, the DMA can operate on Trusted and Non-Trusted memory, by appropriate use of the NS bit. However, when configured from the Non-Trusted world, the DMA must only operate on Non-Trusted memory, using an NS value of 1.

5.3 Fuses

TBSA-M devices require non-volatile storage to store a range of data across power cycles. What is stored varies from device firmware to cryptographic keys and system configuration parameters. Fuses often control life cycle state management and the debuggability of the device.

Non-volatile storage can use a variety of technologies, including floating gate memories or oxide-breakdown antifuse cells. These technologies vary with respect to certain properties, most notably whether they are OTP or *Many-Time-Programmable* (MTP).

Not all types of non-volatile storage technologies are available in all semiconductor processes. For example, floating gate memories are not economic in some bulk CMOS processes. Where required, off-chip non-volatile memory can augment the available on-chip non-volatile storage.

Non-volatile storage technologies sometimes require error correction mechanisms, in order to ensure the correct storage of data during the lifespan of the device.

R010_TBSA_FUSE *A non-volatile storage technology must meet the lifetime requirements of the device, either through its intrinsic characteristics, or through the use of error correction mechanisms.*

Most security assets and settings that need to be stored on-chip require OTP non-volatile storage, in order to ensure that values cannot be changed. Following the industry norm, this document uses the term fuse to refer to on-chip OTP non-volatile storage. A fuse is implemented using an antifuse or an MTP technology with controlling logic to make it OTP.

These are the fundamental requirements for implementing fuses in a TBSA-M device:

R020_TBSA_FUSE *A fuse is permitted to transition in one direction only, from its un-programmed state to its programmed state. The reverse operation must be prevented.*

R030_TBSA_FUSE *A fuse must only be programmed in accordance with its specified mechanism so that its reliable operation is not at risk.*

R040_TBSA_FUSE *It must be possible to blow at least a subset of the fuses when the device has left the silicon manufacturing facility.*

R050_TBSA_FUSE *All fuse values must be stable before any parts of the SoC that depend on them are released from reset.*

R060_TBSA_FUSE *Fuses that configure the security features of the device must be configured so that the programmed state of the fuse enables the feature. That is, the programming of a security configuration fuse always increases security within the SoC.*

Note that R060_TBSA_FUSE is intended to ensure that a security feature cannot be deactivated after it is enabled.

R070_TBSA_FUSE *Lifetime guarantee mechanisms to correct for in-field failures must not indicate which fuses have had errors detected or corrected, just that an error has been detected or corrected. This indicator must only be available after all fuses have been checked.*

Full error information, including error information that indicates which erroneous bits have been detected and corrected, is available to the lifetime guarantee mechanism itself. Arm recommends not disclosing this full information, because it might assist an attacker. The security of the mechanism implementation must be considered. Arm recommends implementing the mechanism in hardware, but this might not be practical in some cases. Arm recommends performing software lifetime guarantee mechanisms soon after boot, so that the error indications are set. Arm also recommends disabling the visibility of the full error information until the next boot.

Assets stored in fuses have a variety of characteristics that determine how they are accessed. The main types of fuses and their characteristics are summarized as follows:

- Confidential fuses: only read by the intended recipient, or a particular hardware module or software process.
- Public fuses: can be accessed by any piece of software or hardware.
- Lockable fuse: must comply with one of the following requirements:
    - They must prevent re-writing of a locked value.
    - A mechanism that prevents the programming of a fuse bit or group of fuse bits is implemented by reserving an additional fuse bit to act as a lock bit.
    - Writing the value is followed by its lock bit being set. Glue logic ensures that no further programming is possible.
    - Writing zero, which corresponds to the un-programmed fuse state, causes no value to be written. It causes only the lock bit to be set.
    - Tamper detection used to detect that the value has been modified.
    - A tamper protection mechanism is implemented by storing a code in additional fuses that is sufficient to detect any modification to the value:
        - Writing the value is followed by storing the detection code.
        - When the value is read by the system, a mechanism must recalculate the code from the value and compare it with the stored code.
        - If the codes do not match, the value must not be returned to the system.
- Open fuse: programmed only once, at any point in the device lifetime.
- Bitwise fuse: programmed one logical bit at a time, regardless of the number of fuses required to store the value.
- Bulk fuse: stores multi-bit values that must be programmed at the same time and are treated as an atomic unit.

In the deployed lifecycle state, bitwise and bulk fuses must also comply with the following requirements:

R080_TBSA_FUSE *A confidential fuse whose recipient is a hardware IP must not be readable by any software process.*

R090_TBSA_FUSE *A confidential fuse whose recipient is a hardware IP must be connected to the IP using a path that is not visible to software or any other hardware IP.*

Usually, this is implemented as a direct wire connection.

R100_TBSA_FUSE *A confidential fuse whose recipient is a software process might be readable by that process and must be readable by privileged software.*

R100_TBSA_FUSE permits a kernel level driver to access fuses for a user space process. The confidentiality relies on the kernel level driver only passing fuse values to the correct user space process.

R110_TBSA_FUSE *A confidential fuse whose recipient is a Trusted world software process must be protected by a hardware filtering mechanism that can only be configured by secure software, for example an NS-bit filter.*

R120_TBSA_FUSE *It must be possible to fix a lockable fuse in its current state, regardless of whether it is programmed or un-programmed.*

R130_TBSA_FUSE *The locking mechanism for a lockable fuse can be shared with other lockable fuses, depending on the functional requirements.*

An example of R130_TBSA_FUSE is a single locking mechanism for all fuses that are programmed by the silicon vendor.

R140_TBSA_FUSE *A bulk fuse must also be a lockable fuse to ensure that any unprogrammed bits cannot be programmed later.*

R150_TBSA_FUSE *Additional fuses that implement lifetime guarantee mechanisms must have the same confidential and write lock characteristics as the logical fuse itself.*

5.4 Cryptographic keys

Fundamental to the security of a system are the cryptographic keys that provide the authenticity and confidentiality of the assets that the system uses.

It is important to treat a key as an atomic unit when it is created, updated, or destroyed. This principle applies at the level of the requesting entity. Replacing part of a key with a known value, and then using that key in a cryptographic operation, makes it much easier for an attacker to discover the key using a divide and conquer brute-force attack. This is especially relevant when a key is stored in memory units that are smaller than the key. An example of this principle is a 128-bit key that is stored in four 32-bit memory locations. Entities, for example Trusted firmware functions, that implement key creation, updating or destruction services should ensure that their clients cannot observe or use keys in a way that breaks the assumption of atomicity.

R010_TBSA_KEY *A key must be treated as an atomic unit. It must not be possible to use a key in a cryptographic operation before it has been fully created, during an update operation, or during its destruction.*

R020_TBSA_KEY *Any operations on a key must be atomic. It must not be possible to interrupt the creation, update, or destruction of a key.*

R030_TBSA_KEY *When a key is no longer required by the system, it must be put beyond use to prevent a hack at a later time from revealing it.*

If a key is put beyond use, it must be impossible to use or access it. This is achieved either by hiding the key through blocking access to it, or by removing the key from the system through scrubbing the storage location that contains the key.

Keys have a range of characteristics that influence the level of protection to be applied, and how a key is used.

5.4.1 Cryptographic schemes

A cryptographic scheme provides one or more security services, and is based on a purpose and an algorithm requiring specific key properties and key management.

Keys are characterized as private, public or symmetric, according to their classification and use.

Broadly, each key should be used for a single purpose, for example encryption, digital signature, integrity, or key wrapping. The main motivations for this principle are:

- Limiting the uses of a key limits the potential harm if the key is compromised.
- The use of a single key for two or more different cryptographic schemes reduces the security provided by one or more of the processes.
- Different uses of a single key can cause conflicts in how each key is managed. For example, keys used in different cryptographic processes may have different lifetimes. In this case, a key may be retained longer than is best practice for one or more uses of that key.

If a scheme can provide more than one cryptographic service, this principle does not prevent use of a single key. An example is use of a symmetric key both to encrypt and to authenticate data in a single operation. Another example is use of a digital signature to provide both authentication and integrity.

Re-using part of a larger key in a scheme that uses a shorter key, or using a shorter key in a larger algorithm and padding the key input, can leak information about the key, so these practices, too, are prohibited.

R035_TBSA_KEY *A key must only be used by the cryptographic scheme for which it was created.*

5.4.2 Static and ephemeral keys

Different keys in the same the system can have very different lifespans. These lifespans are also known as cryptoperiods. Some keys are programmed during SoC manufacture and never change, while others will exist only during a communication session.

A static key is a key that cannot change after it is introduced to the device. It is stored in an immutable structure like a ROM, or in a set of fuses. Although a static key cannot have its value changed, this does not preclude it from being revoked or made inaccessible by the system.

R070_TBSA_KEY *A static key must be stored in an immutable structure, for example a ROM or a set of bulk-lockable fuses.*

Ephemeral keys have a short lifespan. In many cases, they only exist between power cycles of the device. Ephemeral keys are created in the device in a number of ways.

- Derivation: sometimes it is useful to create one or more keys from a source key. This method is called key derivation. Derivation is usually used to create ephemeral keys from static keys.

  A key derivation operation must use a cryptographic one-way function that preserves the entropy of the source key, and the operation must be unique for each derived key. Common derivation constructions are based on use a keyed *Hash Message Authentication Code* (HMAC) or a *Cipher-based Message Authentication Code* (CMAC). Refer to the recommendations for key derivation in [18] for a detailed treatment.

Collectively, the inputs to the one-way derivation function are referred to as source material.

R080_TBSA_KEY *To prevent the re-derivation of previously used keys, only Trusted code can have access to all of the source material.*

This requirement allows Non-Trusted code access to part of the source material, provided that this is insufficient to re-derive previously used keys.

- Injection: A key is introduced into the system from storage or through a communication link. One example is the key in a license certificate. To ensure that the key is encrypted during transit, the injection is often protected by another key.
- Generation: Ephemeral keys are generated on the device by simply sampling random numbers or by using random numbers to create a key, for example in a Diffie-Hellman key exchange protocol.

When an ephemeral key is no longer required, it must be removed securely from the system. This must happen even if the event that makes the key redundant is unexpected.

R090_TBSA_KEY *If an ephemeral key is stored in memory or in a register in clear text form, the storage location must be scrubbed before being used for another purpose.*

5.4.3 Device unique and common keys

It is important to distinguish two types of keys.

Device unique: A device unique key is statistically unique for each device, so the probability of another device having the same key value is insignificant. For TBSA-M systems, a key with at least 128-bits of entropy is considered to be sufficient for device uniqueness.

