Platform Security Requirements 1.0

Document number:	DEN 0106
Release Quality:	Alpha
Issue Number:	0
Confidentiality:	Non-Confidential
Date of Issue:	09/10/20

Contents

About this document

Release Information

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

Date	Version	Confidentiality	Change
Oct 2020	ALP-0	Non-confidential	First alpha-quality release.

Platform Security Requirements

Arm Non-Confidential Document Licence ("Licence")

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 documents.

Ref	Document Number	Title
[1]	DEN 0079	PSA Security Model
[2]	DEN 0063	PSA Firmware Framework for M
[3]	SP 800-90B	NIST Special Publication 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation
[4]	SP 800-90C	NIST Special Publication 800-90C Recommendation for Random Bit Generator (RBG) Constructions https://csrc.nist.gov/publications/detail/sp/800-90c/draft
[5]	DEN 0077A	Firmware Framework for Arm® v8-A

Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
Cryptographic hash	A one-way function which maps data of arbitrary size to a bit string of fixed size.
DPM	Debug Protection Mechanism
HUK	Hardware Unique Key
MTP	Multi-time programmable
NVM	Non-volatile memory
OTP	One-time programmable
PE	Processing Element
PSA	Platform Security Architecture
ROTPK	Root of Trust Public Key; also known as a Boot Validation Key.
SoC	System-on-Chip
TRTC	Trusted Real Time Clock

Potential for change

The contents of this specification are subject to change.

Conventions

Typographical 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 such as 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

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.

Rule identifiers are introduced when the document reaches REL quality.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title (Platform Security Requirements).

- The number and issue (DEN 0106 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 document specifies the bare-minimum security requirements expected of System-on-Chips (SoC) across multiple markets. It is primarily intended for chipset designers who require compliance with various security requirements. Architects, designers, and verification engineers can use this specification to support the process of certification with independent laboratories.

This document does not specify a specific system architecture or the use of specific components. Other documentation from Arm provides guidance on how to best meet the security requirements using the Arm architecture and system IP. System designers are encouraged to check conformance to the specified security requirements.

2 Security goals

The *PSA Security Model* outlines the important principles of a secure system in the form of *security goals*. These goals are embodied in various Arm specifications and are used as the basis for developing the detailed hardware security requirements within this document.

2.1 Cryptographic identity

To securely communicate with a specific system, each system must be uniquely identifiable.

2.2 Security lifecycle

A system must ensure that the protection of assets and the availability of device functions follow a prescribed and constrained path from manufacture to device disposal. Therefore, the system must have a state machine that it can use to make appropriate security decisions within a particular context. This is known as a *security lifecycle*.

Each *security state* in the *security lifecycle* defines the security properties of the system. The security state depends on software measurements, hardware configuration, debug mode, and the lifecycle phase. For example, a lifecycle state can cover scenarios such as development, provisioning, deployment, returns, and end-of-life.

2.3 Attestation

A system must be able to provide evidence of its trustworthiness to relying parties, which requires the identity and security state of the system to be proven through attestation. To have validity, the system must be part of a governance program. Such a program includes roles such as evaluation labs, attestation verifiers, and relying parties.

2.4 Secure boot

Secure boot and secure loading processes are necessary to prevent unauthorized software from being executed.

2.5 Secure update

Mechanisms are needed to update the system firmware when a new feature is introduced or when a vulnerability is fixed. The update process itself must be secure against abuse.

2.6 Rollback protection

Previous versions of software can contain exploitable vulnerabilities as time passes. Therefore, preventing rollback of firmware to a previous version is essential; however, rollback for recovery purposes might be permitted if authorized. Furthermore, configuration data might need versioning and rollback protection.

2.7 Security by isolation

Isolation of trusted services and resources from less trusted services is essential. Isolation aims to prevent one service from compromising the assets of other services. It is probable that software contains flaws that can be exploited to compromise the security of a system.

This document uses the term *Trusted world* to refer to hardware resources whose state supports the trusted services and the term *Non-trusted world* to refer to hardware resources whose state supports the untrusted services.
An example software architecture that uses security by isolation is the PSA Firmware Framework [2][5].

2.8 Secure interfaces

Interaction over isolation boundaries is essential if isolated services serve a purpose. However, the interfaces must ensure that they cannot be used to compromise the system. It might also be necessary to ensure the confidentiality and integrity of any data exchanged over such interfaces.

