Platform Security Guide for A-Profile

Document number:	ARM DEN 0123
Release Quality:	Alpha
Issue Number:	0
Confidentiality:	Non-confidential
Date of Issue:	26/06/21

Contents

About this document

This document is one of a set of resources provided by Arm that can help organizations develop products that meet the security requirements of PSA Certified on Arm-based platforms. The PSA Certified scheme provides a framework and methodology that helps silicon manufacturers, system software providers and OEMs to develop more secure products. Arm resources that support PSA Certified range from threat models, standard architectures that simplify development and increase portability, and open-source partnerships that provide ready-to-use software. You can read more about PSA Certified find more Arm resources here:

https://www.psacertified.org/
developer.arm.com/platform-security-resources

This Platform Security Guide for A-profile provides guidance on building a secure platform using Arm technology.

Release Information

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

Date	Version	Confidentiality	Change
June 2021	Alpha 0	Non-confidential	First alpha version.

Platform Security Guide

Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585 version 4.0

References

This document refers to the following Arm documents.

Ref	Document Number	Title
[1]	DEN 0112	Platform Threat Model and Security Goals
[2]	DEN 0128	Platform Security Model
[3]	DEN 0106	Platform Security Requirements
[4]	DEN 0072	Platform Security Boot Guide

Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
Completer	A component that responds to requests made by a Requester.
CoreSight	Debug technology from Arm
Cryptographic hash	A one-way function which maps data of arbitrary size to a bit string of fixed size.
DPM	Debug Protection Mechanism
Glue logic	Custom circuitry used to interface to off-the-shelf IP blocks.
HUK	Hardware Unique Key
IP	A reusable unit of logic, cell, or integrated circuit layout design that is the Intellectual Property of a vendor.
MAC	Message Authentication Code
MMU	Memory Management Unit within a PE
MTP	Multi-time programmable
NSAID	Non-secure Access Identifier
NVM	Non-volatile memory
OTP	One-time programmable
PE	Processing Element
Requester	A component capable making a request to a Completer.
ROTPK	Root of Trust Public Key; also known as a Boot Validation Key.

Term	Meaning
SMMU	System Memory Management Unit used to control access to memory by peripherals with Requester capability.
SoC	System on Chip
TLB	Translation Lookaside Buffer
TRTC	Trusted Real Time Clock
TrustZone	Security technology from Arm

Potential for change

The contents of this specification are subject to change.

Conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms like IMPLEMENTATION DEFINED.

Also used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link, which can be:

- A cross-reference to another location within the document.
- A URL, for example http://infocenter.arm.com.

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`.

In both cases, the prefix and the associated value are written in a monospace font, for example **`0xFFFF0000`**. To improve readability, long numbers can be written with an underscore separator between every four characters, for example **`0xFFFF_0000_0000_0000`**. Ignore any underscores when interpreting the value of a number.

Status and anticipated changes

First draft, major changes, and revisions to be expected.

Requirement identifiers will be introduced in a later version of this document.

Feedback

Arm welcomes feedback on its documentation. If you have comments on the content of this book, send an email to errata@arm.com. Give:

- The title (Platform Security Guide for A-Profile)
- The number and issue (ARM DEN 0123 1.0 Alpha 0)
- 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

This Platform Security Guide for A-profile forms part of the process for secure by design systems, as illustrated in Figure 1.The Platform Security Model (PSM) describes the essential goals and defines the terminology in a way that is agnostic of any implementation. PSM is motivated by the generic Threat Model for compute centric devices. The Platform Security Requirements (PSR) provides requirements to guide a system designer which are agnostic of any specific implementation.

This document, Platform Security Guide for A-profile, contains system recommendations that meet the security requirements described in PSR. It covers the use of the A-profile system architecture and the Armv8-A instruction set, and various trade-offs between functionality, cost, complexity, and time to market. The rationale and threat model for each requirement are described in PSR and are not described here.

These recommendations are not required for any compliance program. However, these recommendations may be useful in informing the design choices that are made by different silicon vendors for various market requirements.

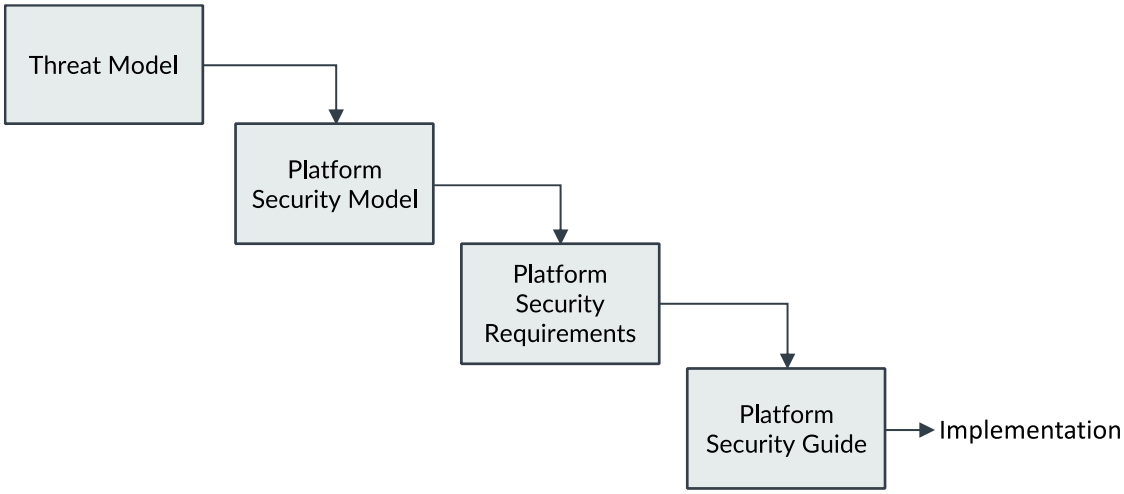

Figure 1: Document Relationship

1.1 How to read this document

Each section summarizes the security requirements from Platform Security Requirements for a particular topic, and then describes the Arm recommendations on how to meet those requirements. Other methods not mentioned in this document might also meet the requirements. Arm welcomes such feedback, see Feedback on page x.

In this version of Platform Security Guide, the Platform Security Requirements are not yet numbered. The requirements will be numbered in a future update to this document.

2 Lifecycle

The PSR requires a lifecycle control mechanism for all security functionalities. The lifecycle contains a number of irreversible states and must be captured in some on-chip non-volatile storage. Table 1 lists the requirements.

Table 1: Lifecycle requirements

PSR requirement	Description
	The security lifecycle must have a designated initial state.
	The security lifecycle must have a designated secured state which enforces the security requirements.
	The security lifecycle must have a designated terminal state from which no further transitions are allowed.
	A transition into the terminal state must put secrets and private cryptographic keys beyond use.

The number of lifecycle states depends on the supply chain of the SoC. For example, there may be several provisioning stages before reaching a designated secure state. It is typical for at least the following states to be available:

- Chip manufacturing state
- Device manufacturing state
- Secured and operational state
- Debug state
- Return-to-manufacturer, or decommissioned, state

Additional lifecycle states might be added that are specific to a market, application, or supply chain. Examples include more specific return-to-manufacturer states or diagnostic states. These lifecycle states are often orthogonal or complementary to the security lifecycle states that are described in this document.

On-chip non-volatile storage can use a variety of technologies, for example floating gate memories or oxide-breakdown anti-fuse cells. These technologies vary with respect to certain properties, including whether they are one-time-programmable (OTP) or many-time-programmable (MTP). Not all non-volatile storage technologies are available in all semiconductor processes. Where needed, off-chip non-volatile memory can be used to augment the available on-chip non-volatile storage.

Non-volatile storage technologies generally require error correction mechanisms to ensure the correct storage of data over the lifespan of the device. Because anyone with physical access can tamper with

off-chip non-volatile memory, the product threat model must indicate whether this threat must be mitigated. For mitigations, see Section 14 Secure storage.

The lifecycle state may also need to be available for attestation purposes. This implies a certain availability of the lifecycle information.

2.1 Fuses

Following the industry norm, this document uses the term fuse to refer to on-chip OTP non-volatile storage. Fuses can be implemented using anti-fuses, which are inherently one-time-programmable, or an MTP technology with controlling logic to make it one-time-programmable.

A fuse can transition in one direction only, from its un-programmed state to its programmed state. The reverse operation shall be prevented. The following guidance is provided:

- A fuse shall be programmed only once. This is because multiple programming operations might degrade the programmed cell and introduce a fault, for example, the fuse appears as un-programmed. Hardware mechanisms for OTP may be available to enforce programming restrictions that prevent operations that could lead to failure.
- All fuse values shall be stable before any parts of the SoC that depend on the value are released from reset.
- Fuses that configure the security features of the device shall be configured so that the programmed state of the fuse enables the feature. The programming of a security configuration fuse will always increase security within the SoC.

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

The full error information will be available to the lifetime guarantee mechanism. The security of the mechanism implementation must be considered. Arm recommends implementing the mechanism in hardware, but this might not always be practical.

Fuses can be programmed in one of two known ways:

- Bitwise fuses can be programmed one logical fuse bit at a time.
- Bulk fuses store multi-bit values that must be programmed at the same time and are treated as an atomic unit.

Both types of fuse need to comply with the following properties:

- Depending on system requirements it must be possible to lock any bitwise or bulk fuse in its current state, regardless of whether it is programmed or un-programmed.
- The locking mechanism for a lockable fuse can be shared with other lockable fuses, depending on the system requirements. For example, there can be one locking mechanism for all fuses that are related to a particular feature, or the set of fuses that are programmed by the silicon vendor.

- Additional fuses that are used to implement lifetime guarantee mechanisms shall have the same confidentiality and write lock characteristics as the logical fuse itself.

Assets stored in fuses have a variety of characteristics that determine the way that the fuses are accessed. These assets are classified as confidential or public fuses.

2.1.1 Fuse locking

Some fuses are programmed and locked in the factory, either by the Silicon Vendor or OEM. Other fuses may be programmed at a later point in time, depending on customer requirements.

To ensure that fuses safely retain their value, one of the following hardware approaches is recommended:

- Prevent re-writing of a locked value. A mechanism that prevents the programming of a fuse bit or group of fuse bits can be implemented by reserving an additional fuse bit to act as a lock bit. This is often called a write-lock.
- 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, only the lock bit to be set.
    - Use tamper detection to detect that the value has been modified. A tamper detection mechanism can be implemented by storing a code in additional fuses. The code must be sufficient to detect any modification to the value. The behavior upon detection will be system specific, but in general should result in a higher security state.
    - 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 shall not be returned to the system.
- Prevent reading of a fuse by un-authorized system specific components. This is often called a read-lock. See section 2.1.2.

2.1.2 Confidential fuses

Confidential fuses must only be read by the intended recipients. Examples of intended recipients include a particular hardware module or software process.

A confidential fuse whose recipient is only a hardware IP should not be readable by any software process. Typically, this type of confidential fuse is connected to the IP using a hardware data path that is not visible to software or any other hardware IP. It may be necessary to read back a bitwise or bulk fuse as part of the programming process. In such cases, a read-lock fuse can be deployed to disable such a read path once correct programming has been confirmed.

A confidential fuse whose recipient is a software process should be readable by that process. This type of confidential fuse may need to be readable by higher privileged software. This is necessary where that higher privilege software implements the process access controls. For example, a kernel level driver could provide fuse access control for user space processes. The confidentiality relies on the kernel level driver only passing fuse values to the correct user space process.

A confidential fuse whose recipient is a Trusted world software process shall be protected by a hardware filtering mechanism that can only be configured by Trusted software, for example an MPU, an MMU, or an TrustZone NS-bit filter.