Common - A common key is present on multiple devices. Common keys are sometimes referred to as class keys.

5.4.4 Source

Different keys in the same the system are restricted in their domain of operation in order to further isolate Trusted world assets from those in the Non-Trusted world.

Non-Trusted world:

R100_TBSA_KEY *A key that is accessible to, or generated by, the Non-Trusted world must only be used for Non-Trusted world cryptographic operations, which are operations that are either implemented in the Non-Trusted world, or have both operands and results in the Non-Trusted world.*

Trusted world:

R110_TBSA_KEY *A key that is accessible to, or generated by, the Trusted world can be used for operations in both Non-Trusted and Trusted worlds, and even across worlds, provided that:*

- *The Non-Trusted world cannot access the key directly.*
- *The Trusted world can control the use of the key through a policy.*

An example key usage policy would cover at least the following:

- The world the input data is permitted to be read from.
- The world the output data is permitted to be written to.
- Permitted operations.

In the Assisted architecture, the Source key characteristic is extended to include Trusted hardware, when the key is derived or generated solely by hardware.

R140_TBSA_KEY *A Trusted hardware key must not be directly accessible by any software.*

A Trusted hardware key is used for Trusted world cryptographic operations, but its usage in a Non-Trusted world must be subject to a policy.

R150_TBSA_KEY *The Trusted world must be able to enforce a usage policy for any Trusted hardware key that can be used for Non-Trusted world cryptographic operations.*

5.4.5 Root keys

A TBSA-M-compliant SoC must provide authentication and encryption services through the use of embedded cryptographic keys. The exact number of embedded keys and their type depends on the target requirements, and is not specified in this document.

However, a TBSA-compliant device must embed at least two root keys, one for confidentiality and one for authentication. Other keys can be derived from these keys:

- A *Hardware Unique Key* (HUK), which provides the RoT for confidentiality.
- A root authentication key, which is the public key half of an asymmetric key pair. This key might belong to an RSA or to an *elliptic curve cryptosystem* (ECC), and is referred to as the *RoT Public Key* (ROTPK).

Examples of other embedded root keys are:

- Endorsement keys: these asymmetric key pairs prove identity, and therefore trustworthiness, to the external world.
- Additional symmetric keys for firmware decryption and provisioning. These keys are either unique to the device, or are class keys that are common across a family of devices.

The use of ECC for asymmetric cryptography is often beneficial, because its smaller key sizes lessens storage and transmission requirements. For example, depending on the algorithm and parameters chosen, the RSA algorithm of key size 3072 bits gives comparable security to an ECC algorithm of key size in the range 256-383 bits [19].

System architects should also review the comparative performance of RSA and ECC implementations in terms of throughput for each relevant key use case.

R160_TBSA_KEY *A TBSA-M device must either entirely embed an ROTPK, or the information that is needed to securely identify it.*

When no longer in use, Arm recommends hiding the ROTPK using a non-reversible mechanism, for example a sticky register bit that is activated by the boot software.

An ROTPK key size appropriate for a security strength of 128-bits as recommended by NIST [19] must be used. The reason for this is to support the longevity of the device beyond the year 2030.

R180_TBSA_KEY *An elliptic-curve-based ROTPK must be at least 256 bits in size.*

R190_TBSA_KEY *An RSA-based ROTPK must be at least 3072 bits in size.*

If an asymmetric cryptosystem is implemented, the following approach is permitted to reduce the ROTPK storage footprint.

Instead of the key itself, a cryptographic hash of the key is stored in on-chip non-volatile storage. The public key can then be stored in external non-volatile memory. When required, the key must be retrieved from external memory before it is used, and successfully compared with the stored hash by Trusted hardware or software. This approach is known as hash locking. Because this approach is not susceptible to a second pre-image attack, only half of the digest bits from an approved hash algorithm need to be stored. For example, a common truncation mode is for the leftmost 128 bits from a SHA-256 digest to be used [19].

R200_TBSA_KEY *If a cryptographic hash of the ROTPK is stored in on chip non-volatile memory, rather than the key itself, it must be immutable.*

R220_TBSA_KEY *A TBSA-M device must embed an HUK in confidential-lockable-bulk fuses.*

R230_TBSA_KEY *The HUK must have at least 128 bits of entropy.*

R240_TBSA_KEY *The HUK must only be accessible by Trusted code or Trusted hardware that act on behalf of Trusted code.*

The storage size and accessibility options for root keys are summarized in Table 3.

Table 3: Root key summary

Name	On-Chip Data Size	Off-Chip Data Size	Access to On-Chip Data
ROTPK – RSA	3072 bits (Key)	0 bits	During boot ROM execution. Only
	128 bits (Digest)	3072 bits (Key)	During boot ROM execution only.
ROTPK – ECC	256 bits (Key)	0 bits	During boot ROM execution only.
HUK	128 bits (key)	0 bits	Trusted code/Trusted hardware only.

5.5 Trusted boot

5.5.1 Overview

The secure configuration of a TBSA-M device depends on Trusted software that forms part of a chain of trust, beginning with the Trusted boot of the SoC. TBSA-M security is not possible without a Trusted boot mechanism.

Trusted boot is based on an immutable Trusted boot image. It is the first code to run on the host processor, and is responsible for verifying and launching the next stage boot. The Trusted boot image must be fixed within the SoC before it can be deployed and is stored in an embedded ROM. This ROM is referred to as the Boot ROM. Boot ROMs are typically implemented as either mask ROM, or by embedded flash with hardware support to ensure that, once programmed, the Boot ROM cannot be subsequently altered. The Boot ROM contains both the boot vectors for all processors, and the Trusted boot image.

R010_TBSA_BOOT *A TBSA-M device must embed a Boot ROM with the initial code that is needed to perform a Trusted system boot.*

The immutability of the initial Boot ROM is critical to the security of the device. If write-disabled embedded flash is used, then consider use of fuses or Trusted Subsystems or special write-once registers to disable writes to the boot partition. The robustness of the Boot ROM implementation depends on how strongly the design can demonstrate that vulnerabilities in code running on the host cannot lead to mutability of the Boot ROM. When the underlying storage technology is mutable with, for example, embedded flash, Arm recommends using OTP fuses or a controlling Trusted subsystem (e.g. Arm CryptoCell) to provide the necessary robustness guarantees.

Typically, the boot loader is divided into several stages. The first stage is the Boot ROM. Later stages might be loaded from non-volatile storage into Secure RAM and executed there, or executed directly from eFlash. In this document, the second stage boot loader is referred to as Trusted Boot Firmware. The firmware that is loaded by the Trusted Boot Firmware is called Trusted Runtime Firmware.

5.5.2 Boot types

There are two classes of boot: a cold boot and a warm boot. A cold boot is not based on a previous system state. Normally, a cold boot occurs when the platform is powered up, and a hard-reset signal is generated by a power-up reset circuit. However, depending on the design, a hard reset option that triggers a cold boot may also be available to the user in case of a software lock-up.

A warm boot can deploy one of the following methods to reuse the stored system state, on resuming from sleep, for example:

- The Boot ROM can use a platform-specific mechanism that is designed into the Boot ROM to distinguish between a warm boot and a cold boot.
- The SoC can use platform-specific registers to support an alternate reset vector for a warm boot.

R020_TBSA_BOOT *If the device supports warm boot, a flag or register that survives warm boot must exist to distinguish between warm and cold boots. This register or flag must be programmable only by the Trusted world and must be reset after a cold boot.*

Typically, any storage that is required to support these mechanisms is implemented within a power domain that is always powered up.

5.5.3 Boot configuration

If the SoC implements multiple processor cores, then the designated boot processor core is called the primary processor core. After the de-assertion of a reset, the primary processor core executes the Boot ROM code, and the remaining cores are held in reset, or a safe platform-specific state, until the primary processor core initializes and boots them.

R030_TBSA_BOOT *On a cold boot, the primary processor must boot from the Boot ROM. It must not be possible to boot from any other storage unless Trusted kernel debug is enabled. For more information about Trusted kernel debug, see section 5.10.*

In one possible implementation, the platform power controller holds all secondary processors in a reset state, while the primary processor executes the Boot ROM until it requests the secondary processors to be released. In an alternative implementation, all processors execute from the generic boot vector in the Boot ROM after a cold boot. However, the Boot ROM identifies the primary processor and permits it to boot using the Trusted boot image, while the secondary processors are made inactive.

The Armv8-M architecture, when implemented with the Security Extension, will boot into Secure Thread mode for both warm and cold boot.

The Trusted Boot ROM contains sensitive code that verifies and decrypts the next stage of the boot. If an attacker read and disassembled the ROM image, they would gain valuable information that could be used to target an attack that circumvents the verification mechanism. For example, timing information is used to target a fault injection attack.

Arm recommends making the Trusted boot image within the Boot ROM accessible only during boot. Device designers must consider implementing a non-reversible mechanism which prevents access by, for example, hiding the Trusted boot image using a sticky register bit that is activated by the boot software. This recommendation excludes the initial code that supports warm boot.

Arm recommends that the Trusted Boot Firmware image should be stored encrypted using an approved algorithm when the image is stored in external NVM. See Chapter 7 for more information. This practice deters the acquisition of the image by an attacker to inspect for vulnerabilities.

The Trusted Boot Firmware image is encrypted either by using a HUK-derived key or by using a common static key. Using a HUK-derived key requires a unique image for each device. Using a common static key enables the

same image to be used across a set of devices. Arm recommends authenticating externally held Firmware using an approved algorithm.

Arm also recommends protecting the key that decrypts Trusted boot firmware from being accessed or re-derived after boot, in order to mitigate the threat of attacks revealing the plaintext of Trusted Boot Firmware image. The key and its source material must be either inaccessible or accessible only by the Trusted world.

The Trusted Boot Firmware code verifies and, if successful, launches the next stage boot, which is Trusted Runtime Firmware. For TBSA-M devices, this firmware might be held in either on-chip or off-chip non-volatile memory. When the Trusted Runtime firmware is held in on-chip NVM, then this may be executed in place following verification. However, if this firmware is held in off-chip NVM, its launch is a non-trivial operation. This is because portions of the image must be copied to RAM before authentication. When loaded into RAM, the image is optionally decrypted before it is verified. If verification is successful, the image is executed. Verification is based on public key cryptography, which uses a digital signature scheme. Arm recommends using different keys for decryption and for Trusted Boot Firmware.