2.9 Binding

Sensitive data, for example, user or service credentials, and secret keys, must be bound to a system to prevent cloning and disclosure outside of the system owning the data. This might require confidentiality and integrity assurance if stored in non-private storage. Cryptographic keys are typically used to secure sensitive data, which are themselves system-sensitive data and must be bound to the system. It might also be necessary to bind the data to a security lifecycle state.

2.10 Trusted services

Trusted services must ensure that other goals are met. Trusted services might include configuration of the hardware to support security lifecycle, isolation, and cryptographic services that might use bound secrets to support attestation, secure boot, secure loading, as well as binding of data.

3 Scope

The following class of threats are covered in this document:

Threat	Summary
T.ROGUE_CODE	An attacker succeeds in loading and executing rogue code on the device in order to obtain assets or escalate privileges.
T.TAMPERING	An attacker replaces or tampers with off-chip storage, memory or peripherals in order to obtain assets or escalate privileges.

T.CLONING	An attacker with physical access reads data in off-chip storage or memory. This enables reverse engineering or cloning of assets to other systems.
T.DEBUG_ABUSE	An attacker succeeds in accessing debug features in order to illegally modify system behavior or access assets.
T.WEAK_CRYPTO	An attacker breaks the cryptography used by the device in order to access assets or impersonate the device. This threat only relates to algorithm strength, key size, and random number generation.
T.IMPERSONATION	An attacker pretends to be the device in order to intercept assets that are provisioned to the device.
T.POWER_ABUSE	An attacker abuses power management controls using software in order to access assets.
T.SOFT_SIDE_CHANNELS	An attacker uses software-observable side channels to infer information about assets.

The following threats are out of the scope of this document, some of which might need to be mitigated for specific markets or certification levels:

Threat	**Summary**
T.INVASIVE_ATTACK	An attacker uses invasive techniques, in which systems are physically unpackaged and probed, in order to recover assets.
T.GLITCHING	A physically present attacker uses power, clock, temperature, and energy glitch attacks that cause faults such as instruction skipping, malformed data in reads/writes, or instruction decoding errors.
T.PHYS_SIDE_CHANNELS	An attacker infers the value of sensitive on-chip code or data by using physical non-invasive techniques, such as differential power analysis or timing attacks. An example asset can be a cryptographic key.
T.DENIAL_OF_SERVICE	An attacker damages an asset or prevents an asset from being accessed.
T.SUPPLY_CHAIN	While the guidance in this document provides mitigation against potential attacks in a supply chain, such as firmware tampering, it does not directly address supply chain security.
T.APPLICATIONS	Threats to the Non-trusted world and general application security.

4 Compliance

To be compliant with this document, there must be evidence-backed documentation that shows the design meets all applicable requirements that this document describes. Typically, the design team achieves this through verified output of a design review of the system. Arm recommends that this assessment is conducted as part of a Secure Development Lifecycle.

Normative requirements are described within tables. Text outside of the tables is informative.

The design team must provide evidence of fulfillment for each requirement. This confirmation must include justification for the compliance in the form of a brief outline, and references to the relevant detailed specifications. In general, requirements might not be applicable if the threats that they mitigate can be shown to not form part of the threat model of the system, or that any vulnerabilities that might result from not meeting a requirement can be demonstrated to be mitigated in another way. In some cases, it is necessary to provide more robust security. In these cases, supporting evidence must be documented alongside the requirement.

In several areas, this document provides recommendations. Where possible, these recommendations are provided to give guidance on reasonable default design choices. The threat model and functional requirements of the system is key in determining how requirements are met and the recommendations to follow. The development of a product threat model is beyond the scope of this document.

5 Security requirements

At an abstract level, a system is comprised of a collection of assets, alongside operations that act on those assets. In this context, an asset is defined as code or data that has an owner and an intrinsic value; for example, a monetary value. All data sets are assets that are associated with a value, even if that value is zero. A data set can be any stored or processed information.

High-value assets that require protection belong to the Trusted world, while low-value assets that do not require strong protection should belong to the Non-trusted world. The classification, ranking, and mapping of assets to worlds depends on a product's requirements, and is beyond the scope of this document.

This section describes the security requirements that an SOC meets. The requirements are described in tables and are distinct from the supporting text. The text provides additional context to ease comprehension of the rationale for each requirement.