2.1.3 Public fuses

Public fuses can be accessed by any piece of software or hardware. The SoC designer must consider the implications of making a fuse public on a case-by-case basis.

2.2 Authorization

A transition to a final lifecycle state, like Return Merchandise Authorization (RMA), must be part of a process involving the owner of the security lifecycle. Authorization is required to prevent unintended loss of service or loss of important data. Table 2 lists the requirements.

Table 2: Terminal state authorization requirement

PSR requirement	Description
	A transition into the terminal state must be authorized by the owner of the security lifecycle.

The owner of the security lifecycle can vary depending on the type of product that is being made. For example, an OEM may choose to restrict access to certain functionality, like full debug, until the terminal state is reached. Another example is that an OEM might allow secure boot to be disabled only after all device secrets have been put beyond use.

The method of authorization can vary according to the operational processes of the product. For example:

- If the product is registered within an ecosystem, then the ecosystem may need to add the product credentials to a deny list before permitting the transition. This ensures that any credentials leaked from the device can no longer be used to access important services.
- The product might seek approval from the owner by means of a password, pin code, or other authentication mechanism. This ensures that an unauthorized user cannot revoke secrets that are used to secure data. Also, the owner can be made aware of any data loss, like secrets, that might occur after the transition has finished. We recommend the use of multiple factors of authentication. For example, physical presence can be used as a secondary factor to counter

remote attacks. An example of physical presence is a physical button, key combination, or other form of physical user input.

3 Boot ROM and reset

3.1 Secure boot keys

The requirements listed in Table 3 indicate that the Boot ROM must be able to securely boot the next stage of the boot process. This implies that the Boot ROM has a cryptographic key to authenticate the first firmware component.

Table 3: Secure boot keys requirements

PSR requirement	Description
	The SoC must have an on-chip Boot ROM with the initial code that is needed to perform a secure boot.
	The SoC must either contain an on-chip public key for secure boot, or the information that is needed to securely identify it.
	If a cryptographic hash of the ROTPK is stored in on chip non-volatile memory, rather than the key itself, it must be immutable.

On-chip OTP NVM memory can be used to store a cryptographic hash of the key that is used to authenticate the firmware. A hash is typically smaller in size than the full key, especially in the case of the RSA algorithm. The Boot ROM uses the stored hash to check the integrity of the full key, which may be stored on external flash. Use of a hash is the most economical use of OTP NVM. The use of a hash may be more flexible with supply chains because the key hash can be programmed at a later stage of manufacturing if there is not enough OTP to program the key.

For information about appropriate cryptographic algorithms, see section 13 Cryptography.

The secure boot flow is normally divided into two phases:

- The Trusted world secure boot flow that executes Trusted world components.
- The Non-Trusted world secure boot flow that executes normal world component, for example UEFI or U-Boot.

DEN 0072 Platform Security Boot Guide provides the security requirements for the boot process. The open-source Trusted Firmware-A project demonstrates an implementation of a boot flow that meets the requirements of this document.

Boot code is highly critical to overall system security. Some markets will want to encrypt the firmware images to make it more difficult for attackers to identify vulnerabilities within the boot code. This encryption would require an extra on-chip key for decryption, typically held in fuses or equivalent on-

chip NVM. To simplify firmware provisioning, the same decryption key is often used across a class of devices.

3.2 Warm boot

When a system boots from a powerless state, it is known as a cold boot. Some systems support a type of reset where the system boots from a powered state. This type of boot is called a warm boot. Because a warm reset only resets a subset of system components, it allows the boot process to skip certain reinitialization routines and potentially reuse RAM contents from the previous boot session. This allows for fast reboots but also raises security concerns. Table 4 lists the requirements. Recommendations will be included in a future update to this document.

Table 4: Warm boot requirements

PSR requirement	Description
	If the system supports warm boot, a flag or register that survives warm boot must exist to distinguish between warm boots and cold boots.
	Where a flag or register that survives warm boot exists to distinguish between cold and warm boot, the flag or register must be programmable only by the Trusted world and must be reset during a cold and a warm boot.
	Where a flag or register is used to distinguish between cold and warm boots, the default should be for cold boot, and should use a value that any unauthorized perturbation will result in a cold boot.
	If a boot status register is implemented, it must be accessible only by the Trusted world.

3.3 Boot parameters

Table 5 lists the boot parameter requirements. Recommendations will be included in a future update to this document.

Table 5: Boot parameter requirements

PSR requirement	Description
	The Boot ROM must be aware of the current security lifecycle state.
	It must not be possible to bypass secure boot unless a Trusted Debug mode permits this.
	Any Boot ROM configuration data outside of on-chip OTP memory must be authenticated using an on-chip public key.

3.4 Boot robustness

It is desirable to minimize the amount of code that is embedded in the Boot ROM. This is because of the possibility of code issues or vulnerabilities. For example, a cryptographic library or USB stack could contain a non-trivial amount of code. Because the Boot ROM is the first stage of a secure boot process, any exploitable vulnerability could be used to either bypass secure boot, or to create system instability.

A simplified Boot ROM implementation should contain only the necessary initialization code, and a hash algorithm that is used to verify a loadable code module with a stored cryptographic hash in OTP NVM. The loadable code can include more complicated code, like asymmetric cryptographic algorithms and additional device drivers. This allows a greater portion of Boot ROM code to be updated without a metal mask change and permits later delivery of that code during SoC development.

When the Boot ROM is invoked, it typically sets up an execution stack and may temporarily use other parts of memory for large objects.

To prevent DMA transactions changing the execution state, Arm recommends disabling DMA for all requesters on reset. This is a simple way to ensure compliance with Table 6. However, this might be too restrictive for some system designs. Alternatively, the SoC designer can use a memory access filter that prevents unauthorized writes to the Boot ROM execution environment and other relevant places.

Table 6: Boot robustness requirements

PSR requirement	Description
	The Boot ROM execution state must be protected from DMA reads and writes.

3.5 Temporal isolation

The Boot ROM, or an early boot stage component, might have immutable secrets that should not be accessible to runtime software. If the secrets are left accessible, a runtime attack might leak them. Because these secrets are not needed by runtime software, they should be hidden by the boot software using a hardware mechanism. An example of a hardware mechanism would be an SoC register that hides portions of the address map once it is written to. That register should ideally be write-once, or writeable only by trusted software.

Examples of such immutable secrets may include:

- Boot ROM code
- Hardware unique key
- Root parameters, like SoC or IP parameters

A typical usage scenario for temporal isolation is to hide a single hardware unique secret. Key Derivation Functions (KDF) can then be used to derive multiple hardware unique keys. Each derived key can be given to a specific runtime component and used only for a specific purpose. With only the output of a

good key derivation function, it is technically infeasible to determine the secret used to derive a key. Arm recommends reading NIST 800-56C for examples of good KDF algorithms and usage.

An example of the various keys that can be derived is shown in Figure 2, where the Boot ROM or equivalent passes the following into a key derivation function:

- The hardware unique key
- The current security lifecycle state
- The purpose of the derived key

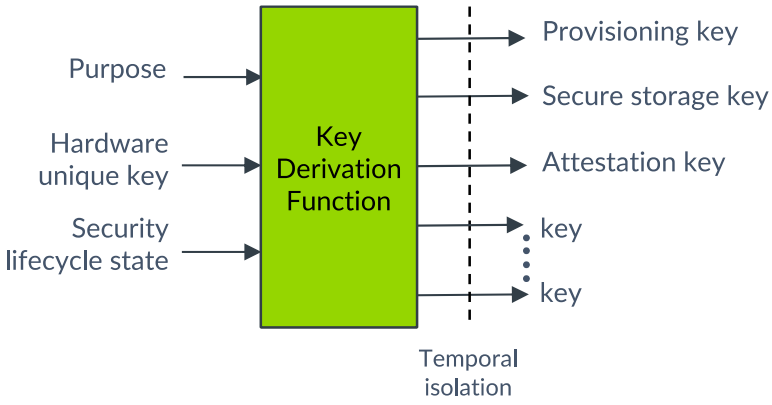

Figure 2 Key derivation with temporal isolation

The Boot ROM uses the KDF to generate all the required keys. The derived keys can be used for specific purposes, like provisioning, secure storage, attestation, and potentially other types of keys. Before the derived keys are given to their respective components, the Boot ROM activates the temporal isolation to hide the hardware unique key from access. Typically, the temporal isolation is achieved by implementing a sticky bit in a hardware register, which when set, causes the location of the hardware unique key to be inaccessible until the next system reset.

Because the security lifecycle state can be included as a parameter in the derivation process, a change in the security lifecycle state will change the derived keys. There are a few useful scenarios for this property:

- A valid provisioning key can only be derived during a factory provisioning state.
- A valid attestation key can only be derived when the system is not in a debug state.
- A valid secure storage key can only be derived when the system is not in a debug or decommissioned state.

More keys and detailed policies can be created using the same mechanism.

3.6 Reset

During normal operation, all external interfaces to the SoC should be disabled. They may be selectively enabled by the Boot ROM or secure boot firmware when they are needed. Table 7 lists the requirements.

Table 7: External Interface reset requirements

PSR requirement	Description
	External interfaces must be disabled on each cold reset.

If the SoC implements multiple processing elements (PEs), the designated boot processor core is called the primary PE. After the de-assertion of a reset, the primary PE usually executes first, while the secondary PEs are held in reset, or a safe platform-specific state, until the primary PE initializes and boots them. A more complex SoC might have a dedicated security co-processor. This co-processor would control the reset logic holding PEs in a halted or safe state, load and validate firmware, and then allow the Primary PE to execute. Table 8 lists the requirements.

Table 8: Secondary PE reset requirements

PSR requirement	Description
	All secondary PEs must remain inactive until permitted to boot by the primary PE.

4 Clock and power

This section lists the clock and power related security requirements and some design options that can meet the requirements.

4.1 Power management operations

Power management is a sensitive operation that can affect the execution of the Trusted world and its operations. Table 9 lists the requirements. Therefore, it is important that sensitive power management operations are not handled by the Non-trusted world:

- The basic core management operations, like turning CPUs on or off, is typically implemented in the Trusted world, typically at EL3. The Non-trusted world software communicates with the Trusted world power management code using the Power State Coordination Interface (PSCI) defined by Arm.
- Dynamic Voltage and Frequency Scaling (DVFS) controls is typically implemented by a separate on-chip System Control Processor (SCP). The Non-trusted world software communicates with this processor using the System Control Management Interface (SCMI) defined by Arm.

In both cases, the parameters must be checked by the trusted entity to ensure they remain within safe limits.

Where possible, non-critical or non-sensitive power management code should run in the Non-trusted world in order to minimize the attack surface of the Trusted world and System Control Processor.

Table 9: Power management requirements

PSR requirement	Description
	The power control mechanism must integrate a Trusted management function to control clocks and power. It must not be possible to directly access reset, clock, and power management mechanisms from the Non-Trusted world.

4.2 Glitch resistance

A non-invasive physical attacker can perturbate the clock or power lines to cause malfunction. These injected faults can sometimes be exploited by an attacker to elevate privileges or leak secrets. The required training and equipment cost to perform this type of attack is becoming increasingly affordable to adversaries. The need to protect against these attacks will depend upon the Protection Profile of any applicable certification scheme or industry segment. To counteract these types of attack it is necessary to employ counter measures, for example:

- Ensuring that Trusted software always fails in a secure and safe way when a glitch occurs. This may be impractical depending on the amount of software that needs protection. A more comprehensive description of the topic is described the paper Secure Application Programming in the presence of Side Channel Attacks, published by Riscure.
- Including a monitoring unit within the SoC that detects if the clock, temperature or voltage vary beyond safe limits.