A boot status register is implemented to indicate the boot state of each Trusted processor. For example, the boot status register enables the application processor to check whether other Trusted processors booted up correctly. The register must be made available to secure debug so that the debugging agent may act accordingly. The register can also be used as a general boot status register.

R090_TBSA_BOOT *If a boot status register is implemented, then it must be accessible only by the Trusted world.*

5.5.4 Stored configuration

Some aspects of the secure boot behavior, which are governed by the Trusted ROM, might depend on stored configuration information. For example, in the case of a warm boot, configuration information might be stored in Trusted registers that are immutable between secure boot executions. This is implemented using a sticky register bit to prevent access to the data. The sticky bit is set by the secure boot code when the necessary operations of a cold or warm boot have been performed. The stick bit is then reset by triggering a warm or a cold boot.

In the case of a cold boot, the Trusted ROM behavior might be entirely fixed in the implementation. However, it can also be influenced by additional configuration information stored in fuses.

Fuse configuration information is used for the following purposes:

- Selection of the boot device.
- Storage of the root public authentication key.
- Storage of a root key for boot image decryption.
- Storage of other boot specific parameters.

5.5.5 Secure lockdown

For certain applications, Arm recommends that the Boot ROM fixes certain registers soon after reset, so that they cannot be subsequently changed until the next reset. This is because boot ROM firmware, in practice, is smaller, and more tightly audited, than software that follows. Therefore, lockdown may be a countermeasure against certain types of vulnerabilities in software that is executed post-boot. Examples of state which the Boot ROM might want to fix are:

- Secure Vector Table Offset Register (VTOR_S).
- Secure MPU.
- Non-secure MPU.
- Security Attribution Unit.

Armv8-M Cortex implementations present signals at the processor boundary. These might be wired into a write-once register to facilitate lockdown.

5.5.6 Assisted architecture

At each step in the boot chain, each stage must verify the next. If the Trusted Boot Firmware is encrypted, a decryption step is also required. Verification of an image is based on a cryptographic hash function and asymmetric cryptography. Decryption of an image is based on symmetric cryptography. As the underlying cryptographic algorithms are CPU-intensive, the Assisted architecture implements hardware acceleration. See Section 4.8.

In an Assisted architecture, the symmetric key that decrypts the Trusted Boot Firmware is used only by the accelerator peripheral, and is not visible to software.

R100_TBSA_BOOT *In an Assisted architecture, the key to decrypt the Trusted Boot Firmware image must be visible only to the acceleration peripheral.*

5.6 Trusted timers

5.6.1 Trusted clock source

Trusted clock sources are required to implement Trusted watchdog timers and Trusted time. Any clock source that the Trusted world depends on is classified as a Trusted clock source, and can only be configured from the Trusted world.

In addition to this, a Trusted clock source must be robust against tampering that happens outside the control of the associated Trusted manager. Two recommended protection strategies are possible:

- Internal clock source: The clock source is an integrated autonomous oscillator within the die and cannot be easily altered or stopped without deploying invasive techniques.
- External clock source: The clock source is an external XTAL or clock module and connects to the main SoC through an I/O pin. In this case, an attacker can easily stop the clock or alter its frequency. If this is the case, then the main SoC must implement monitoring hardware that can detect when the clock frequency is outside its acceptable range.

Arm recommends that, if clock monitoring hardware is implemented, the hardware must expose a status register indicating whether the associated clock source is compromised. This register must be readable only from the Trusted world, in order to prevent leakage or modification of information that may assist an attacker.

To signal a clock frequency violation, it might be useful to add a Trusted interrupt to any Trusted clock monitoring hardware.

5.6.2 General Trusted timer

Trusted timers are required to provide time-based triggers to Trusted world services. A TBSA-M system must support one or more Trusted timers.

R030_TBSA_TIME *At least one Trusted timer must exist.*

R040_TBSA_TIME *A Trusted timer must only be modified by a Trusted access. Examples of modifications are the timer being refreshed, suspended, or reset.*

R050_TBSA_TIME *The clock source that drives a Trusted timer must be a Trusted clock source.*

5.6.3 Watchdog

A TBSA-M system must support one or more Trusted watchdog timers.

Trusted watchdog timers are required to protect against denial of service, which could occur when, for example secure services depend on the RTOS scheduler. In such cases, if the Trusted world is not entered before a pre-defined time limit, a reset is issued and the SoC is restarted.

A Trusted watchdog timer might need to signal an interrupt in advance of the reset, permitting some state save before a reboot. The watchdog timer must use a mechanism that can indicate to boot software whether the expiry of the watchdog timer is the reason for the reboot.

R060_TBSA_TIME *At least one Trusted watchdog timer must exist.*

R070_TBSA_TIME *After a system reset, a Trusted watchdog timer must be started before execution of the immutable boot code transfers control to the next firmware stage.*

R080_TBSA_TIME *A Trusted watchdog timer must only be modified by a Trusted access. Examples of modifications are the timer being refreshed, suspended, or reset.*[*]

Following a reset, a Trusted system timer must be initiated prior to execution of the handover from the immutable boot ROM to the first part of the boot process which executes from mutable code. If mitigation against manipulation of embedded flash or RAM at reset is part of the device threat model, then Arm recommends initiating the watchdog by hardware before any code execution starts.

[*]Note For reasons of practicality, compliance with R080_TBSA_TIME is preferred, but not mandatory, during debug. Allowing invasive (halting) debug of the Non-Trusted world in the deployed state may also affect the Trusted world timers in many implementations. System designers should consider whether allowing invasive debug of the Non-Trusted world creates exploitable vulnerabilities in the operation of Trusted timers, including Trusted Watchdog timers, and should take measures to mitigate this possibility.

Arm recommends using a clock speed of at least 1 Hz when the device is not in a power saving cycle.

R090_TBSA_TIME *Before needing a refresh, a Trusted watchdog timer must be capable of running for a time period that is long enough for the Non-Trusted re-flashing of early boot loader code.*

R100_TBSA_TIME *A Trusted watchdog timer must be able to trigger a reset of the SoC, after a pre-defined period of time. This value is fixed in hardware or programmed by a Trusted access.*

R110_TBSA_TIME *A Trusted watchdog timer must implement a flag that indicates the occurrence of a timeout event that causes a warm reset, to allow post-reset software to distinguish this from a power-up cold boot.*

R120_TBSA_TIME *The clock source driving a Trusted watchdog timer must be a Trusted clock source.*

5.6.4 Trusted time

Many Trusted services, for example feature enablement, rely on the availability of Trusted time. Typically, Trusted time is implemented using an on-chip real-time counter that is synchronized securely with a remote time server.

An ideal implementation of a *Trusted real-time clock* (TRTC) consists of a continuously powered counter driven by a continuous and accurate clock source, with Trusted time programmable only from the Trusted world. However, devices that contain a removable battery must deal with power outages.

A suitable solution for dealing with power outages is realized by implementing a counter together with a validity mechanism, for example a status flag, that indicates whether a valid time has been loaded.

A TBSA-M system deploying this solution implements Trusted time using a TRTC that consists of a Trusted hardware timer. The Trusted hardware timer is associated with a mechanism indicating whether the current

time is valid, and receives a Trusted clock source. The mechanism indicates when the Trusted timer has been updated by a Trusted service, and indicates when power is removed from the timer. Arm recommends that the Trusted timer and its validity mechanism reside in a power domain that remains powered up as much as possible.

When Trusted time is lost because of a power outage, the response depends on the target specifications. For example, it might be acceptable to restrict specific Trusted services until the TRTC has been updated by the appropriate Trusted service.

R130_TBSA_TIME *A TRTC must be configured only by a Trusted world access.*

R140_TBSA_TIME *All components of a TRTC must be implemented within the same power domain.*

R150_TBSA_TIME *On initial power up, and following any other outage of power to the TRTC, a validity mechanism must indicate that the TRTC is not Trusted.*

R160_TBSA_TIME *The TRTC must be driven by a Trusted clock source.*

5.7 Version counters

A compliant TBSA-M system must implement a core set of Trusted non-volatile counters, which are required for version control of firmware and Trusted data held in external storage. In order to prevent replay attacks, it is important that these counters cannot be rolled back.

The following counters are mandatory:

- A Trusted firmware version counter.
- A Non-Trusted firmware version counter.

Ideally, an SoC implements version counters using on-chip MTP storage, for example embedded flash technology.

OTP storage, based on anti-fuse technology, is widely available and cost effective. A non-volatile counter is implemented by mapping each possible value that is greater than one onto a separate fuse bit. Each counter increment is achieved by programming a further bit. As one bit is required for each value, this is costly for large counters. For example, a 10-bit counter requires 1024 bits of storage. For this reason, practical limitations must be imposed on the maximum count values for fuse-based implementations.

The size requirement for a version counter depends on the target specification. For a TBSA-M system, the minimum requirements are:

R010_TBSA_COUNT *An on-chip non-volatile Trusted firmware version counter implementation must provide a counter range of at least 0 to 63.*

R020_TBSA_COUNT *An on-chip non-volatile Non-Trusted firmware version counter implementation must provide a counter range of at least 0 to 255.*

All on-chip non-volatile version counters must also meet the following requirements:

R030_TBSA_COUNT *It must only be possible to increment a version counter through a Trusted access.*

R040_TBSA_COUNT *It must only be possible to increment a version counter. It must not be possible to decrement it.*

R050_TBSA_COUNT *When a version counter reaches its maximum value, it must not roll over, and no further changes must be possible.*

R060_TBSA_COUNT *A version counter must be non-volatile, and the stored value must survive a power down period up to the lifetime of the device.*

Furthermore, Trusted version counters might also be required to support version control of other platform 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.

5.8 Entropy source

Many cryptographic protocols depend on challenge response mechanisms that utilize truly random numbers. This means that an embedded *true random number generator* (TRNG) is an important element of a TBSA-M system.

When platform requirements demand a TRNG, there is normally an associated requirement specifying the quality of the source. More commonly, a set of tests must be passed by a compliant source.

The quality of a random source is normally described in terms of entropy. In information theory, entropy is measured on a logarithmic scale in the range [0,1]. For a given string of bits provided by a TRNG, the maximum entropy of 1 is achieved if all bit combinations are equally probable.

A formal treatment of entropy is found in [13].