5.1 Security Lifecycle

During its creation and use, the system progresses through a series of states. These states indicate the assets present in the system and the functionality that is available or has been disabled. Progression through the states is usually controlled using a write-once mechanism. The figure contains the minimum number of states and it is expected that a full product will contain more that are specific to a product family, OEM manufacturing, or market requirements. An example of a security lifecycle is show in Figure 1.

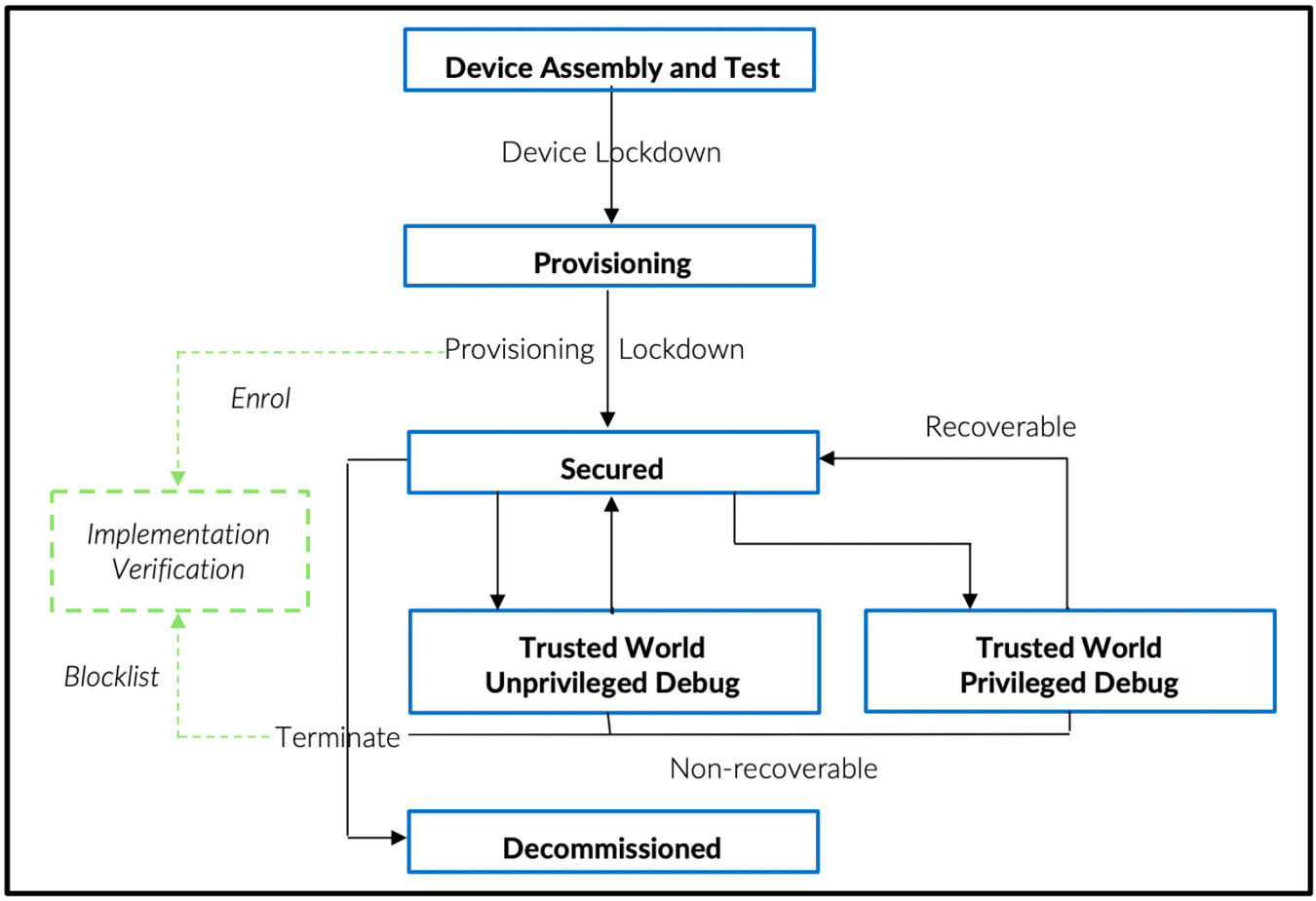

Figure 1: Example of a security lifecycle

As a minimum, a system compliant with this document provides a lifecycle control mechanism in which:

- The lifecycle state is held in, or derivable from, protected or one-time programmable memory.
- All lifecycle state transitions are restricted to a designated set that includes at least:
  - A designated initial state from which all systems start.
  - A designated deployed state which mandates the use of the system's security features.
  - A designated terminal state (Decommissioned) from which no further transitions are permitted. This is also known as *Return Merchandise Authorization* (RMA).