Arm recommends a mixture of techniques to obtain good coverage against these attacks.

4.3 Suspend or hibernate

A system may need to support hibernate or suspend operations. These operations put the SoC into a very low power state, while the main memory may be copied to some non-volatile storage device like flash or a hard disc drive or the DRAM may be placed in a self-refresh mode. Table 10 lists the requirements. Recommendations on suspend-to-DRAM scenarios will be included in a future update to this document.

Table 10: Suspend or hibernate requirements

PSR requirement	Description
	If suspend to RAM is implemented and the main die is powered down so that any DRAM protection keys need to be saved and restored, these operations must be handled by the Trusted world.
	The keys must be stored in either on-chip Trusted storage or wrapped using a key derived from an on-chip hardware unique key.
	Security critical suspend state information that is stored off-chip must be encrypted and authenticated using an on-chip key.

5 Memory system

The Security Model [1] requires as a minimum hardware-based isolation between the security sensitive processing and the remainder of the software. This is reflected in Table 11. For isolation of memory, we describe Arm TrustZone technology in this section.

Table 11: Memory isolation requirements

PSR requirement	Description
	The SoC must provide a hardware-based mechanism for isolating the memories of the Trusted world from the Non-trusted world.

5.1.1 TrustZone technology overview

TrustZone technology lies at the heart of an Arm processor element. While it is executing code, the processor element can operate in one of two possible states that correspond to the Trusted and Non-trusted worlds. These are known as the Secure and Non-secure states, respectively. Context switches between Security states can only be made using dedicated instructions and code to ensure that strict isolation is maintained. The context switch mechanism enforces fixed code entry points and ensures that code running in the Non-secure state cannot access registers that belong to the Secure state. Conceptually, the Secure and Non-secure states can be regarded as two virtual processor elements.

When the Arm processor performs a memory access, the MMU translation provides an extra bit that indicates the security state that is associated with the transaction. When this bit is set, it indicates a Non-secure (NS) transaction. The mechanism is tightly coupled to the cache and consequently an NS bit is stored in every cache line. The NS bit can be considered as an extra address bit that is used to access the Trusted and Non-trusted worlds as completely independent address spaces.

When a memory access reaches the external bus, the NS bit from the cache is translated into two transaction bits: one NS bit for reads and one NS bit for writes. The on-chip interconnect must guarantee that these bits are propagated to the target of the access, and the target must determine from the address and NS bits if the access is to be granted or denied.

By propagating the security state of the processor element through the on-chip interconnect to target based transaction filters, the TrustZone technology is extended into the SoC design, creating a robust platform supporting fully isolated Trusted and Non-trusted worlds.

Figure 3 shows the components typical of an SoC based on TrustZone technology. The processor cluster is supported by a number of security hardware IPs that utilize TrustZone technology, like the NS-bit, to work within the Trusted world. Memory protection is used to isolate the target application from other applications at runtime by providing a partitioning of on-chip and off-chip memory into the Trusted and Non-trusted worlds. The partitioning is achieved using NS bit target-based filtering. The configuration must be performed by the Trusted world.

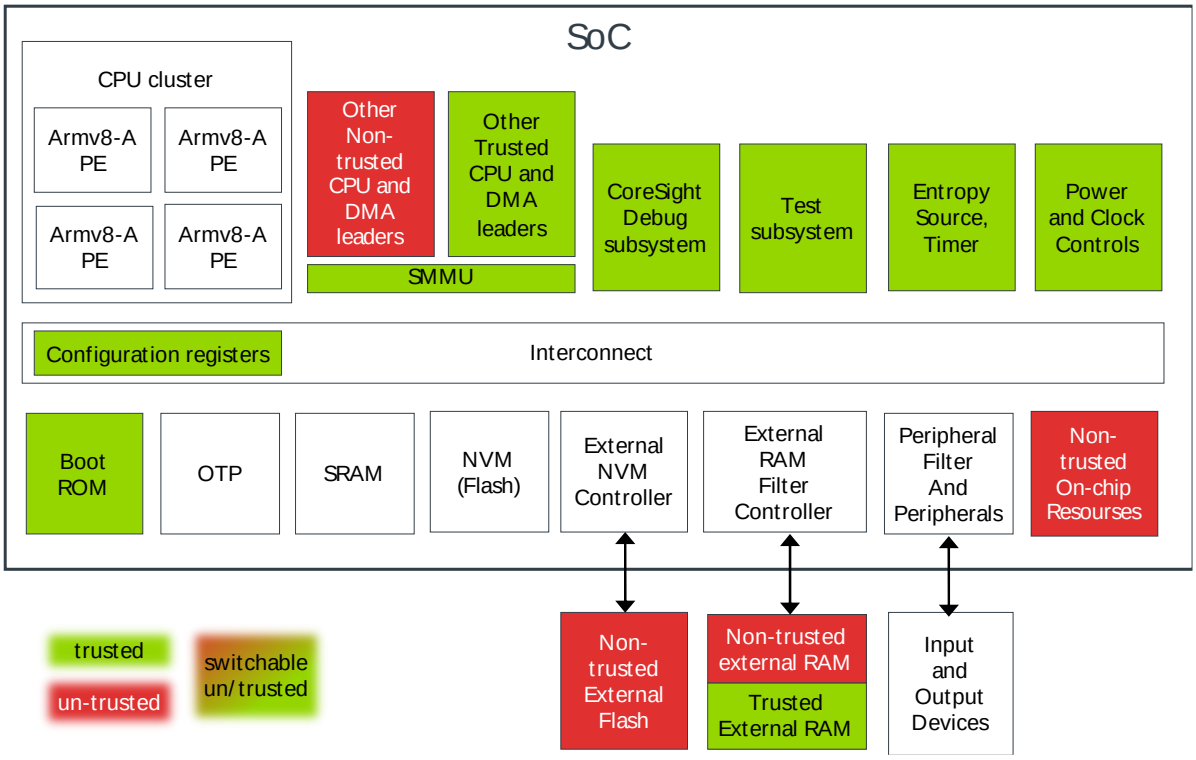

Figure 3: SoC components isolated using TrustZone technology

When the processor is in the Non-secure state (EL2, EL1, or EL0), all memory transactions are Non-secure.

When the processor is in the Secure state (EL3, S-EL2, S-EL1, or S-EL0), the security state of memory transactions is determined by the MMU:

- If the MMU is enabled, the memory transaction type is determined by the translation tables.
- If the MMU is disabled, all Secure state accesses default to Secure transactions on the bus.

Further partitioning of the Non-trusted world can be achieved by using a hypervisor that controls the MMU and System Memory Management Units (SMMU). Alternatively, Non-secure Access Identifier (NSAID) filters can be used, though they may be less flexible. For more information, see the Arm System memory Management Units documentation and the TrustZone Controller documentation.

An important property of a TrustZone system is that a Trusted service might access both Secure and Non-secure memory. Table 12 lists the requirements. To achieve this, two possible approaches are evident:

- A Trusted service can issue either Secure or Non-secure memory transactions. Transaction filters only permit a Secure transaction to access Secure memory, and therefore a Secure transaction cannot access Non-secure memory. This is the recommended approach.

- A Trusted service always issues Secure memory transactions, but the transaction filters permit a Secure transaction to access any memory, Secure or Non-secure. This approach has been implemented in legacy systems but Arm no longer recommends this approach.

When using the first approach, software that is executing in a Secure state that wants to access Non-secure memory must explicitly issue Non-secure memory transactions. This is done through translation table control flags in the Translation Lookaside Buffers (TLB). The second approach leads to aliased entries in the cache and the TLB, which can cause coherency and security problems.

Table 12: TrustZone implementation requirements

PSR requirement	Description
	A Trusted operation can issue Secure transactions, and might be able to issue Non-secure transactions.
	A Non-Trusted operation must only issue Non-secure transactions.

5.2 Physical memory mapping

Figure 4 shows how resources, for example a set of memory mapped peripheral interfaces, are placed into the physical memory map that is based on the world they belong to.

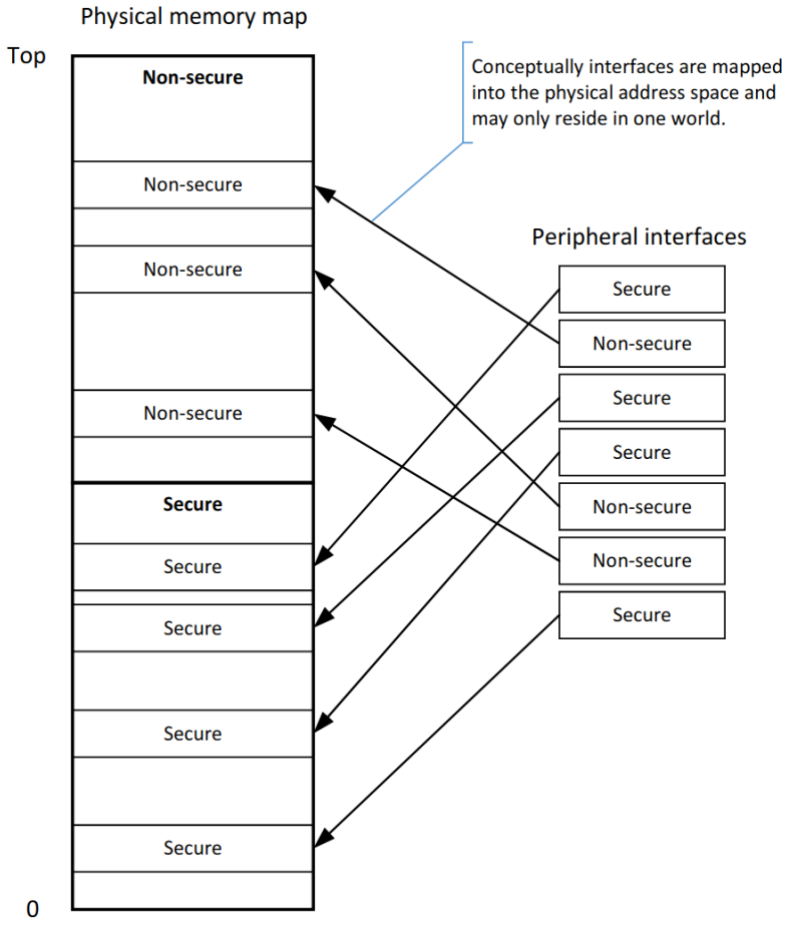

Figure 4 Physical memory map with peripherals mapped into multiple worlds

If the peripherals are grouped together on a local interconnect node, the required mapping can be achieved through memory translation. Figure 5 shows the incorporation of a DRAM that is divided into Secure and Non-secure regions, using remapping logic.

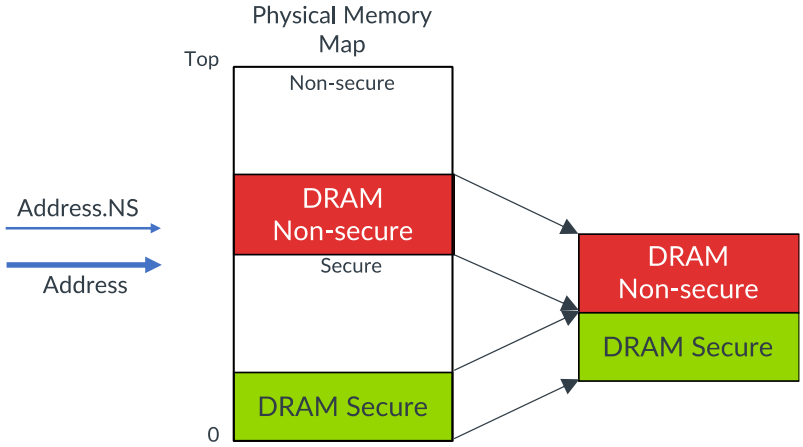

Figure 5 Physical memory map with external DRAM mapped into multiple worlds

In this example, and in many real-world cases, the DRAM is simply split into only two regions, Secure and Non-secure. To map the two DRAM regions correctly into the larger physical address map, remapping logic must be implemented. In simple implementations this can be fixed logic. However, using programmable logic offers greater flexibility if software is updated. If programmable logic is used then the relevant configuration registers must belong to the Trusted world, and therefore only be accessible to Secure transactions.