A hardware realization of a TRNG consists of two main components: an entropy source and a digital post processing block, as shown in Figure 13:

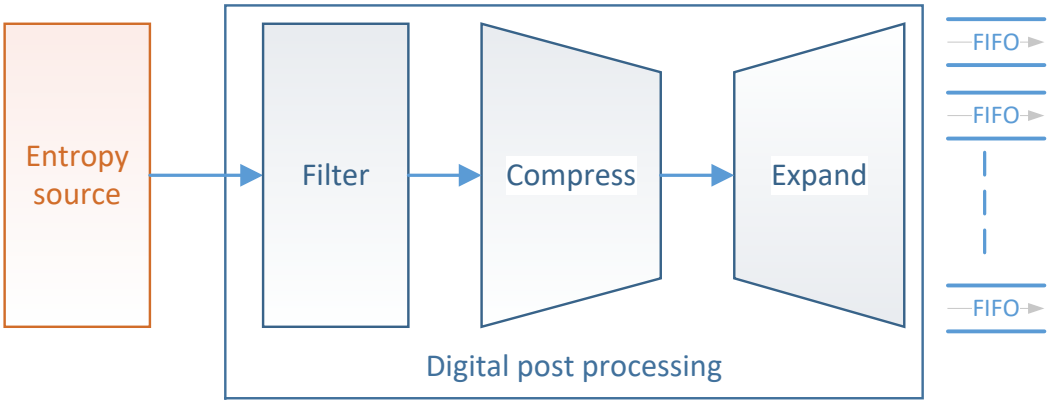

Figure 13: Entropy source top level

The entropy (noise) source incorporates the non-deterministic, entropy-providing circuitry that provides the uncertainty associated with the digital output by the entropy source.

The digital post-processing block collects entropy from the analog source through sampling, to monitor the quality of the source, and to filter it appropriately, to ensure a high level of gathered entropy. For example, repeated periodic sequences are clearly predictable and must be rejected. This is important because fault injection techniques can induce predictable behavior into a TRNG and attack the protocols that use it.

For any entropy source design, the quality of the entropy is reduced as the sample rate increases. Any design has a maximum safe ceiling for the sample rate, and this sample rate might not be high enough to meet the overall system requirements.

Although it is possible to design a filtering scheme removing common and predictable patterns that can occur in an entropy source, other, more complex patterns might persist, which degrade the available entropy. The extent of any such degradation depends on the quality of the source, and in some cases additional digital processing might be required to compensate for it.

A common compensation technique utilizes a cryptographic hash function to compress a large bit string of lower entropy into a smaller bit string of higher entropy. However, this function comes at the expense of available bandwidth.

To counter this situation, the digital post processing stage expands the entropy source to provide a greater number of bits per second, by using the filtered or compressed source to seed a cryptographically strong pseudo random sequence generator with a very large period.

A definitive treatment of these steps is found in the *NIST Draft Special Publication 800-90b*.

R010_TBSA_ENTROPY *The entropy source must be an integrated hardware block.*

Although some or all of the digital post processing can be performed in software by a Trusted Service, Arm recommends a full hardware design.

It is not possible to construct a TRNG that yields exactly one bit of entropy per output bit, so it is permissible to provide output samples together with their assessed entropy in bits. For example, the TRNG might provide 32-bit samples that contain only 24 bits of entropy. If the assessed entropy of each sample is variable, the TRNG must provide an assessed entropy value with each sample, unless the assessed entropy is a fixed and known constant.

R020_TBSA_ENTROPY *The TRNG must produce samples of known entropy.*

There are many possible choices for measuring entropy. Following the guidance in NIST SP 800-90 [13], Arm recommends using a conservative measure called min-entropy. Min-entropy is used as a worst-case measure of the uncertainty associated with observations of X. If X has min-entropy m, then the probability of observing any particular value is no greater than 2-m.

A number of test suites exist to ensure the quality of a TRNG source. Arm recommends that the TRNG design passes the following test suits:

Table 4: Entropy test suites

Name	Details
NIST 800-22	A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, April 2010 **http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html**
DieHard	**https://en.wikipedia.org/wiki/Diehard_tests** **https://wayback.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/**
DieHarder	**http://www.phy.duke.edu/~rgb/General/dieharder.php**
ENT	**http://www.fourmilab.ch/random/**

R030_TBSA_ENTROPY *The TRNG must pass the NIST 800-22 test suite.*

Details of the NIST 800-22 test suite can be found at [14].

R040_TBSA_ENTROPY *On production parts, it must not be possible to monitor the analog entropy source using an external pin.*

5.9 Cryptographic acceleration

In the Assisted architecture, the hardware offers acceleration of some of the cryptographic operations to meet the performance requirements of the system. This permits hardware management of the cryptographic keys, which are the most valuable assets in the system. By managing the keys in hardware, the threat space is drastically reduced.

If large amounts of data must be processed, cryptographic algorithms are often accelerated, which makes symmetric and hashing algorithms the most commonly accelerated. Asymmetric algorithms are complex, so full accelerators are also complex and often large. A common trade-off is to accelerate only the most computing-intensive parts, for example big integer modulo arithmetic.

Figure 14 shows an example architecture for symmetric algorithm acceleration and an associated Key Store:

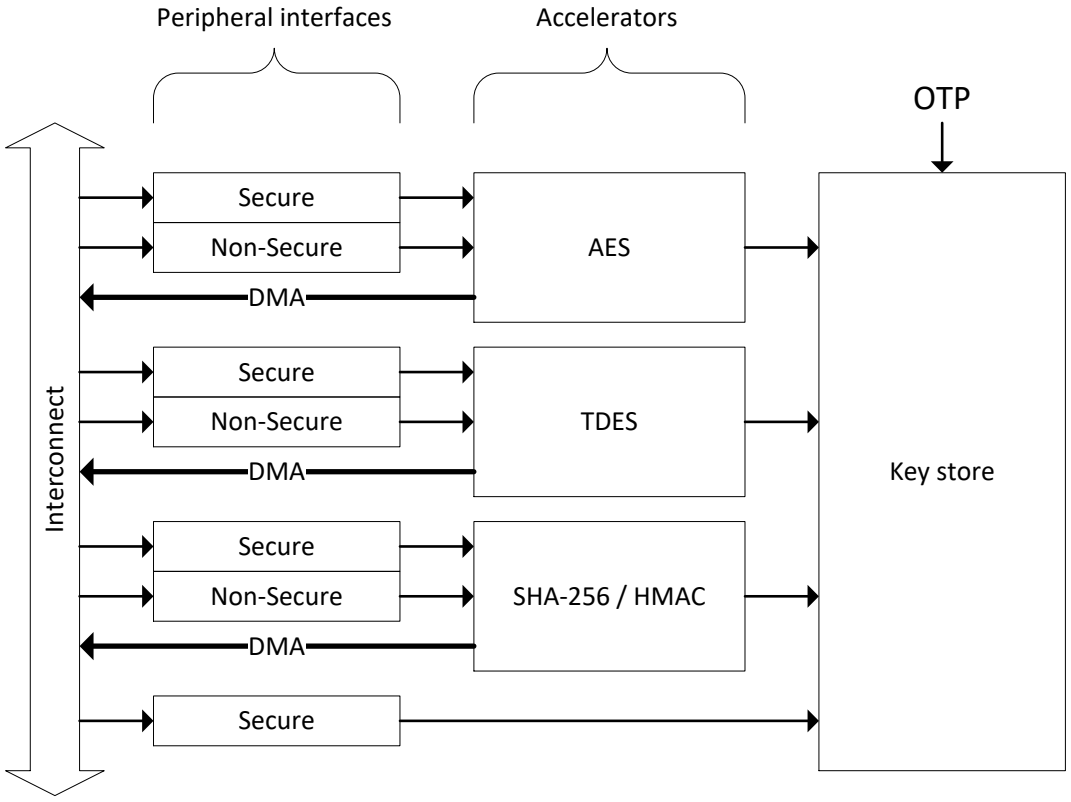

Figure 14: Example of a symmetric crypto acceleration architecture

The accelerators and the Key Store are peripherals within a TBSA-M SoC, and must meet the associated requirements.

The Key Store contains entries of keys and their associated metadata. The keys might have been injected through the secure peripheral interface, from Trusted software, or directly from OTP. The metadata associated with a key can include policy restrictions, by indicating which accelerator engines can access the key, exactly what operation is permitted, and which worlds the input and outputs must be in. By storing keys in a Key Store, the period of time that the keys are directly readable by software is significantly reduced.

The accelerators are used by both the Trusted and Non-Trusted worlds, and have both Secure and Non-secure interfaces. These interfaces permit software to request cryptographic operations on data that is stored in memory, and either supply a key directly, or index a key and its metadata in the Key Store. When programmed, the accelerator reads data using its DMA interface, performs the operation, and writes the resultant data.

More advanced versions of this acceleration architecture might support key derivation functions, for instance if the resultant data from a decryption is not written to memory using DMA, but is instead placed into the Key Store.

5.10 Debug

As SoCs have become increasingly complex, the mechanisms for debugging the hardware and software have also increased in complexity. The fundamental principles of debugging, which require access to the system state and system information, conflict with the principles of security, which require the restriction of access to assets. This section brings together the high-level security requirements for all debug mechanisms in the SoC.

Armv8-M supports the following debug modes:

- Self-hosted debug: The processor itself hosts a debugger. Developer software and a debug kernel run on the same processor. For more information, see the *Armv8-M Architecture Reference Manual*, Section B11.
- External debug: The debugger is external to the processor. The debugging might be either on-chip, for example in a second processor, or off-chip, for example a JTAG debugger. External debug is particularly useful for:
    - Hardware bring-up. That is, debugging during development when a system is first powered up and not all of the software functionality is available.
    - Processors that are deeply embedded inside systems.

For more information, see the *Armv8-M Architecture Reference Manual*.

The Armv8-M architecture also includes definitions for invasive and non-invasive debug. From a security perspective, there is no need to distinguish between these, because non-invasive debug can leak any assets accessed by that processor.

5.10.1 Protection mechanisms

Debug mechanisms give an external entity access to system assets, so protection mechanisms must be in place to ensure that the external entity is permitted access to those assets. These are called *Debug Protection Mechanisms* (DPMs) and may implemented be solely in hardware or in a mixture of hardware and firmware.

R010_TBSA_DEBUG *All debug functionality must be protected by a DPM so that only an authorized external entity can access the debug functionality. There might be scenarios in which all external entities can access the debug functionality.*

In devices which share JTAG/SWD with functional I/O, this requirement must be interpreted as prohibiting Non-secure software observing or influencing secure debug activities.

R020_TBSA_DEBUG *A DPM must be implemented either solely in hardware or together with software running in the Trusted world.*

Non-Trusted and Trusted system assets are partitioned according to the worlds in which they are accessible.