- A transition into the Decommissioned state should atomically zeroize, or otherwise put beyond use, all secret keys in the manner described by FIPS 140-2. The transition must also be authorized by the RoT owner in order to prevent an attacker from erasing important secrets.
- Some systems might have enough hardware capability to hide root secrets during an invasive debug state such as Trusted World Privileged Debug. This makes it possible for the system to go back to a 'secured' state. This is represented by the "Recoverable" transition in the Figure 1.
- Booting, debugging, and scan access are governed by a secure lifecycle policy.

	The system must enforce a security lifecycle.
	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.
	A transition into the terminal state must be authorized by the owner of the security lifecycle.

It should be noted that the system can also contain other lifecycles that are specific to a market, application or supply chain. These lifecycles are orthogonal and complimentary to the security lifecycle described here.

5.2 Boot ROM and reset

The secure configuration of a system depends on Trusted software that in turn forms part of a chain of trust that begins with the secure boot of the SoC. Secure boot ensures the integrity of firmware and critical data by detecting tampering or unauthorized changes. Secure boot ensures the integrity of firmware and critical data by detecting tampering or unauthorized changes.

Secure boot is based on an immutable secure boot image. It is the first code to run on the processor core and it is responsible for verifying and launching the next stage of boot. The secure boot image must be fixed within the SoC at manufacture time and is stored in an embedded ROM. This ROM is referred to as the Boot ROM.

Boot ROMs are typically implemented as either mask ROM, or by embedded flash with hardware support to ensure that, once programmed, it cannot be subsequently altered. The Boot ROM contains the boot vectors for the main processing elements as well as the secure boot image. Typically, the boot loader is divided into several stages, the first of which is the Boot ROM. Later stages will be loaded from non-volatile storage into Secure RAM and executed there. In this document, the second stage boot loader is referred to as Secure Boot Firmware.

5.2.1 Boot keys

The Secure Boot Firmware must be authenticated by the Boot ROM using an on-chip public key, which is here referred to as the *Root of Trust Public Key* (ROTPK)[1]. The specific public key algorithm used for authentication is defined by the implementation but subject to the security requirements defined in section 5.13 Cryptography.

To minimize the amount of required on-chip memory, an SoC should only store a cryptographic hash of the public key, while the public key is held in external storage. On each boot, the Boot ROM can then calculate the hash of the loaded public key and compare it with the hash in on-chip memory to ensure it has not been tampered.

Further details on the secure boot sequence and authentication mechanisms can be found in the *Trusted Boot and Firmware Update* document, and in the reference implementation provided by the *Trusted Firmware* project.

[1] Some markets recommend encrypting all data in external storage, including public keys, to reduce the amount of information exposed to potential attackers.

	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 ROTPK, 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.

5.2.2 Boot types

A cold boot is a boot that is not based on a previous system state. Normally, a cold boot only occurs when the platform is powered up or a hard reset signal is generated by a power-on reset circuit. A hard reset that triggers a cold boot might also be available in case of a software lock-up.

A warm boot is a boot that is based on a previous system state to facilitate a more rapid activation of the system than might be possible with a cold boot. Where a warm reboot is required, it is necessary to deploy some method to signal the use of stored state, examples include:

- The Boot ROM can distinguish between a warm boot and a cold boot via a status register.
- The SoC can use an alternate reset vector for a warm boot, causing the Boot ROM to execute warm boot specific code.

Typically, any storage needed to support these mechanisms is implemented within an always-on power domain.

In the case of a cold boot, the Boot ROM behavior might be entirely fixed in the implementation. However, it can also be influenced by additional configuration information, such as fuse values.

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

	If the system supports warm boot, a flag or register that survives warm boot must exist to enable distinguishing between warm and cold boots. This register or flag must be programmable only by the Trusted world and must be reset after both a cold and a warm boot.
	Where a flag or register that survives warm boot exists to distinguish between cold and warm boot, it 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, then it must be accessible only by the Trusted world.