In general, the mapping of resources into Secure or Non-secure memory can be achieved using either fixed or programmable logic, for example TLB-based translations of the physical address, or using a target-based filter. This type of filter creates Secure and Non-secure memory regions using ranges based on all address bits, except ADDRESS.NS. Incoming transactions are permitted only if the following conditions are true:

- Region is Secure and ADDRESS.NS = 0
- 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 6 shows the resulting address map.

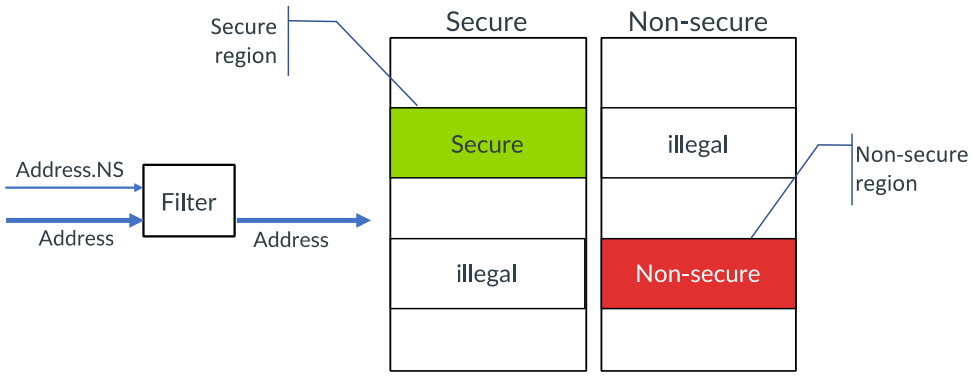

Figure 6 Physical address space filters

The aliasing in the address map that results after filtering places constraints on the memory layout from the point of view of a bus requester, for example an Arm processor.

At the interconnect level, and before filtering, ADDRESS.NS forms an additional address bit, and each memory transaction must include this bit together with all other address bits to the point where the filter constraints are applied.

The Arm TrustZone Address Space Controller (TZC) is an example of this type of target-based filter. In a TZC filter, a region can be configured as accessible by any combination of accesses. For example, it is possible to configure a region to be accessible to both Secure and Non-secure transactions. This violates the security requirements described in 5.1.1. The TZC filter can be configured to silently block illegal transactions, or to block and signal a security exception through a bus error or an interrupt. If an interrupt is generated, it is classified as a Trusted interrupt.

Memory access requirements are listed in Table 13.

Table 13: Memory access requirements

PSR requirement	Description
	If programmable address remapping logic is implemented in the interconnect, its configuration must be possible only from the Trusted world.
	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.
	The target transaction filters configuration space must only be accessed from the Trusted world.

5.3 Transaction routing and cache control

Bus transactions must propagate the security state, called the NS bit. For the Arm processor element, the security state of the transaction is made available, so that it can be 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, where:

- ARPROT[1] indicates a Trusted write when low.
- AWPROT[1] indicates a Trusted read when low.

Similarly, a hardware IP that is an AXI bus requester will generate the same signals to indicate the security state of each transaction. In system designs that use a network-on-chip interconnect approach, it might be possible to re-configure the routing of packets so that they arrive at a different interface. Even though the access address remains unchanged, this is dangerous and can lead to an exploit. Any such configuration shall only be possible from the Trusted world using Secure transactions. The requirements are listed in Table 14.

The different techniques for address remapping and filtering are both methods of constraint that bind storage locations to worlds. Whatever the method of constraint, it must not be possible for a memory transaction to bypass it. A particular example is the case where multiple caches that are up-stream from a target filter are synchronized using a coherency mechanism. If this type of mechanism, for example bus snooping, is implemented, the mechanism must force a coherency transaction to pass through the target filter.

Table 14: Memory-map configuration requirements

PSR requirement	Description
	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.

Any Snoop Control Unit (SCU) should be protected from the Non-trusted world to ensure that the Trusted world timing and behavior is not influenced by execution in the Non-secure state. The SCU Secure Access Control Register determines whether Non-secure accesses can reprogram the SCU configuration registers or access the processor-local timers.

Secure and Non-secure entries can co-exist within caches and TLBs. There is no need to invalidate cache data when switching between worlds. The Non-secure state can only generate Non-secure accesses, so can only hit on cache lines marked as Non-secure. However, the Secure world can generate both Secure and Non-secure accesses, which may require some cache management if the security state is changed between accesses.

Entries in the TLB record which world generates a particular entry, and although the Non-secure state can never operate on Secure data, the Secure world can allocate non-secure cache lines. Also, the caches are enabled and disabled separately for each of the Exception levels. Cache control is independent for each world. But cache control is not independent for all Exception levels, for example, EL0 code can never enable or disable the caches directly, and EL2 can override the EL1 cache behavior.

5.4 Instruction fetches

Table 15 lists the requirements related to secure instruction fetches. Secure instruction fetch, SCR_EL3[SIF] is a mechanism in the Arm architecture that should be set to prevent Secure world from executing Non-secure code. When enabled, this setting can prevent attacks where the Secure world is made to branch to attacker-controlled code in the Normal world.

Table 15: Instruction fetch requirements

PSR requirement	Description
	A Secure transaction must not fetch Non-secure instructions

6 Processing elements

Various Armv8-A architecture features meet the PSR requirements for processing elements.

6.1 Isolation of execution contexts

Table 16 lists the requirements for isolating execution context.

Table 16: Isolation of execution context requirements

PSR requirement	Description
	The SoC must provide a hardware-based mechanism for isolating the execution contexts of the Trusted world from the Non-Trusted world.

The Arm architecture defines four Exception Levels (ELs), ranging from EL0 (least privileged) to EL3 (most privileged). Higher levels can protect against interference from lower exception levels by using the MMU and setting up exception handlers. The processor element also executes in one of two Security states, called Secure and Non-secure. As a result, exception levels and privilege levels are defined within a particular Security state. Table 17 summaries the Exception Levels and Security states.

Table 17: Exception levels and security states

Exception level	Non-secure state	Secure state
0	Name: EL0 Runs: User space World: Non-Trusted	Name: S-EL0 Runs: Trusted Applications World: Trusted
1	Name: EL1 Runs: Kernel space World: Non-Trusted	Name: S-EL1 Runs: Trusted OS World: Trusted
2	Name: EL2 Runs: Hypervisor space for virtualization support World: Non-Trusted	Name: S-EL2 Runs: Secure Partition Manager World: Trusted
3	N/A	Name: EL3 Runs: Secure monitor for security state control World: Trusted

When the processor is in the Non-secure state, EL2, EL1, or EL0, all memory transactions are Non-secure.

When the processor is in the Secure state, EL3, S-EL2, S-EL1, or S-EL0, the security state of a memory transaction is determined by the MMU. If the MMU is enabled, the memory transaction type is determined by the translation tables. If the MMU is disabled, all Secure state accesses default to Secure transactions on the bus.

A processor core configuration bit in the Security Configuration Register (SCR) determines the security state of exception levels below EL3. This bit can only be updated by EL3 software. In the AArch64 64-bit architecture the bit is SCR_EL3.NS, which when set selects the Non-secure state.

Transitions between the Trusted world and the Non-trusted world are managed by a dedicated software module called the Secure Monitor that runs at EL3. The Secure Monitor must support the safe save and restore of processor state and maintain the isolation between the two worlds.

A Secure Monitor Call (SMC) instruction is used to enter EL3 and invoke the Secure monitor code. Because this instruction can only be executed in a privileged mode, a user process that requests a change from one world to the other must do so using an SVC instruction, which is usually done using an underlying OS kernel. Furthermore, an SMC should be trapped by EL2, and prevent even the OS kernel (EL1) from directly invoking the Secure Monitor (EL3). Allowing an untrusted virtual machine to issue SMCs to EL3 may lead to privilege escalation attacks.

Interrupts and exceptions can also be configured to cause the processing element to switch into EL3. Independent exception and vector tables support this functionality.

The Armv8.4-A architecture introduces the Secure EL2 extension, adding support for virtualization in the Secure world. This brings the features that are available for virtualization in the Non-secure state to the secure state. A key feature in virtualization support is the addition of a hypervisor-controlled second stage of translation. This allows a hypervisor to control which areas of physical memory are available to a virtual machine.

The Arm System MMU architecture 3.2 and higher supports stage 2 translations in the secure state for other I/O requesters. See section 9.2.

Together, these features bring virtualization support to the Secure world and provide a hardware basis that meets a few practical architectural security requirements. In short, secure virtualization enables:

- Isolating EL3 software from Secure EL1 software
- Isolating distinct Secure EL1 software components from each other
- Restricting Secure EL1 software from accessing parts of the Non-trusted world

The architecture extensions provided by Secure EL2 enables trusted world virtualization. Without S-EL2, a Trusted OS, executing at S-EL1, can access any location in the system address map. Because the Trusted OS is a privileged entity, a weakness in its implementation could be exploited to access any memory address in the system, secure or non-secure. Indeed, privilege escalation attacks like this have been reported on some Trusted OS implementations. Secure EL2 using stage 2 translation can mitigate this escalation by restricting the memory regions that are accessible by software executing at Secure EL1

or Secure EL0. Control of stage 2 translations in the secure state of a System MMU extends this protection to include DMA-capable trusted hardware resources.

S-EL2 software must be carefully designed to ensure it is robust against attacks. An implementation is provided by the Trusted Firmware open-source project.

6.2 Non-executable memory

Buffer overflows are a common software vulnerability caused by memory unsafe languages. Stack smashing exploits, and similar variants, use buffer overflow attacks to introduce malicious code into memory and trigger execution. Because this form of attack is well known, can be performed remotely, and does not require high skill, it is essential that mitigations are provided.

To counteract this common form of attack, the industry has developed system controls that allow memory regions to be configured as either executable or non-executable. When a region is executable, it can no longer be written to by the program. It is typical for a program loader or startup code to configure code regions as executable, while the rest of the memory space is configured as non-executable. Table 18 lists the requirements.

Table 18: Non-executable memory requirements

PSR requirement	Description
	The SoC must provide a hardware-based mechanism that ensures runtime data in memory is never executable.

The Virtual Memory System Architecture (VMSA) describes the execute-never bit (XN). The XN bit can be used by privileged software to mark data memory as non-executable and mark executable memory as read-only. More specifically, UXN and PXN bits can be used to provide more granular permissions across exception levels.

The architecture also provides control bits in the system control register, SCTLR_ELx, to make all writable addresses non-executable. Enabling this control makes locations like the stack non-executable. A location that is writable at EL0 is never executable at EL1, regardless of how the PXN and SCTLR_ELx controls are configured.

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

6.3 Speculation control

The vulnerability underlying cache timing side-channels is that the pattern of allocations into a CPU cache, and which cache sets have been used, can be determined by timing. Measuring the time taken to access entries that were previously in the cache or to access entries that have been allocated, may leak information that could be read by other, less privileged software.