R030_TBSA_DEBUG *There must be a DPM to permit access to all assets (Trusted).*

R040_TBSA_DEBUG *There must be a DPM to permit access to all Non-Trusted world assets (Non-Trusted). This mechanism must not permit access to Trusted world assets.*

Arm recommends making DPMs lifecycle-aware. The requirements described in this section apply to the deployed lifecycle state. See Section 6.

5.10.2 Debug Protection Mechanism overlap

The DPM requirements lead to an overlap of the worlds or spaces that each DPM unlocks, as shown in **Figure 15** and **Table 5** below.

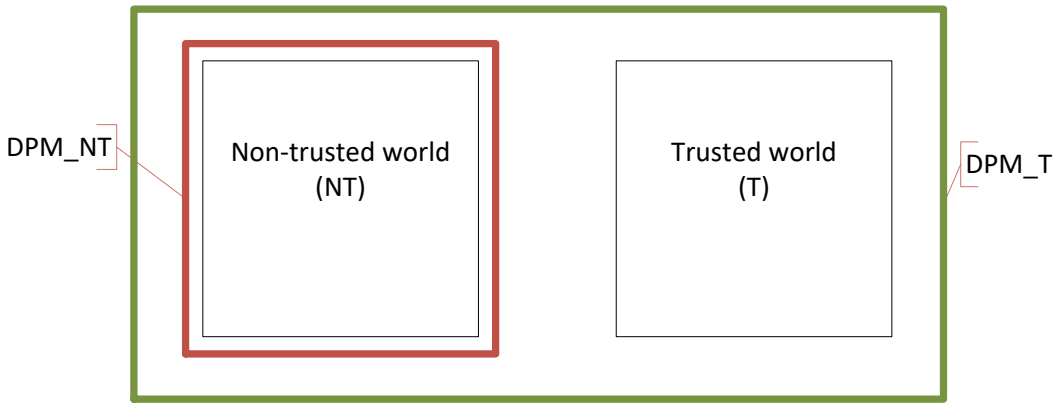

Figure 15: DPM overlap

Table 5: DPM overlap

Master DPM	Unlock opens
DPM_T	Trusted world spaces Non-Trusted world spaces
DPM_NT	Non-Trusted world spaces

For some applications, it may be desirable to implement further DPMs to give a finer grain control of debug access for Armv8-M. DPMs can control according to whether access is privileged or not, and whether it accesses Trusted or Non-Trusted spaces.

5.10.3 Debug Protection Mechanism states

Each DPM must have states that reflect access to the debug mechanisms. These states must be controlled by fuses and the unlock mechanism. This is captured in the following requirements:

R050_TBSA_DEBUG *All DPMs must implement the following fuse-controlled states:*

- *Default: Debug is permitted.*
- *Closed: Only an unlock operation is permitted, to transition to Open.*

These must be determined by a Boolean value (dpm_enable) that is stored in a public-open-bitwise fuse or derived from the Device Lifecycle state stored in fuses, see Figure 16.

R090_TBSA_DEBUG *The DPM controlling Trusted world functionality must also have another fuse-controlled state:*

- *Locked – The unlock operation is disabled (no state transition is possible).*

These must be determined by a Boolean value (dpm_lock) that is stored in a Public-Open-Bitwise fuse or derived from the Device Lifecycle state stored in fuses, see Figure 16*.*

R120_TBSA_DEBUG *All DPMs must have the following state:*

- *Open – Debug is permitted.*

The Open state can only be entered from the Closed state after a successful unlock operation.

Note: The fuses and unlock mechanisms for each DPM do not have to be unique. For example, one fuse can be used as the dpm_enable for both DPMs, and one unlock mechanism can unlock both DPMs.

Table 6 shows the DPM states and their allowable transitions.

Table 6: DPM states

DPM state	Debug access	Transition(s)	Notes
Default	Yes	None except through Reset	
Closed	No	Open – after a successful unlock operation	
Open	Yes	None except through Reset	
Locked	No	None	Only required for Trusted world Optional in Non-Trusted world

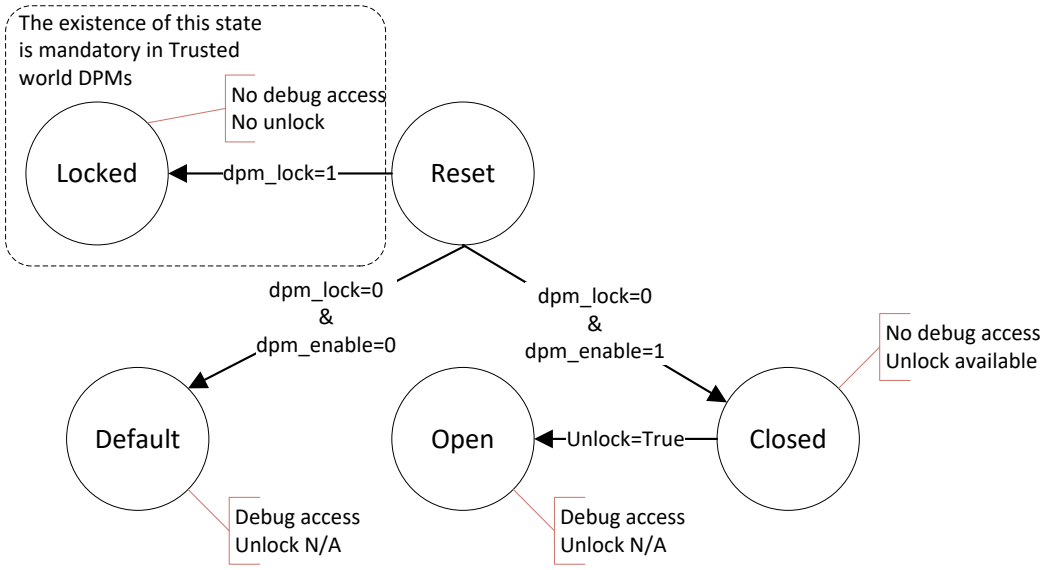

Figure 16: DPM states

Note: The power domain and reset of the DPM state must be carefully considered to ensure that all operations of the SoC are debugged. For example, debugging of the Trusted Boot ROM during cold and warm boots might require the state to be stored in a domain that is permanently powered up, and has an independent reset.

The DPMs are required to protect the system assets, which necessitates the following requirement:

R150_TBSA_DEBUG *The Trusted world DPM must be enabled, using the respective dpm_enable fuses, or locked, using the respective dpm_lock fuses, before any Trusted world assets are provisioned to the system.*

5.10.4 Unlock operations

To perform the state transition from Closed to Open, the debug protection mechanism must perform an unlock operation to ensure that the external entity has access to a token authorizing access to the associated assets. The token might be a simple device-unique password, a cryptographically signed certificate, or a response to a challenge. Which form to use often depends on the trade-off between complexity on the device and complexity on an external server. For example, it is more complicated to implement signature checking on a device than to compare passwords, but managing a database of unique passwords is more complicated than one or two private keys on a server.

To prevent the leak of an unlock token that affects multiple devices, Arm recommends unique unlock tokens for each device. Device manufacturers should consider the point in the supply chain that is appropriate to provision unlock tokens, by taking the security model of the device into account.

To ensure that the external entity knows which unlock token to use, Arm recommends either that the device stores a unique ID in Public-Lockable fuses, or that the Unique ID is derivable from a value in public-lockable fuses.

Arm also recommends that devices deploy other additional protection for the unlock token, depending on the type of token, and the threat model. Protection mechanisms include the use of non-volatile unlock failure counters managed by Trusted firmware, and a nonce to protect against replay attacks.

Unlock token: password

Password-based unlock operations are implemented as a simple comparison. However, Arm recommends not storing a copy of the password on the device itself. Instead, Arm recommends storing a cryptographically-strong hash of the password that is created using a *one-way function* (OWF). When the password token is injected using an interface from the external entity, it is passed through the same OWF and compared with the stored hash.

Because the comparison is simple, it must be protected from brute force attacks, by making the password sufficiently large:

R200_TBSA_DEBUG *A password unlock token must be at least 128bits in length.*

For password-based unlock tokens in particular, Arm recommends that devices deploy other additional countermeasures, for example non-volatile unlock failure counters managed by Trusted firmware.

This recommendation ensures that different external entities are given different tokens for a device, depending on their ownership of assets in the system:

R210_TBSA_DEBUG *Each debug protection mechanism must use a unique password unlock token.*

Unlock token: certificate-based

Certificate-based unlock operations require the injection of a certificate that has been cryptographically signed by a private key. The certificate should be paired with the device. Arm strongly recommends that the unique ID of the device be included in a certificate unlock token. There are circumstances in which a vendor wants to distribute class-wide certificates to field engineers or other agents who require debug access. In these circumstances, the class should be as small as practical, in accordance with the principle of least privilege, and consistent with the threat modelling of the device. Certificates should contain fields, authenticated by digital signature, which indicate the permissions granted by the signing authority.

The debug protection mechanism must check the signature of the certificate:

R230_TBSA_DEBUG *An unlock operation using a certificate unlock token must use an approved asymmetric algorithm to check the certificate signature.*

R240_TBSA_DEBUG *An unlock operation using a certificate unlock token must have access to an asymmetric public key stored on the device. The asymmetric public key that is used to authenticate the certificate unlock token must be immutably stored on the device, or have been loaded as a certificate during secure boot and authenticated by a chain of certificates that begins with the ROTPK.*

R250_TBSA_DEBUG *A certificate unlock token must indicate which DPM(s) it is able to unlock using an authenticated field.*

R260_TBSA_DEBUG *A loadable public key for certificate unlock token authentication must include an authenticated field indicating which DPM(s) it is authorized to unlock.*

R270_TBSA_DEBUG *A certificate unlock token must only unlock a DPM that its authenticated field is authorized to unlock.*

In the simplest case, the chain of certificates is of length one, and the ROTPK and the public key used to authenticate the unlock token are one and the same.

Unlock token: challenge-response

Challenge-response unlock operations use a cryptographic protocol to demonstrate that a debugger knows a shared secret without revealing the secret to an eavesdropper. Using this method, the debugger first obtains a

random challenge from the device. The debugger then computes the response by applying a cryptographic hash function to the device challenge combined with the secret. The response is then authenticated by the device.

The shared secret follows the size and uniqueness requirements for unlock token passwords described above.

5.10.5 Other debug functionality

Complex SoCs often include extra debug functionality beyond the main processor. Examples of this are initiators on the interconnect, which are controlled directly from an external debug interface, and system trace modules. Care must be taken to ensure that any extra debug functionality is controlled by the correct DPM. They must be evaluated based on their access to assets that belong to each world, and assigned the appropriate DPM.