5.2.3 Boot parameters

Some Boot ROM implementations can be influenced by additional configuration information stored in on-chip one-time programmable memory (OTP). An example of OTP memory are fuses. Configuration information can be used for the following purposes:

- Selection of the device containing the first loadable firmware image.
- Storage of the ROTPK.
- Storage of a root key for boot image decryption.
- Storage of other boot specific parameters.

The effect of these parameters on Boot ROM behavior need to be carefully considered for each state in the security lifecycle. Some parameter values might need to be disallowed depending on the state of the security lifecycle. For example, some factory test parameters are expected to be disabled once the provisioning lifecycle state is reached.

	The Boot ROM must be aware of the current security lifecycle state.
	It must not be possible to boot from any other storage device unless a Trusted Debug mode permits this. For detailed information about debug, see Debug.
	Any Boot ROM configuration outside of on-chip OTP memory must be authenticated using an on-chip public key.

5.2.4 Boot robustness

The execution of the Boot ROM must not be perturbed by other agents (for example, through DMA) or via interfaces (for example, PCI, JTAG). Therefore, external interfaces and direct memory access must be appropriately restricted when a cold reset occurs. This prevents secure boot from being bypassed on reset.

In many cases, DMA and various interfaces can be re-enabled by boot software when it is safe to do so, for example after secure boot and security protections have been configured. More information about debug modes is described in the Debug section.

	The Boot ROM execution state must be protected from DMA reads and writes.
	External interfaces must be disabled on each cold reset.

Arm recommends disabling DMA for all masters on reset. This is a simple way to ensure compliance with the above rule. However, it is recognized that this might be too restrictive for some system designs.

Careful analysis of Boot ROM code is essential because a vulnerability can undermine the entire system security.

The Boot ROM contains sensitive code that verifies and optionally decrypts the next stage of the boot. For some devices, if an attacker were able to read and disassemble the ROM image, they could gain valuable information that could be used to target an attack that circumvents the verification mechanism. For example, timing information can be used to target a fault injection attack.

Contingent on the threat model, it might aid robustness if the Boot ROM code and data is accessible only during boot. Device designers should consider implementing a non-reversible mechanism which prevents access by, for example, hiding the ROM using a sticky register bit that is activated by the boot software.

5.2.5 Secondary processing elements

If the SoC implements multiple processing elements (PE), the designated boot PE is called the primary. After the de-assertion of a reset, the primary boot PE executes the Boot ROM code, and the remaining PEs are held in reset or a safe platform-specific state until the primary boot PE initializes and boots them. There are at least a few possible examples:

- The platform power controller can hold all secondary PEs in a reset state, while the primary boot PE executes the Boot ROM until it requests for the secondary PEs to be released.
- All PEs execute from the generic boot vector in the Boot ROM after a cold boot. However, the Boot ROM identifies the primary boot PE and permits it to boot using the secure boot image, while the secondary PEs are made inactive.
- An on-chip security subsystem executes its own private Boot ROM when a system reset occurs, and then releases the application processor once completed. The definition of a security subsystem is defined in *Security subsystems*.

	All secondary PEs must remain inactive until permitted to boot by the PE.

5.3 Clock and power

Platforms with a high degree of power control might integrate an advanced power management subsystem using dedicated hardware, which executes a small software stack from local RAM. In such cases, the management subsystem has control over some Trusted assets, for example:

- Reset examples include:
  - State machines that sequence the assertion and de-assertion of resets in relation to the reset hierarchy, the system clocks and any power states.
  - Re-synchronization of resets at clock boundaries
- Clock generation and selection examples include:
  - Registers to enable or disable clocks
  - Registers that manage clock glitch and/or frequency detectors.
  - Configuration of Phase-locked loops or other clock sources.
  - Clock dividers and other glitch-less clock switching and clock gating mechanism.
- Power control examples include:
  - Access to power controllers, switches, or regulators.
  - State machines for sequencing when changing power states.
  - Logic or processing to intelligently apply power states either on request, or dynamically.
- State saving and restoration. To dynamically apply power states, some subsystems can also perform saving and restoration of system states without the involvement of the main application processor.

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

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

If the system can be suspended, various system state might need to be stored in off-chip storage. To prevent an attacker from modifying or reading state in external storage, the state must be protected using authenticated encryption.