The novel feature of speculation-based cache timing side-channels is their use of speculative memory reads, which are common in very high-performance CPUs. By performing speculative memory reads to cacheable locations beyond an unresolved branch or other change in program flow, the result of those reads can themselves be used to form the addresses of further speculative memory reads. These speculative reads cause allocations of entries into the cache whose addresses are indicative of the values of the first speculative read. This becomes an exploitable side-channel if untrusted code can control the speculation such that a first speculative read is to a location it would not otherwise be able to accessible. But the effects of the second speculative allocation within the caches can be measured by that untrusted code.

Table 19 lists the requirements for speculation control.

Table 19: Speculation control requirements

PSR requirement	Description
	If a processor implements speculative execution, it must provide a way for software to control or disable speculation.

The Arm architecture introduced new barrier instructions that allows software to have fine-grain control over processor speculation. The barrier instructions were first described in Armv8.5-A and can be implemented from Armv8.0-A onwards. In summary, these barrier instructions provide the following controls:

- The CSDB (Consumption of Speculative Data Barrier) instruction ensures that no instruction, other than a branch, in program order after the CSDB can be speculatively executed. Such speculation could be based on the results of any data value predictions of any instructions, on PSTATE.NZCV predictions of any instructions other than a conditional branch, or SVE state predictions for any SVE instruction in program order before the CSDB that have not been resolved.
- The SSBB (Speculative Store Bypass Barrier) instruction ensures that any stores before the SSBB using a virtual address will not be bypassed by any speculative executions of a load after the SSBB to the same virtual address. The SSBB barrier also ensures that any loads before the SSBB to a particular virtual address will not speculatively load from a store after the SSBB.
- The PSSBB (Physical SSBB) instruction ensures that any stores before the PSSBB using a particular physical address will not be bypassed by any speculative executions of a load after the PSSBB to the same physical address. The PSSBB barrier also ensures that any loads before the PSSBB to a particular physical address will not speculatively load from a store after the PSSBB.

Processors that do not implement these instructions may instead provide a control register to completely disable speculative execution or disable the re-ordering of stores and loads. This is likely to have a more profound impact on performance compared to using cores that implement the barrier instructions.

6.4 Control flow integrity

The combination of two architecture extensions, Pointer Authentication (from Armv8.3-A onwards) and Branch Target Identification (from Armv8.5-A onwards), provide hardware-based control flow integrity.

Pointer authentication is a countermeasure for Return-Oriented Programming (ROP) exploits. Pointer authentication uses the fact that the actual address space in 64-bit architectures is less than that addressable by 64-bits. There are unused bits in pointer values that can be re-purposed to hold a Pointer Authentication Code (PAC) for a pointer. A PAC could be inserted into each protected pointer before writing it to memory, and then the pointer integrity verified before it is used. An attacker who wants to modify a protected pointer would have to be able to generate the correct PAC to be able to control the program flow. The pointer authentication scheme makes it difficult for an attacker to generate a correct PAC code.

Different pointers have different purposes within a program. Pointers should be valid only in a specific context. In pointer authentication, two parts to ensure this:

- Separate keys for major use cases
- Computing the PAC over both the pointer and a 64-bit context

The Arm pointer authentication specification defines five keys:

- Two for instruction pointers
- Two for data pointers
- One for a separate general-purpose instruction for computing a Message Authentication Code (MAC) over longer sequences of data

The instruction encoding determines which key to use. The context is useful for isolating different types of pointers used with the same key. The context is specified as an additional argument together with the pointer when computing and verifying the PAC. The PAC is added and checked by special instructions, and the PAC uses a cryptographically strong algorithm to resist forging. ROP exploits can be defended against by protecting a return address with a PAC.

Branch Target Identification (BTI) provides a defense against Jump-Oriented Programming (JOP) exploits. BTI ensures that indirect branches must land on corresponding instructions. In the BTI mechanism, every indirect instruction is paired with a corresponding legal instruction. Almost all other instructions are invalid branch targets, and branching to an incompatible instruction raises a branch target exception. Forcing branches to land on certain instructions makes it difficult for an attacker to find desirable gadgets and therefore defends against the use of this technique. Branch target identification is available for PEs implementing Armv8.5-A and later.

More information is available in the Arm white paper Providing protection for complex software.

6.5 Memory safety

The use of memory-unsafe languages like C and C++ means that memory related errors will continue to be vulnerable to exploitation. These include bounds violations, use-after-free, use-after-return, use-out-of-scope, and use-before-initialization errors.

The Memory Tagging Extension (MTE) provides architectural support for run-time, always-on detection of various classes of memory error. Memory tagging, also known as coloring, versioning, or tainting, can aid software debugging to eliminate memory related vulnerabilities before final code production.

Memory tagging is a lightweight, probabilistic version of a lock and key system. One of a limited set of lock values can be associated with the memory locations forming part of an allocation, and the equivalent key is stored in unused high bits of addresses used as references to that allocation. On each use of a reference the key is checked to make sure that it matches with the lock before an access is made. On freeing an allocation, the lock value that is associated with each location is changed to one of the other lock values. This means that further uses of the reference have a reasonable probability of failure. This extension implements support for storage, access and checking of the lock values in hardware, allowing software to select and set the values on allocation and deallocation. The general idea of memory tagging on 64-bit platforms is as follows:

- Every tagging granularity (TG) byte of memory aligned by TG is associated with a tag of tag size (TS) bits. These TG bytes are called the granule.
- TS bits in the upper part of every pointer contain a tag.
- Memory allocation chooses a tag, associates the memory chunk being allocated with this tag, and sets this tag in the returned pointer. For example, this could be implemented within the standard C library function malloc.
- Every load or store instruction raises an exception on mismatch between the pointer and memory tags.
- On freeing an allocation, the tag associated with each location is changed to one of the other tag values. This means that further uses of the reference have a reasonable probability of failure.

The Armv8.5-A Memory Tagging Extension, implements support for storage, access and checking of the lock values in hardware, allowing software to select and set the values on allocation and deallocation. Implementing the Memory Tagging Extension requires the system architecture to provide additional storage for memory tags. There is execution cost in tagging heap and stack objects during allocation and deallocation. It is possible to use memory tagging only during development. However, there is compelling evidence that testing alone will miss many memory-safety bugs which occur in deployment. The benefits and costs of always-on detection should be weighed by architects in the context of the threat model.

7 Interrupts

Each world typically needs to access peripherals or IP blocks that generate interrupts. Isolation of interrupts is important to prevent leakage of sensitive system information from the Trusted world to the Non-trusted world. Therefore, interrupts need to be assigned to the appropriate world at design time. In some cases, a peripheral or IP block may be able to generate interrupts in multiple worlds if it is configured to do so. It is important that the Trusted world peripherals only generate Trusted world interrupts, unless specified otherwise by Trusted world software. Table 20 lists the requirements.

Table 20: Interrupt assignments requirements

PSR requirement	Description
	An interrupt originating from a Trusted operation must by default be mapped only to a Trusted target.

In the Armv8-A system architecture, the Generic Interrupt Controller (GICv3) supports isolation of interrupts between worlds by assigning each interrupt to one of the groups listed in Table 21. Each interrupt group determines the Security state for interrupts in that group, depending on the Exception level of the core. Separate enable bits control whether interrupts in a group can be forwarded to the PE.

Table 21: Allocation of groups to security states

Group 0	Secure Group 1	Non-Secure Group 1
Typically used by EL3 firmware, the Secure Monitor	Typically used by a Trusted OS (S-EL1) or Secure Partition Manager (S-EL2)	Typically used by a Rich OS (EL1) or a hypervisor (EL2)

Interrupt handling is dependent on the security state of the PE:

- When an interrupt needs to be processed in the current world, the world directly handles the interrupt.
- When an interrupt needs to be processed in a different world, then the software needs to transfer control to the other world, which will then handle the interrupt. Upon completion, software is expected to transfer control back to the world that was interrupted.

Software at S-EL1 also has the option to configure the GIC to trap interrupts destined to the Non-Secure Group. More information is available in the Arm Generic Interrupt Controller Architecture Specification for version 3.0 and 4.0.

Table 22: Security exceptions and interrupt requirements

PSR requirement	Description
	Security exceptions or interrupts must only be handed by the Trusted world
	Any configuration to mask or route a Trusted interrupt must only be carried out from the Trusted world.

Non-trusted world access to information about Trusted world activity needs to be kept to a minimum, Table 23 lists the requirements. For example, Trusted world exceptions, interrupts, or other information that can aid timing attacks. If a fatal error occurs in the Trusted world, it should not leak information to the Non-trusted world.

The register GICD_CTLR[DS] (disable security) must be set to 0. This ensures Non-secure accesses are not permitted to access and modify registers that control Group 0 interrupts.

Table 23: Interrupt status requirements

PSR requirement	Description
	Any status flags recording Trusted interrupt events must only be read from the Trusted world, unless specifically configured by the Trusted world to be readable by the Non-Trusted world.

Some SoCs contain a dedicated microcontroller block to handle reset lines and general power management. For example, on Arm-produced development platforms, this is known as a System Control Processor (SCP). More complicated SoCs may also have a Manageability Control Processor (MCP) to provide out-of-band, zero touch management.

Interrupts from certain peripherals might be wired directly to these subsystems instead of the application PEs. These subsystems must ensure that Trusted world related data is not leaked to any Non-trusted entity.

8 Debug

A debug infrastructure provides standardized physical entry ports into devices, typically through SWD or JTAG. Devices may contain one or more debug ports, and may be used for a different purpose used across the device lifecycle, for example:

- Flash programming
- Provisioning of Trusted world secrets
- Loading code to RAM for execution
- Memory upload/download
- Device configuration
- Extraction of data or logs
- Recovery of bricked devices
- Tracing, both invasive and non-invasive
- Breakpoints during development

Operationally, these features support the following scenarios:

- In-field service, obtaining diagnostics from deployed devices, failure analysis
- Secure multi-stage manufacturing and credentialing, where multiple parties each provide parts of the firmware without necessarily sharing mutual trust
- Return Merchandise Authorization (RMA) mechanisms
- Prototyping and testing

Debug capabilities may allow malicious attackers to extract sensitive data or take control of a device using debug ports. The aim of debug security is to limit access to debugging capabilities to authorized parties, possibly segmenting access rights to effectively limit what a debugger can and cannot do.

8.1 Debug ports

An Arm SoC typically supports at least two types of debug mode:

- Self-hosted debug: The processor itself hosts a debugger. Developer software and debugger software run on the same processor.
- External debug: The debugging occurs either on-chip, for example, in a second processor, or off-chip, for example, a JTAG debugger.

External debug mechanisms need to be access controlled to prevent abuse. External debug access must be controlled, based on:

- The requestor of the debug access

- The type of debug capability being requested
- The current security lifecycle state

The enforcement of these properties must be provided by an on-chip component, which is referred to as a debug protection mechanism (DPM). An SoC might include one or more DPMs. A DPM authenticates each requestor using one of the following methods:

- Secret-based authentication. For example, this may be a password. A device does not store the password itself, but a cryptographically strong hash of the password. The device also limits the number of attempts in a time window to prevent attackers from making many guesses.
- PKI-based authentication. A cryptographic token containing unlock information that is signed by a trusted authority. The device itself uses a public key to check if the signature is valid.

Regardless of the choice of authentication, the requirements in Table 24 Platform Security Requirements must be met.

Table 24: Debug requirements

PSR requirement	Description
	A DPM must be aware of the current security lifecycle state.
	A DPM must contain enough on-chip memory to securely identify debug requests.
	A DPM must be implemented either solely in hardware, or together with software running in the Trusted world.
	All external debug functionality must be protected by a DPM so that only an authorized external entity can access the debug functionality.
	A DPM unlock password or token must be at least 128 bits in length.