5.10.6 Arm debug implementation

The Arm processor and CoreSight IPs include an Authentication Interface with the signals shown in Table 7:

Table 7: Arm authentication interface

Signal	Name	Action
DBGEN	Debug Enable	Enables invasive and non-invasive debug of Non-secure state. Debug components are disabled but accessible.
NIDEN	Non-invasive Debug Enable	Enables non-invasive debug of Non-secure state.
SPIDEN	Secure Privileged Invasive Debug Enable	When asserted with DBGEN enables invasive & non-invasive debug of Secure state.
SPNIDEN	Secure Privileged Non-Invasive Debug Enable	When asserted with NIDEN, enables non-invasive debug of Secure state.

These signals can be mapped to the debug protection mechanisms, as shown in **Table 8**.

Table 8: DPM mapping to authentication interface

World	Debug Functionality	Equation
Non-secure	Non-secure Invasive debug	DBGEN
	Non-secure Non-invasive debug	DBGEN | NIDEN
Secure	Secure Invasive debug	DBGEN & SPIDEN
	Secure Non-invasive debug	(SPIDEN | SPNIDEN) & (DBGEN | NIDEN)

5.10.7 Basic architecture

In the TBSA-M Basic architecture, DPMs might be implemented in software, including the unlocking of any external debug interfaces. There are two commonly used implementations:

- Space is reserved in the non-volatile memory map for the unlock token, and the unlock operation is performed by the secure boot process.
- The external debug interface receives an unlock token, and requests processing by the Trusted world.

In both cases, software must read the relevant fuses to understand the state of the DPM, and have target registers that unlock the relevant debug features of the device.

R280_TBSA_DEBUG *The device must implement registers that, when written to by software, unlock the associated hardware debug features. Access to the secure DPM registers must be restricted to privileged Trusted world software.*

5.10.8 Assisted architecture

In the Assisted architecture, both the Trusted and Non-Trusted DPMs are implemented in discrete hardware connected to the external debug interface. The unlock tokens are injected via the external debug interface. The tokens are then verified by the hardware that asserts the required signals to the rest of the device, or by firmware the boot ROM.

R290_TBSA_DEBUG *The DPM_T and DPM_NT must be implemented solely in hardware or together with firmware in immutable boot ROM.*

5.10.9 Unprivileged Debug Extension

Armv8.1-M provides a debug feature called Unprivileged Debug Extension (UDE). UDE allows system designers to restrict debug visibility to specific unprivileged software components. In the Secure execution state UDE allows debugging of specific secure partitions without giving any visibility of a debugger to other secure partitions. In the Non-secure state, with operating system support, UDE is used to debug unprivileged software components in an environment in which those components are mutually distrustful. The device lifecycle may require that all debug, including UDE, is prohibited when the device is deployed. In this case, Arm recommends that the SoC prevents a debugger authenticating itself to the CPU by intercepting signals between the debug interface and the CPU using a logic block which will drive the signals appropriately. Control of the interception block must be placed in the Trusted world. Among other things, this prevents using UDE mode in the non-secure state to invoke halting debug. Depending on the threat model of the device, using halting debug may support an attack.

5.11 External interface peripherals

TBSA-M based SoCs contain many of the functions of the final consumer device, but they are often required to talk to other electronic peripherals in order to receive and transmit data. Examples of these *External Interface Peripherals* (EIPs) include sensors, actuators and communication devices, for example Wi-Fi or Bluetooth Low energy modules. Some interfaces are simply connections through SPI or UART, whereas others can embed the controllers within the SoC itself.

Because these interfaces often receive Trusted user data, thought must be given to assets that are transferred across these interfaces. The following questions can aid this thought process:

- Which on-chip world do the assets belong to?
- Are the assets entering or leaving the device?
- Are the assets encrypted or not?
- Are the assets authenticated?

- If the assets are encrypted or authenticated, how was the key exchanged?
- What is the impact if the assets are modified?
- Can commands be received from an external device?

Often the easiest approach is to let the Non-Trusted world manage the interface, and to let the Trusted world supply the data to be transferred. This is acceptable when the Non-Trusted world is no more of a security risk than the external connection. For example, non-authenticated encrypted content is sent through the Non-Trusted world, because changing the encrypted content does not compromise the security of any assets. However, if the assets being transferred include user data and are not authenticated, the Non-Trusted world can perform a man-in-the-middle attack in the same way as an attacker with access to the external interface.

Therefore, if any secret values are not encrypted, the Non-Trusted world must not be able to access them and the external interface must be correspondingly protected.

R010_TBSA_EIP *If an EIP is used to send or receive clear or unauthenticated Trusted world assets, it is implementing a Trusted operation and must meet the requirements of a Trusted peripheral.*

R020_TBSA_EIP *When an EIP can receive commands from an external device, for example PCIe, then the system must enforce a policy to check that those commands do not breach the security of the TBSA-M device.*

The requirement R020_TBSA_EIP does not only apply to the commands that can affect the Trusted world. Unrestricted access to the Non-Trusted world by an external device is still a security risk.

R040_TBSA_EIP *Any sensitive user data that is stored must be stored in Secure storage.*

Threat analysis of biometric sensors used in particular applications might indicate that they must be under the control of the Trusted world. For example, if a camera is used in several use cases and one of these cases is for UV retinal imaging, the UV LED activation should be under control of the Trusted world.

R050_TBSA_EIP *When a sensor has modes that allow it to be used for the acquisition of assets in both the Trusted world and the Non-Trusted world, activating features for Trusted world sensing must be under the control of the Trusted world.*

5.12 DRAM protection

Some TBSA-M designs might also deploy external DRAM to store assets. In this case, Arm recommends following the advice given in reference [6].

6 Device lifecycle management [Normative]

Designs compliant with TBSA-M architecture must have a mechanism to manage the security lifecycle of the device. This mechanism governs the behavior of the device, in both hardware and firmware, in each stage of the lifecycle, protecting any security assets introduced into the device and reducing the risk of IP theft and reverse engineering.

Device lifecycle management is implemented either by a state machine in Trusted firmware which controls the introduction of OTP assets and fuses, or by an equivalent state machine implemented in hardware. Lifecycle state transitions should be atomic: it should not be possible for firmware outside the RoT to observe a partially completed transition. Also, state transitions should be robust against external events, for example power loss.

Details of how the device life manages the progression from device manufacture, provisioning of assets through deployment, including any changes of ownership through to discontinuation, are described in the PSA SM, and PSA Firmware Framework documents.

The PSA lifecycle tracks the state of the PSA RoT through its lifetime, from development and manufacturing, through use in the field, to debug and repair (shown as RMA in the figure below) states. Depending on its lifecycle state, the PSA RoT will have different security properties. For example:

- In early development and manufacture states, secrets and identities may not have been provisioned and debug ports may not yet have been locked down.
- In some debug and repair (RMA) states, secrets could potentially be compromised, or boot state and attestation might not be trustworthy anymore.

On a device compliant with TBSA-M, many objects may have their own lifecycles. For example:

- Some Trusted subsystems, for example Secure Element and Security Element style devices, may have their own local life cycles and provisioning processes.
- The application itself may have an application lifecycle, tracking whether the device has been enrolled into a Trusted service, or securely associated with a particular set of credentials.

The PSA lifecycle states indicate the RoT assets present in the device, and the functionality available or disabled in each state (e.g. debug, boot options, the writability of eFlash partitions). Figure 17 shows a generic example TBSA-M device lifecycle:

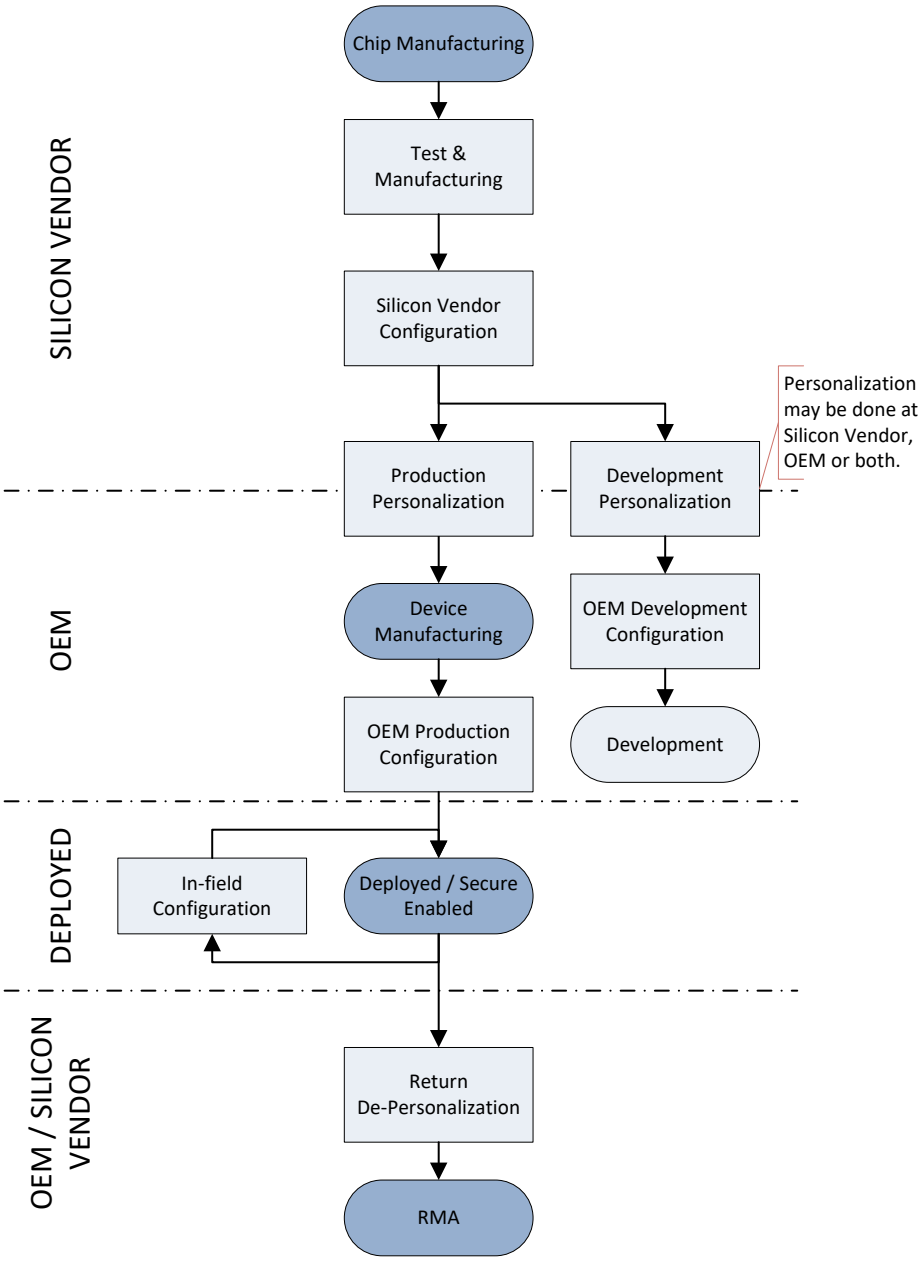

Figure 17: Illustration of a device lifecycle

This device lifecycle begins in the chip manufacturing state, which is completely open and contains only the assets that are fixed in hardware. At this point, the device must be fully testable, in order to permit checking for manufacturing defects. The device is then configured in multiple steps by the silicon vendor and the purchasing *Original Equipment Manufacturer* (OEM) through the programming of fuses.

Note: In the automotive industry, the OEM refers to the car maker, and Tier 1 supplier refers to device (ECU) manufacturer. Here the term OEM means either of these entities, depending on the particular application and supply chain.

Configuration includes personalization, which is the injection of cryptographic assets, for example unique keys. These assets are broadly grouped into two categories:

- Production assets: These are highly sensitive values that must be protected as soon as it is verified that they have been correctly programmed.

- Development assets: These are values known to the OEM or silicon vendor, or both, and are used during the development of the system.

Devices that are destined for sale to consumers are personalized with production assets by the silicon vendor and OEM. Such devices are configured to enable the security mechanisms required to protect those assets, and any other assets that are accessible to the device, for example in flash memory. When this configuration is complete, the device enters the Deployed state.

A device that is in the Development state has a subtly different configuration from the production parts, because features, for example debug can still be enabled. These parts are not intended to leave the OEM.

A device that is in the Deployed state only permits configuration operations that support the required use cases and has access to the security functions of the device. Its debugging and testing features are disabled, and secure boot is mandatory.

The *Return Merchandise Authorization* (RMA) lifecycle state is a terminal state used for devices that are returned to the manufacturer for failure analysis. When a device is put into the RMA state, it loses access to its secret keys and the ability to operate securely. The device should regain access to all debugging and testing capabilities, in order to perform reliability, warranty and liability investigations.

As a minimum, Arm recommends that a device compliant with TBSA-M provides a lifecycle control mechanism in which:

- The lifecycle state is held in, or derivable from, the state of fuses.
- All lifecycle state transitions are restricted to a designated set, as outlined in the example shown in Figure 17, in which there is at least:
    - A designated initial state from which all devices start.
    - A designated deployed state which mandates the use of the device's security features.
    - A designated terminal state (RMA) from which no further transitions are allowed.
- A transition into the RMA state should be authorized by the RoT owners, and should atomically zeroize, or otherwise put beyond use, all secret keys in the manner described by FIPS 140-2.
- Booting, debugging and scan access are governed by a secure lifecycle policy.

7 Cryptography requirements [Normative]

TBSA-M requires that cryptographic algorithms are aligned with standards which meet or exceed 128-bit security. This means that:

- Symmetric ciphers must be equivalent to at least AES with 128-bit keys.
- Asymmetric cryptography must be equivalent to at least:
    - ECDSA with P256.
    - RSA/DSA 3072.
- Hash functions must be equivalent to at least SHA-256.

Arm recommends drawing the core set of approved algorithms from the Commercial National Security Algorithm Suite, which supersedes NSA Suite B Cryptography [15]. Alternatively, designers should refer to the approved cryptographic algorithm lists that SOG-IS, IPA, and CC have published for the EU[24], Japan[25], and China[26].

For specific devices it may be appropriate to select algorithms according to the certification regime that the marketplace requires and according to the recommendations and requirements of the relevant governing security agency. Furthermore, the deployment time frame of the device, resource dimensioning, and performance of algorithm choices may play a part in this selection. These criteria should be a documented part of the device security review and are outside the scope of this document.

Depending on the threat model of the device, it may be necessary to anticipate migration to newer algorithms, potentially quantum-resistant ones, within the lifetime of the device. In addition, if protection for long-term data privacy is required, adopting a higher security level for symmetric cryptographic algorithms may be needed.

8 Related documents

A detailed description of Trusted Boot and Firmware Update requirements is found in [8]

A good treatment of how to design an address map to use TrustZone for Armv8-M and several types of TrustZone Filters to protect memory and peripherals is found [5].

The Arm Platform Security Architecture software framework and set of API specifications, providing fundamental security functions within a common security model is described in [7].

Appendix: TBSA-M checklist [Normative]

Below is a complete list of mandatory requirements for TBSA-M. Users of this architecture should also consider Arm recommendations that might accompany the requirements included in this document.

Ref name	Base System Requirements (Section 5, Section 5.1)
R001_TBSA_BASE	The SoC must provide a hardware-based mechanism for separating the Trusted World from the Non-Trusted world.
R002_TBSA_BASE	The SoC must provide a hardware-based mechanism which is able to separate partitions within the Trusted world.
R010_TBSA_BASE	A Non-Trusted world operation must only access Non-Trusted world assets.
R020_TBSA_BASE	A Trusted world operation may access both Trusted and Non-Trusted world assets.
R030_TBSA_BASE	An SoC using TrustZone Isolation must be based on an Armv8-M architecture PE with the Security Extension and MPUs implemented.
R040_TBSA_BASE	The hardware and software of a TBSA-M device must work together to ensure that all the security requirements are met.

Ref name	Infrastructure Requirements (Section 5.2)
R010_TBSA_INFRA	A Trusted operation can issue Secure or Non-secure transactions.
R020_TBSA_INFRA	A Non-Trusted operation must only issue Non-secure transactions.
R030_TBSA_INFRA	A Non-secure transaction must only access Non-secure storage.

R040_TBSA_INFRA	If programmable address remapping logic is implemented in the interconnect then its configuration must be possible only from the Trusted world.
R050_TBSA_INFRA	A unified address map that uses target side filtering to disambiguate Non-secure and Secure transactions must only permit all Secure or all Non-secure transactions to any one region. Secure and Non-secure aliased accesses to the same address region are not permitted.
R060_TBSA_INFRA	The target transaction filters configuration space must only be accessed from the Trusted world.
R070_TBSA_INFRA	Security exception Interrupts must be wired or configured as Secure interrupt sources.
R080_TBSA_INFRA	Configuration of the on-chip interconnect that modifies routing or the memory map must only be possible from the Trusted world, unless it is not possible for such modifications to affect secure transactions.
R090_TBSA_INFRA	All transactions must be constrained; it must not be possible for a transaction to bypass a constraining mechanism.
R100_TBSA_INFRA	If shared volatile storage is implemented, then the associated location or region must be scrubbed, before it can be reallocated from Trusted to Non-Trusted.
R110_TBSA_INFRA	If shared volatile storage is implemented in a TrustZone for Armv8-M based system, then the associated location must not be executable or NSC immediately after it is reallocated from Non-Trusted to Trusted.
R120_TBSA_INFRA	An interrupt originating from a Trusted operation must by default be mapped only to a Trusted target. By default, this must be the case following a system reset.
R130_TBSA_INFRA	Any configuration to mask or route a Trusted interrupt must only be carried out from the Trusted world.
R140_TBSA_INFRA	The interrupt network may be configured to route an interrupt originating from a Trusted operation to a Non-Trusted target.
R150_TBSA_INFRA	Any status flags recording Trusted interrupt events must only be readable from the Trusted world, unless specifically configured by the Trusted world to be readable by the Non-Trusted world.
R160_TBSA_INFRA	A TBSA-M system must integrate a Secure RAM.
R170_TBSA_INFRA	Secure RAM must be mapped into the Trusted world only.
R180_TBSA_INFRA	If the mapping of Secure RAM into regions is programmable, then configuration of the regions must only be possible from the Trusted world.
R190_TBSA_INFRA	The advanced power mechanism must integrate a Trusted management function to control clocks and power. It must not be possible to directly access clock and power functionality from the Non-Trusted world.

R210_TBSA_INFRA	If access to a peripheral, or a subset of its operations, is dynamically switched between Trusted world and Non-Trusted world, then this must only be done under the control of the Trusted world.
R220_TBSA_INFRA	If the peripheral stores assets in local embedded storage, a Non-Trusted operation must not be able to access the local assets of a Trusted operation.
R230_TBSA_INFRA	A Trusted operation must be able to distinguish whether commands and data were received at an interface accessible to the Trusted world only, or at an interface accessible to the Non-Trusted world.
R240_TBSA_INFRA	A Trusted operation that exposes a Non-secure interface must apply a policy check to the Non-Trusted commands and data before acting on them. The policy check must be atomic and, following the check, it must not be possible to modify the checked commands or data.

Ref name	Fuse Requirements (Section 5.3)
R010_TBSA_FUSE	A non-volatile storage technology must meet the lifetime requirements of the device, either through its intrinsic characteristics, or through the use of error correction mechanisms.
R020_TBSA_FUSE	A fuse is permitted to transition in one direction only, from its un-programmed state to its programmed state. The reverse operation must be prevented.
R030_TBSA_FUSE	A fuse must only be programmed in accordance with its specified mechanism so that its reliable operation is not at risk.
R040_TBSA_FUSE	It must be possible to blow at least a subset of the fuses when the device has left the silicon manufacturing facility.
R050_TBSA_FUSE	All fuse values must be stable before any parts of the SoC that depend on them are released from reset.
R060_TBSA_FUSE	Fuses that configure the security features of the device must be configured so that the programmed state of the fuse enables the feature. That is, the programming of a security configuration fuse always increases security within the SoC.
R070_TBSA_FUSE	Lifetime guarantee mechanisms to correct for in-field failures must not indicate which fuses have had errors detected or corrected, just that an error has been detected or corrected. This indicator must only be available after all fuses have been checked.
R080_TBSA_FUSE	A confidential fuse whose recipient is a hardware IP must not be readable by any software process.
R090_TBSA_FUSE	A confidential fuse whose recipient is a hardware IP must be connected to the IP using a path that is not visible to software or any other hardware IP.