8.2 Authentication methods

The most straightforward solution to prevent unauthorized access to debug ports is to physically disable them in factory once the device is ready to move to a deployment state in its life cycle. This effectively removes the ability to achieve most of the above use cases once a device is deployed and is a radical way to defend against unwanted physical access.

Other protection methods will require some form of authentication before access into the device is granted through debug ports. Authentication methods vary in cost and in the level of protection they provide.

8.2.1 Secret-based authentication

Authentication methods based on the knowledge of a shared secret include passwords and symmetric keys. Time limits need to be set on symmetric-based authentication to thwart brute force attacks.

While those methods may provide a good level of simplicity for some use cases, they also have the following pitfalls:

- Possible leaks during authentication: An authentication protocol that provides a password or key in clear text over the line is open to spying attacks on the way. Implementations should rely on a mechanism to keep the password or key secret. For example, by using a challenge/response mechanism.
- Possible leaks during secret sharing: Passwords or keys need to be shared between device manufacturers and users who require access to those devices. No matter which procedure is used to deliver that secret to legitimate users, it must be transmitted and then stored, exposing an attack surface.
- Impact of leaked secrets: Using the same shared secret in all devices is a class key, where cracking a single device exposes all other devices of the same time. Shared keys should be avoided wherever possible. However, using a different secret for each device requires device vendors to manage a database of secrets, making this a service that needs to be available to deliver secrets to legitimate users. If the database is compromised, system security of all devices is lost.

Despite the potentially large attack surface, authentication based on shared secrets could be considered for use cases where the price of a Public Key Infrastructure (PKI) deployment cannot be justified. Examples include:

- Token distribution is limited to very few participants
- The number of devices is reasonably small
- The value to be gained by attacking the devices is not worth the time and effort

8.2.2 PKI-based authentication

Authentication methods based on asymmetric key pairs overcome most of the limitations related to shared secrets. Key pairs need to be generated and the public part included in certificates signed and tied to a small number of offline root Certification Authorities. The certificates need to be stored on the device. The private part must be kept secret, but is not stored on the device.

Compared to shared secrets, PKI deployments offer significant advantages:

- Signature validation is not based on the exchange of secrets, which removes any possible leak over the wire.
- Only non-confidential certificates are stored, there are no shared secrets. The only users capable of providing a valid token are those in possession of the private key corresponding to a

certificate that can be traced back to the root Certification Authority certificate deployed in all devices.

Deploying a PKI is not an easy task. Some vendors may want to rely on existing infrastructures that are used for secure boot or over-the-air firmware updates, or they might choose to contract specialized companies selling managed PKI services.

8.3 Interoperable authentication

To avoid leaking secrets during authentication, Arm recommends protocols based on challenge/response mechanisms. Arm recommends using the Authenticated Debug Access Control (ADAC) protocol. ADAC supports both secret-based and PKI-based authentication models.

ADAC uses debug tokens, which are signed data indicating debug access rights for the token holder. Access rights are encoded into a single 128-bit debug field indicating which capabilities should be unlocked.

Depending on the rights granted, a debug token may result in recoverable or unrecoverable debug. If unrecoverable debug is supported, then the token must be revocable.

8.4 Processor signals and control registers

The Armv8-A architecture describes the behavior of debug in detail:

- For information about self-hosted debug, see Armv8-A ARM Section D2
- For information about external debug, see Arm v8-A ARM Part H

The Armv8-A architecture also includes definitions for invasive and non-invasive debug:

- When using invasive debug, the act of debugging changes the processor state in a subtle way. For example, when a breakpoint is set using a debugger, the processor stops when the breakpoint is hit, and the timing of the program being debugged changes.
- Non-invasive debug does not change the processor state in any way. For example, generating and collecting trace data from a target will usually not affect the processor, so trace is usually classified as non-invasive debug. Other operations that are usually classified as non-invasive debug are use of the Performance Monitoring Unit (PMU) and performing Program Counter (PC) sampling.

From a security perspective there is no need to distinguish between these methods. This is because both types of debug would leak assets accessed by the PE.

Figure 7 shows a few DPMs, each of which controls a particular area. These areas overlap.

Figure 7: Debug protection mechanisms and scope

The Arm processor and CoreSight IPs include a control interface comprising of the signals in Table 25.

Table 25: Debug Control Signals

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

The CoreSight IP also has the input signal in Table 26.

Table 26: CoreSight Device Debug Enable

Signal	Name	Action
DEVICEEN	Device Debug Enable.	Enables the external debug tool connection to the device and drives the DBGSWENABLE input to the CoreSight components and A-profile PE.

The Arm processor also contains the S-EL2 SUNIDEN register that controls the debug functionality for Trusted user-space, see Table 27.

Table 27: Trusted world user space debug register

Signal	Name	Action
SUNIDEN	All non-secure and secure Unprivileged Non-invasive Debug Enable.	When asserted with DBGEN or NIDEN, SUNIDEN enables debugging of Trusted user-space code (but not of trusted privilege kernels). This is a register bit (not a wire) controlled by the trusted kernel

In many cases, Trusted world debug is necessary for initial system development but not necessary beyond that point in time. In this case, the secure debug control inputs might be set to use the programmed values in OTP fuse memory. For example, a production system may have a debug disable fuse set during the factory production line.

8.5 Other signals

Complex SoCs often include extra debug functionality beyond the application PEs. Examples of this are requesters on the interconnect, which can be controlled directly from an external debug interface, and system trace modules. Care must be taken to make sure that they are controlled by the correct DPM. They must be evaluated based on their access to assets that belong to each world and assigned the corresponding DPM.

8.6 Unlock operations

Arm recommends that each DPM has at least three states: locked, open and closed. On SoC reset, the DPM decides whether to lock or unlock debug. See Figure 8. For example, A DPM might be implemented as part of a Boot ROM. One variable, dpm_enable, is a programmable fuse that activates the DPM, with the fuse value:

- Permanently set to 1 in the factory for production devices

- Permanently set to 0 for development devices that are not intended to be used in secure deployments and do not need the DPM to be enabled

The variable dpm_lock is an ephemeral variable that controls whether debug is enabled for the current boot session, and lasts until reset.

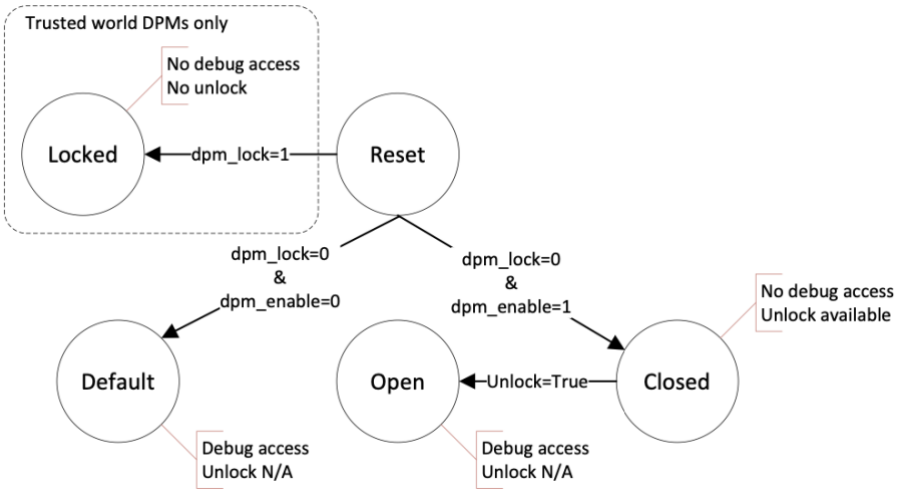

Figure 8: Debug port states

9 Peripherals and subsystems

The PSR requires the protection of sensitive assets from untrusted peripherals capable of Direct Memory Access (DMA), see Table 28. This may involve one or more mechanisms, for example one mechanism to isolate a device to one world and another mechanism to further isolate the device to certain memory regions within the world.

Table 28: Peripheral and subsystem requirements

PSR requirement	Description
	All DMA transactions from an external peripheral must be constrained using an on-chip mechanism.

9.1 Inter-world DMA protection

Peripherals need to be assigned to the world that they need to operate in. Restricting access to the world typically involves adding an appropriate filter between the peripheral and memory regions. Some filters might be programmable or fixed as part of the system design. A simple filter may permit access only to Trusted accesses. A more complex filter may permit access based on certain hardware IDs and may have different policies depending on whether the access is a Trusted access or Non-trusted access.

The A-profile processors in the system are TrustZone aware and send the correct security status with each bus access. However, most modern SoCs also contain non-processor bus requesters, for example, GPUs and DMA controllers. Like with completer devices, system requester devices can be divided into groups:

- TrustZone aware: Some requesters are TrustZone aware, and like the processor, provide the appropriate security information with each bus access.
- Non-TrustZone aware: Not all requesters are built with TrustZone awareness, particularly when reusing legacy IP. Such requesters typically provide no security information with its bus accesses, or always send the same value.

Requestors that are not TrustZone-aware may need to access to certain system resources, and could be set at design time or at runtime:

- SMMU: This is the most flexible option. For a Trusted requester, the SMMU behaves like the MMU in Secure state. This includes the NS bit in the translation table entries, controlling which physical address space is accessed. The SMMU is typically configured by the Operating System with the objective of giving the Non-PE Requester that acts for the PE process the same view of memory. See Figure 9. Arm recommends the use of SMMUs.
- Design time tie-off: If the requester only needs to access a single physical address space, a system designer might fix the address spaces to which it has access, by tying off the appropriate signal. This solution is simple but is not flexible.

- Configurable logic: Logic is provided to add the security information to the requester bus accesses. Some interconnects, like the Arm NIC-400, together with the TrustZone Protection Controller, provide registers that Secure software can use at boot time to set the security of an attached requester accesses. This overrides whatever value the requester provided itself. This approach still only allows the requester to access a single physical address space but is more flexible than a tie-off.

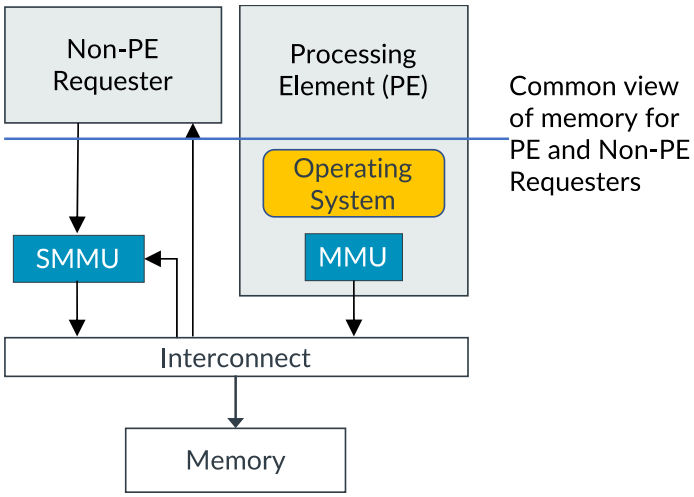

Figure 9 DMA protection with an SMMU

9.2 Intra-world DMA protection

An SMMU can be used in a virtualized system where multiple guest operating systems are managed by a hypervisor. An SMMU might be used to extend the memory protection and isolation scheme of applications, operating systems, and virtual machines in both the Secure and Non-secure state into non-CPU requesters. A DMA-capable IP and a software process may share a common view of memory, therefore retaining the isolation framework in which that software operates. This provides a convenient way for software to interact with DMA-capable IP, without creating security vulnerabilities where MMU protections and TrustZone separation can be bypassed.