R100_TBSA_FUSE	A confidential fuse whose recipient is a software process might be readable by that process and must be readable by privileged software.
R110_TBSA_FUSE	A confidential fuse whose recipient is a Trusted world software process must be protected by a hardware filtering mechanism that can only be configured by secure software, for example an NS-bit filter.
R120_TBSA_FUSE	It must be possible to fix a lockable fuse in its current state, regardless of whether it is programmed or un-programmed.
R130_TBSA_FUSE	The locking mechanism for a lockable fuse can be shared with other lockable fuses, depending on the functional requirements.
R140_TBSA_FUSE	A bulk fuse must also be a lockable fuse to ensure that any unprogrammed bits cannot be programmed later.
R150_TBSA_FUSE	Additional fuses that implement lifetime guarantee mechanisms must have the same confidential and write lock characteristics as the logical fuse itself.

Ref name	Cryptographic Keys (Section 5.4)
R010_TBSA_KEY	A key must be treated as an atomic unit. It must not be possible to use a key in a cryptographic operation before it has been fully created, during an update operation, or during its destruction.
R020_TBSA_KEY	Any operations on a key must be atomic. It must not be possible to interrupt the creation, update, or destruction of a key.
R030_TBSA_KEY	When a key is no longer required by the system, it must be put beyond use to prevent a hack at a later time from revealing it.
R035_TBSA_KEY	A key must only be used by the cryptographic scheme for which it was created.
R070_TBSA_KEY	A static key must be stored in an immutable structure, for example a ROM or a set of bulk-lockable fuses.
R080_TBSA_KEY	To prevent the re-derivation of previously used keys, only Trusted code can have access to all of the source material.
R090_TBSA_KEY	If an ephemeral key is stored in memory or in a register in clear text form, the storage location must be scrubbed before being used for another purpose.
R100_TBSA_KEY	A key that is accessible to, or generated by, the Non-Trusted world must only be used for Non-Trusted world cryptographic operations, which are operations that are either implemented in Non-Trusted world software, or have both clear text and cipher text in the Non-Trusted world.
R110_TBSA_KEY	A key that is accessible to, or generated by, the Trusted world can be used for operations in both Non-Trusted and Trusted worlds, and even across worlds, provided that: The Non-Trusted world cannot access the key directly.

	The Trusted world can control the use of the key through a policy.
R140_TBSA_KEY	A Trusted hardware key must not be directly accessible by any software.
R150_TBSA_KEY	The Trusted world must be able to enforce a usage policy for any Trusted hardware key that can be used for Non-Trusted world cryptographic operations.
R160_TBSA_KEY	A TBSA-M device must either entirely embed an ROTPK, or the information that is needed to securely identify it.
R180_TBSA_KEY	An elliptic-curve-based ROTPK must be at least 256 bits in size.
R190_TBSA_KEY	An RSA-based ROTPK must be at least 3072 bits in size.
R200_TBSA_KEY	If a cryptographic hash of the ROTPK is stored in on chip non-volatile memory, rather than the key itself, it must be immutable.
R220_TBSA_KEY	A TBSA-M device must embed an HUK in confidential-lockable-bulk fuses.
R230_TBSA_KEY	The HUK must have at least 128 bits of entropy.
R240_TBSA_KEY	The HUK must only be accessible by Trusted code or Trusted hardware that act on behalf of Trusted code.

Ref name	Trusted Boot Requirements (Section 5.5)
R010_TBSA_BOOT	A TBSA-M device must embed a Boot ROM with the initial code that is needed to perform a Trusted system boot.
R020_TBSA_BOOT	If the device supports warm boot, a flag or register that survives warm boot must exist to distinguish between warm and cold boots. This register or flag must be programmable only by the Trusted world and must be reset after a cold boot.
R030_TBSA_BOOT	On a cold boot, the primary processor must boot from the Boot ROM. It must not be possible to boot from any other storage unless Trusted Kernel debug is enabled for detailed information about Trusted Kernel debug, see Section 6.10.
R090_TBSA_BOOT	If a boot status register is implemented, then it must be accessible only by the Trusted world.
R100_TBSA_BOOT	In an Assisted architecture, the key to decrypt the Trusted Boot Firmware image must be visible only to the acceleration peripheral.

Ref name	Trusted Timers Requirement (Section 5.6)
R030_TBSA_TIME	At least one Trusted timer must exist.
R040_TBSA_TIME	A Trusted timer must only be modified by a Trusted access. Examples of modifications are the timer being refreshed, suspended, or reset.

R050_TBSA_TIME	The clock source that drives a Trusted timer must be a Trusted clock source.
R060_TBSA_TIME	At least one Trusted watchdog timer must exist.
R070_TBSA_TIME	After a system reset, a Trusted watchdog timer must be started before execution of the immutable boot code transfers control to the next firmware stage.
R080_TBSA_TIME	A Trusted watchdog timer must only be modified by a Trusted access. Examples of modifications are the timer being refreshed, suspended, or reset.
R090_TBSA_TIME	Before needing a refresh, a Trusted watchdog timer must be capable of running for a time period that is long enough for the Non-Trusted re-flashing of early boot loader code.
R100_TBSA_TIME	A Trusted watchdog timer must be able to trigger a reset of the SoC, after a pre-defined period of time. This value is fixed in hardware or programmed by a Trusted access.
R110_TBSA_TIME	A Trusted watchdog timer must implement a flag that indicates the occurrence of a timeout event that causes a warm reset, to allow post-reset software to distinguish this from a powerup cold boot.
R120_TBSA_TIME	The clock source driving a Trusted watchdog timer must be a Trusted clock source.
R130_TBSA_TIME	A TRTC must be configured only by a Trusted world access.
R140_TBSA_TIME	All components of a TRTC must be implemented within the same power domain.
R150_TBSA_TIME	On initial power up, and following any other outage of power to the TRTC, a validity mechanism must indicate that the TRTC is not Trusted.
R160_TBSA_TIME	The TRTC must be driven by a Trusted clock source.

Ref name	Version Counter Requirements (Section 5.7)
R010_TBSA_COUNT	An on-chip non-volatile Trusted firmware version counter implementation must provide a counter range of at least 0 to 63.
R020_TBSA_COUNT	An on-chip non-volatile Non-Trusted firmware version counter implementation must provide a counter range of at least 0 to 255.
R030_TBSA_COUNT	It must only be possible to increment a version counter through a Trusted access.
R040_TBSA_COUNT	It must only be possible to increment a version counter. It must not be possible to decrement it.
R050_TBSA_COUNT	When a version counter reaches its maximum value, it must not roll over, and no further changes must be possible.

R060_TBSA_COUNT	A version counter must be non-volatile, and the stored value must survive a power down period up to the lifetime of the device.

Ref name	Entropy Source Requirements (Section 5.8)
R010_TBSA_ENTROPY	The entropy source must be an integrated hardware block.
R020_TBSA_ENTROPY	The TRNG must produce samples of known entropy.
R030_TBSA_ENTROPY	The TRNG must pass the NIST 800-22 test suite.
R040_TBSA_ENTROPY	On production parts, it must not be possible to monitor the analog entropy source using an external pin.

Ref name	Debug Requirements (Section 5.10)
R010_TBSA_DEBUG	All debug functionality must be protected by a DPM so that only an authorized external entity can access the debug functionality. There might be scenarios in which all external entities can access the debug functionality.
R020_TBSA_DEBUG	A DPM must be implemented either solely in hardware or together with software running in the Trusted world.
R030_TBSA_DEBUG	There must be a DPM to permit access to all assets (Trusted).
R040_TBSA_DEBUG	There must be a DPM to permit access to all Non-Trusted world assets. This mechanism must not permit access to Trusted world assets.
R050_TBSA_DEBUG	All DPMs must implement the following fuse-controlled states: • Default: Debug is permitted. • Closed: Only an unlock operation is permitted, (to transition to Open). These must be determined by a Boolean value (dpm_enable) that is stored in a public-open-bitwise fuse or derived from the Device Lifecycle state stored in fuses, see Figure 17
R090_TBSA_DEBUG	The DPM controlling Trusted world functionality must also have another fuse-controlled state: • Locked: The unlock operation is disabled (no state transition is possible). These must be determined by a Boolean value (dpm_lock) that is stored in a Public-Open-Bitwise fuse or derived from the Device Lifecycle state stored in fuses, see Figure 17.
R120_TBSA_DEBUG	All DPMs must have the following state: Open - debug is permitted.

	The Open state can only be entered from the Closed state after a successful unlock operation.
R150_TBSA_DEBUG	The Trusted world DPM must be enabled, using the respective dpm_enable fuses, or locked, using the respective dpm_lock fuses, before any Trusted world assets are provisioned to the system.
R200_TBSA_DEBUG	A password unlock token must be at least 128bits in length.
R210_TBSA_DEBUG	Each debug protection mechanism must use a unique password unlock token.
R230_TBSA_DEBUG	An unlock operation using a certificate unlock token must use an approved asymmetric algorithm to check the certificate signature.
R240_TBSA_DEBUG	An unlock operation using a certificate unlock token must have access to an asymmetric public key stored on the device. The asymmetric public key that is used to authenticate the certificate unlock token must be immutably stored on the device, or have been loaded as a certificate during secure boot and authenticated by a chain of certificates that begins with the ROTPK.
R250_TBSA_DEBUG	A certificate unlock token must indicate which DPM(s) it is able to unlock using an authenticated field.
R260_TBSA_DEBUG	A loadable public key for certificate unlock token authentication must include an authenticated field indicating which DPM(s) it is authorized to unlock.
R270_TBSA_DEBUG	A certificate unlock token must only unlock a DPM that its authenticated field is authorized to unlock.
R280_TBSA_DEBUG	The device must implement registers that, when written to by software, unlock the associated hardware debug features. Access to the secure DPM registers must be restricted to privileged trusted world software.
R290_TBSA_DEBUG	The DPM_T and DPM_NT must be implemented solely in hardware or together with firmware in immutable boot ROM.

Ref name	External Interface Peripherals Requirements (Section 5.11)
R010_TBSA_EIP	If an EIP is used to send or receive clear or unauthenticated Trusted world assets, it is implementing a Trusted operation and must meet the requirements of a Trusted peripheral.
R020_TBSA_EIP	When an EIP can receive commands from an external device, for example PCIe, then the system must enforce a policy to check that those commands do not breach the security of the TBSA-M device.
R040_TBSA_EIP	Any sensitive user data that is stored must be stored in Secure storage.
R050_TBSA_EIP	When a sensor has modes that allow it to be used for the acquisition of assets in both the Trusted world and the Non-Trusted world, activating features for Trusted world sensing must be under the control of the Trusted world.