If an attacker compromises the firmware of a non-PE requester, it might allow an attacker-controlled subsystem to access system memory during the boot process. The attacker could then interfere with the secure boot process or corrupt memory through time-of-check-time-of-use (TOCTOU) attacks against system firmware components. An SMMU can provide protection against DMA attacks by defaulting to a deny access policy. Firmware might selectively create SMMU mappings and enable DMA for specific subsystems needed to boot the system. To learn more, see the Arm System Memory Management Unit documentation.

SMMU_GBPA[ABORT] should reset to a value of 1 to implement a default deny policy on reset. If it is not possible to have SMMU_GBPA[ABORT] reset to a value of 1, firmware should configure

SMMU_GBPA[ABORT] to have a value of 1 as early as possible during the boot process, preferably in the Boot ROM of the SoC.

A system may include one or more SMMUs. Each SMMU includes a control interface, translation table walker, and Translation Lookaside Buffers containing cached translations. An SMMU may be embedded into an IP block or may sit standalone, as is illustrated by SMMU A and SMMU B in Figure 10. It may also be used in a distributed fashion; whereby multiple Translation Lookaside Buffers are coordinated by a single control interface and walker, as is illustrated by SMMU C in Figure 10.

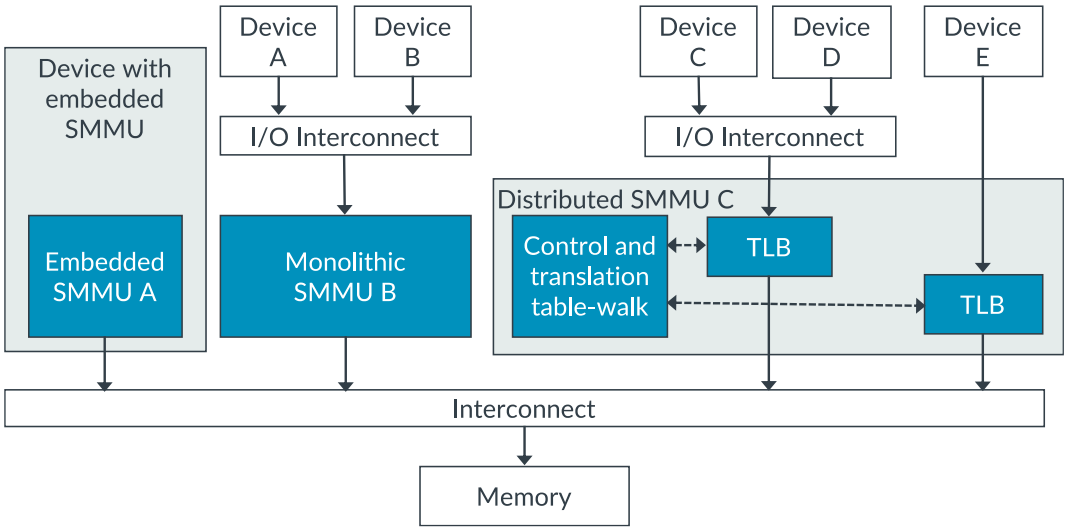

Figure 10 SMMU system implementations

9.3 External security subsystem pairing

External subsystems that hold or process Trusted world assets need to mitigate simple physical attacks, like external bus snooping or unauthorized replacement. Examples of an external subsystem holding Trusted assets typically include an off-chip Secure Element or a flash device. The types of attack are mitigated through the inseparability requirements listed in Table 29.

Table 29: External security subsystem inseparability requirements

PSR requirement	Description
	An off-chip security subsystem must be physically or logically inseparable from the host system. Separation must not reduce system security.
	Communication with an off-chip security subsystem must be protected against eavesdropping.
	Communication with an off-chip security subsystem must detect tampering and replay attacks

PSR requires external subsystems to be securely paired to the SoC, to prevent these subsystems being transferred to an untrusted device. Communication to and from the subsystem also needs to be protected. Pairing can be achieved either through physical means or with cryptography:

- Example of physical means: The external module is placed within the same physical package as the CPU during manufacturing. This assumes that an attack through opening the package and intercepting the inter-die connection is considered out of scope in the threat model. If this sort of attack is in scope, then the cryptographic approach below should be used instead.
- Example of cryptographic pairing: The SoC and the external subsystem are both provisioned with a shared secret, typically during device assembly. The SoC stores this shared secret in secure storage, for example, secure OTP memory, while the subsystem also stores the value internally. Requests and responses between the two components are authenticated using cryptography based on the use of a shared secret. Consider the following examples:
    - For a Trusted Platform Module 2.0, bus security is provided through two mechanisms:
        - Authenticity can be provided for each transaction using a Hash-based Message Authentication code (HMAC) session. An HMAC session uses a shared secret. A dedicated TPM monotonic counter that increases on each transaction provides replay protection.
        - Confidentiality is provided using a mode called parameter and response encryption. For more information about encrypted sessions, see the TPM 2.0 specification.
    - Some flash products have the ability to authenticate and decrypt requests. This is often derived from a shared secret that is provisioned to the host SoC and the flash product during factory provisioning

10 Platform identity

The PSR requires a unique, unclonable identity that is bound to the system hardware. The identity can be used to verify the authenticity of the system. An example use case is remote attestation, where a remote challenger authenticates the device identity in order to enforce security policies. Such policies can control whether a device has access to certain network services or secrets. Table 30 lists the requirements.

Table 30: Attestation key requirements

PSR requirement	Description
	The SoC must include an attestation key that is either held within secure storage controlled by the Trusted world or held within a Security subsystem.
	The attestation key must be unique for each instance or for each group of devices

In a scenario where device owner privacy is required, the attestation key should not be unique for each device. To provide some anonymity, groups of devices can share the same attestation key. This is known as Group Unique Key (GUK). However, to reduce the impact of an exposed GUK, the size of the group should not be too large. The number of devices for each group is likely to be dictated by what is practical during factory provisioning.

The attestation key must use asymmetric cryptography. The public key needs to be published to those parties that rely on the public key to verify any attestation claims signed with the private key. The private key must be held securely on the device, there are at least three possible options that can meet the requirements listed in Table 31:

- An on-chip private key that is hardware unique and protected by the Trusted world or hardware enforced security
- A private key that is held in off-chip encrypted storage
- A private key held in an external security subsystem, like a Secure Element (SE) or Trusted Platform Module (TPM). This option can provide unique hardware-bound identity. These external subsystems, however, must be paired to each SoC during the device manufacturing stage. See Section 9.3 External security subsystem pairing for more about the pairing requirements.

The advantages and disadvantages of these approaches relate to the operational costs of provisioning. The advantage of an SE is that the provisioning of root secrets can be handled by the SE manufacturer and not the SoC maker or OEM, which might mitigate the risks and costs of in-house secret provisioning. However, an SE may add prohibitive costs to the bill of materials depending on the product, and hence all approaches require careful consideration.

Table 31: Attestation key access requirements

PSR requirement	Description
	In an implementation that uses a Security subsystem for cryptographic identities, the attestation key must only be visible to the Security subsystem.
	The attestation key must be protected by a security lifecycle.

It should be noted that when an external TPM is used it, it is susceptible to a reset attack, where a physical attacker can power-cycle the TPM. This causes a TPM to lose all recorded measurements for the current boot session and an attacker can then forge TPM measurements to a state of their choosing. This means that attestation with an external TPM is not suitable for threat models that require protection against adversaries who can perform non-invasive physical attacks.

11 Random number generation

Random numbers are expected to be collected from a hardware block that is accessible by all PEs, typically through separate FIFO queues. There are at least three considerations to consider in an implementation:

- The PEs must not be able to access the raw entropy source directly. Post-processing is often deployed to ensure a more statistically random distribution of the numbers.
- The chip designer must ensure that any two PEs cannot access the same value.
- The threat model should consider whether it is acceptable for one world to starve the entropy of another world, and whether the system should isolate those resources between worlds, using hardware or software mechanisms.

An external subsystem, like a Secure Element or a Trusted Platform Module, can also provide a source of random numbers, provided the connection meets the requirements described in Section 9 Peripherals and subsystems.

12 Timers

Recommendations will be included in a future update to this document.

13 Cryptography

13.1 Algorithms

Cryptographic algorithm choice must depend on factors such as industry, regional, and government standards. Currently Arm recommends at least 128-bit security, see Table 32, though this does not necessarily mean 128-bit keys.

Table 32: Cryptographic strength requirements

PSR requirement	Description
	All use of cryptography must use an algorithm that meets at least 128 bits of security.

Arm recommends using approved algorithms from the Commercial National Security Algorithm Suite, which supersedes NSA Suite B Cryptography. Alternatively, designers should refer to the approved cryptographic algorithm lists that SOG-IS, IPA, and CC have published for the EU, Japan, and China.

Predictions suggest that quantum computers will be able to break some cryptographic algorithms. The estimates for when this will be a problem vary. For devices with potentially long lifespans, Arm recommends that product designers read the Post-Quantum Cryptography white paper from Arm. Stateful hash-based signatures, as described by NIST SP 800-208, are recommended at this current time.

13.2 Side channel resilience

Implementations of cryptographic algorithms, whether in hardware or software, can be vulnerable to whole classes of timed attacks, power analysis and fault injection. Due to the complexity of these topics and the significant investment needed for mitigation, Arm recommends using cryptographic hardware that is certified by an independent certification scheme. For example, a secure element or an on-chip secure enclave IP that has been evaluated or at least meets some pre-certification criteria. The expected certifications shall depend on what the target market recognises.

End products that do not require mitigations against physical attacks should still attempt to address timing attacks if algorithm time can be measured by untrusted software or untrusted adversaries over the network.

13.3 Key operations

The requirements listed in Table 33 need to be considered when creating hardware and software that uses, creates or modifies any key. General guidance cannot be easily provided.

Table 33: Key operation requirements

PSR requirement	Description
	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.
	Any operations on a key must be atomic. It must not be possible to interrupt the creation, update, or destruction of a key.
	A key must only be used by the cryptographic scheme for which it was created.

13.4 Hardware acceleration

If large amounts of data must be processed, then the cryptographic algorithms may need to be accelerated in hardware. Accelerators for symmetric and hashing algorithms are common. The Armv8-A architecture defines cryptography extensions for these operations such that any software can take advantage of available accelerators.

Asymmetric algorithms, however, are complex and can consume a significant amount of die area. A common trade-off is to accelerate only the most compute-intensive parts. For example, big integer modulo arithmetic is an intense part of the RSA algorithm.

13.5 Hardware key stores

A key store holds cryptographic keys and their associated metadata. A hardware key store is a hardware implementation of a key store, whereby the keys are not visible to any software running on the system. The keys might have been injected through a Trusted 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 operations are 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. Accelerators can typically be used by both the Trusted and Non-Trusted worlds, by implementing separate 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.

Figure 11 shows an example architecture for symmetric algorithm acceleration and an associated secure key store.

The accelerators and the Key Store are peripherals within the SoC and must meet the associated requirements as described in Section 9 Peripherals and subsystems.

The Arm CryptoCell family of IP implements a hardware key store and provides independent interfaces and operations for Trusted and Non-trusted worlds.

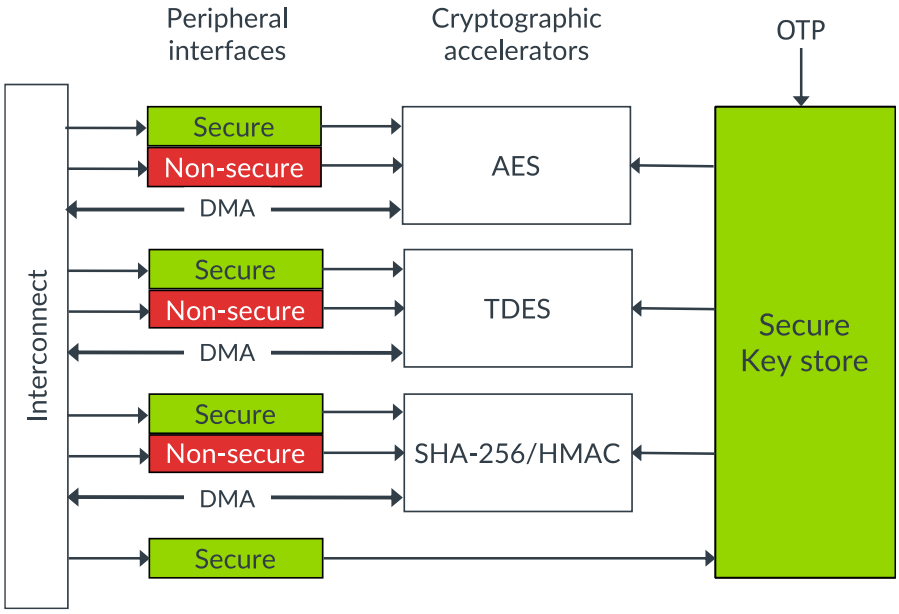

Figure 11: Example architecture of a cryptographic accelerator

14 Secure storage

Trusted world services need to be able to store assets in a secure manner. Examples of important assets that should be stored in secure storage:

- Cryptographic keys
- Certificates
- Firmware settings
- Trusted application data

Secure storage requirements are listed in Table 34.

Table 34: Secure storage requirements

PSR requirement	Description
	Any sensitive data that needs to be stored must be stored in Secure storage.

Secure storage should offer confidentiality, integrity, and freshness to stored items. While all assets benefit from these properties, some assets may not be secret enough to require confidentiality or may not benefit from freshness.

14.1 Confidentiality

An SoC must contain an on-chip Hardware Unique Key (HUK) that is only accessible by the Trusted world. The HUK is used to encrypt and decrypt the contents of off-chip storage, providing confidentiality. It is also required to be unique to the SoC instance, which prevents attacks from cloning assets to other instances. Table 35 lists the HUK requirements.

Table 35: Hardware unique key requirements

PSR requirement	Description
	The SoC must include a hardware unique key (HUK).
	The HUK must have at least 128 bits of entropy.
	The HUK must only be accessible by Trusted code or Trusted hardware that act on behalf of Trusted code.

The HUK might be generated differently depending on supply chain factors and risks:

- During the factory provisioning state of the security lifecycle, the HUK can be generated on a factory line workstation and then installed into the SoC OTP memory using factory provisioning

software. This can help speed up factory lines at the risk of exposing keys on the factory line. While Arm does not recommend this approach, it may be the only viable option for some production lines.

- During the factory provisioning state of the security lifecycle, the HUK is generated by the SoC using a True Random Number Generator (TRNG). The TRNG may be driven exclusively by hardware components or using provisioned software. The time to generate the key is dominated by the time it takes for the SoC to establish enough entropy, which may vary from chip to chip.
- Physically Unclonable Functions (PUF) may be embedded in the SoC design, so that a statistically unique HUK can always be derived from physical variations in each chip instance. Care must be taken to ensure that the PUF design meets the requirement of 128 bits of entropy.

14.2 Integrity

If storage needs to be tamper-resistant against Non-trusted software, then the storage device should be managed by a Trusted service. However, if storage only needs to be tamper-evident against physical attacks, then the storage device can be managed by the non-secure domain.

14.3 Freshness

Confidentiality and integrity properties do not provide complete security. This is because a physical attacker can always replace encrypted content in external storage with older content. This is known as a replay attack. Table 36 lists the data freshness requirements.

Table 36: Freshness requirements

PSR requirement	Description
	An on-chip non-volatile Trusted firmware version counter must be included
	An on-chip non-volatile Non-trusted firmware version counter must be included
	It must only be possible to increment a version counter through a Trusted access.
	It must only be possible to increment a version counter; it must not be possible to decrement it.
	When a version counter reaches its maximum value, it must not roll over, and no further changes must be possible.
	A version counter must be non-volatile, and the stored value must survive a power down period up to the lifetime of the system.
	When a version counter reaches its maximum value, it must not roll over, and no further changes must be possible.

To provide a complete secure storage solution, counters are required to prevent replay attacks. The implementation choice of counter is largely a matter of cost and depends on expected product usage. Several strategies are described here:

- Non-volatile counters can be created using a set of individual on-chip OTP memory bits, which are blown individually over time. This solution is the only realistic counter that can be implemented on-chip when chip fabrication uses process nodes that cannot support multi-time programmable NVM. The die space requirements of OTP NVM technology like e-Fuses or anti-fuses may limit the number of fuses that can be provided. This option is recommended for storage items that are expected to be updated infrequently, like firmware images or root certificates. This form of counter is essential for early secure boot firmware, where external subsystems may not be available.
- An eMMC device can provide replay-protected external storage to the SoC in the form of a Replay Protected Memory Block (RPMB). An RPMB is a special partition in an eMMC device, where all writes to the partition must be authenticated. Authentication is performed using a shared secret that is known to both the Trusted world and the eMMC device. The RPMB also contains a read-only counter that self-increments after every write message. Every authenticated write message must include the latest counter value otherwise the write request is rejected. This prevents replay attacks against data in the RPMB partition. Where SoCs have no multi-time programmable NVM, it is expected that the key used to authenticate write requests is derived from an immutable on-chip secret key. This immutable on-chip key is programmed exactly once in a factory, usually on the production line. The shared secret is also programmed into the eMMC device and then placed into a mode called Secured.

  For more information, see the eMMC 5.1 specification. An eMMC device can provide a larger number of writes compared to the alternative strategies described in this section. Similar schemes may also be available for other types of flash memory.
- An external Secure Element (SE), or a Trusted Platform Module (TPM), can provide monotonic counters to the SoC. Like an eMMC device, a secure element must be securely paired to the SoC for the lifetime of the device. Because the bus to an external secure element is often exposed to attackers, a secure session must always be used between the host and the secure element. A secure session typically deploys an HMAC session based on a shared secret that is known to both the SoC and the secure element. The host SoC must have enough secure storage to host the shared secret. Monotonic counters cannot be reset, which might be too restrictive for some products. TPM NVRAM banks may be used to provide counter values instead, provided that the banks are access controlled.
- Embedding an MTP storage die or SE within the same package as the SoC

Many software components typically want secure storage. Software components include Trusted services, a hypervisor, virtual machines, and applications. A suitable implementation might employ one counter for each software instance, or group together a list of version numbers inside a database file, which is itself versioned using a single count.

For less critical data, the following options can also be explored:

- Using a battery-backed hardware up counter
- Using a hardware up counter in an always-on domain on a remote trusted server with a Trusted service mechanism that can restore the counter value after a cold boot if power is removed. In this case, it is acceptable to limit availability of services until the version count is restored.

In both cases, a Trusted service is responsible for cryptographically pairing the external reference, and for appropriately updating an internal hardware counter.

15 Main memory

Like external non-volatile storage, external RAM may require similar forms of protection. This is because it may be easier for some attackers to read or modify its contents. The level of protection needed depends on the product security requirements. Memory protection activation and deactivation requirements are given in Table 37.

Table 37: Main memory requirements

PSR requirement	Description
	The activation and deactivation of external memory protection must only be possible from the Trusted world.

15.1 Isolation

When the SoC is configuring various security controls, the SoC should not be dependent on any external RAM. Instead, on-chip secure RAM must be used. Table 38 lists the requirements. Because the Secure RAM is used for secure boot and certain secure operations, it must be on-chip and accessible only to the Trusted world. The TZ Memory Adaptor acts as a single region address space controller for on-chip memory which needs to be access only in secure mode.

Table 38: Secure RAM requirements

PSR requirement	Description
	The SoC must integrate a Secure RAM.
	Secure RAM must be mapped into the Trusted world only.
	If the mapping of Secure RAM into regions is programmable, then configuration of the regions must only be possible from the Trusted world.

The amount of Secure RAM to include in an SoC depends on product requirements. SoC designers should budget for the memory needed by Trusted firmware. Depending on protection requirements an SoC designer may also budget for the memory needed by a Secure Partition Manager and/or Trusted OS instances.

Larger Trusted world software may be offloaded to external RAM provided that the memory system is aware of world isolation. The Arm CoreLink TZC-400 TrustZone Address Space Controller is an example of a security IP that protects multiple regions of external memory by configuring certain regions to be restricted to Trusted or Non-trusted accesses.

15.2 Confidentiality

When assets are held in external memory, they are suspectable to snooping or cloning by attackers with physical access. For example:

- DRAM tends to retain its contents for a limited amount of time after it is unpowered. The time frame can range from several seconds to minutes. An attacker with physical access may exploit this property to perform what is known as a cold boot attack. To conduct the attack, the attacker must initiate a cold reset of the SoC once the target assets are in memory. Afterwards, an attacker can then either:
    - Boot an alternative operating system or software program, for example from UART or USB, in order to dump the contents of RAM to a file. The file can then be analyzed.
    - Freeze the DRAM chips using a can of freeze spray, liquid nitrogen, or equivalent, in order to preserve the memory content. The attacker can then safely remove the DRAM modules and transport it to another system without losing data.
- NVRAM that is used as main memory does not lose its content, and is easy for a physical attacker to access. In some cases, an attacker may be able to substitute DRAM for NVRAM in order to persistently capture contents. Alternatively, attackers may use battery-backed DRAM to achieve the same result. Once the attacker is aware that assets are in memory, they can remove the NVRAM modules and analyze the contents.
- DDR main memory interposers, or equivalent, can be attached to the memory bus in order to monitor reads and writes to main memory.

Other variants of the above threats also exist. It is important to consider whether a product market requires protection against these attacks, considering the likelihood and impact.

To prevent important assets from leaking, they may need to be encrypted when leaving the SoC boundary. Table 39 lists the requirements. When there are a very small number of assets, encryption using software techniques may suffice, but this has limited practicality. A typical system may have many assets belonging to different entities and may be loaded in unpredictable memory locations. In this case, a hardware-based memory encryption system can transparently ensure that all external memory writes are encrypted, and all external memory reads are decrypted.

To prevent an attacker from stealing encrypted memory and decrypting them on another platform, the key used to decrypt and encrypt memory needs to be unique for each SoC instance.

Table 39: Memory encryption and key requirements

PSR requirement	Description
	Keys used by a memory protection block must be unique to the SoC.
	If the mapping of cryptographic hardware into the memory system is configurable, then it must only be possible to perform the configuration from the Trusted world.

To prevent an attacker from performing a cold boot attack using software techniques, the key used to encrypt and decrypt external memory must change at least once on every system reset. Systems with long uptimes may choose to change keys periodically within the same boot session.

It is common for memory maps to include aliased regions. When a hardware memory encryption system is used to encrypt external memory, care must be taken to ensure that all aliases pointing to external memory area are always subject to the same encryption and decryption. Table 40 lists the requirements.

Table 40: Memory aliasing and encryption requirements

PSR requirement	Description
	If a memory region is configured for encryption, then there must not exist any alias in the memory system that can be used to bypass security.

Further recommendations will be included in a future update to this document.

15.3 Integrity

This includes software attacks like Rowhammer attacks and physical attacks, like malicious DRAM modules. Recommendations will be included in a future update to this document.

15.4 Freshness

This includes physical attacks, for example memory bus interposers or programmable DIMMs. Recommendations will be included in a future update to this document.