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

5.4 Memory system

This document describes a hardware infrastructure that provides strong isolation between the operations and assets of the Trusted and Non-Trusted worlds. Operations and assets are connected by transactions, in which a transaction represents read or write access to storage containing the asset. As described in the Security Goals, the system sees the memory map as into two spaces, Secure and Non-secure storage, in which Trusted world assets are held in Secure storage and Non-Trusted world assets are held in Non-secure storage. Each transaction originates from either the Trusted world or Non-Trusted world.

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

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

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

It is dangerous for a Trusted operation to execute code belonging to the Non-trusted world, and therefore should not be permitted. This can be enforced using one of the following methods:

- Disabling instruction fetches from the Non-trusted world into the Trusted world. This can be fixed in hardware or configurable by Trusted firmware.
- Careful code review of Trusted operations to ensure secure transactions never fetch instructions from Non-secure memory.

	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.
	A Non-secure transaction must only access Non-secure memory.
	A Secure transaction must not fetch Non-secure instructions

Designs that use a network-on-chip interconnect might be able 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 must only be possible from the Trusted world using Secure transactions.

It is possible to have world-aware peripherals, in which the peripheral is visible in both Trusted world and Non-Trusted world address aliases at the same time. It is also possible for that peripheral to use signals to determine if the access is from the Trusted world, using a Trusted address alias, or from the Non-Trusted world, using a Non-Trusted address alias. This arrangement does not use filters, but the Secure aliases of the peripheral address space must be in a non-executable area of memory.

	If programmable address remapping logic is implemented in the interconnect, then 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.
	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.

Assets from different worlds can at different times occupy the same physical volatile storage locations. This is called shared storage. The underlying storage can be volatile, for example, on-chip RAM, external RAM, or peripheral space. The shared storage can also be non-volatile, such as flash or NVRAM.

Similarly, volatile RAM can be shared between software at different privilege levels within the same world. Software at each privilege level in each world is referred to as a security domain.

Before any shared storage can be reallocated from one security domain to an untrusted security domain, the asset must be securely removed. This process is called scrubbing. Scrubbing is the process of overwriting an asset using any of the following methods:

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

The scrubbing process must be completed before reallocation. Scrubbing can be performed by either Trusted hardware or Trusted software.

	Shared storage must be scrubbed before it can be reallocated to a different security domain.
	Shared storage must not be executable immediately after reallocation to a different security domain.

When a copy of Trusted data is held in a cache, it is important that the implementation does not permit any mechanism that provides the Non-Trusted world with access to that data. If a hardware engine is used for scrubbing, careful attention must be given to the sequence to make sure that the relevant cached data is flushed and invalidated before the scrubbing operation.

5.5 Processing elements

Most security breaches are caused by software vulnerabilities. Therefore, a key aspect of hardware system architecture is selecting and configuring security features of a host processor. The goal is to support a secure software framework which minimizes the likelihood of threats identified in the security development lifecycle of the product combining with vulnerabilities in software being exploited by an attacker during product deployment.

Arm recommends that hardware security features are selected according to the product's software architecture and threat model.

At a minimum, the SoC must ensure that:

- the execution state of the Trusted world cannot be tampered by the Non-trusted world. The implications for memory transactions and interrupts are covered in other sections.
- there is enough hardware support to ensure that writable data in memory is never executable. This mitigates common "shellcode" exploits.
- controls are implemented to disable speculative execution on processors that have this characteristic.

	The SoC must provide a hardware-based mechanism for isolating the execution contexts of the Trusted world from the Non-Trusted world.
	The SoC must provide a hardware-based mechanism that ensures runtime data in memory is never executable.
	If a processor implements speculative execution, it must provide a way for software to control or disable speculation.

Other PE features should be considered beyond the base security requirements here within, such as:

- Protection against *return-oriented programming* (ROP) attacks and *jump-oriented programming* (JOP) attacks. The protection should prevent malicious code from executing illegal subsets of code functions.
- Detection of *use-after-free* (UAF) vulnerabilities; a memory safety issue in some programming languages.

5.6 Interrupts

Each world may receive interrupts of some form. An interrupt that is only meant to be received by the Trusted world is referred to as a Trusted interrupt. In most cases, a Trusted interrupt must not be visible to a Non-Trusted operation, in order to prevent information leaks that might be useful to an attacker. Consequently, the on-chip interrupt network must be able to route any interrupt to any world. However, the routing of Trusted interrupts must only be configured from the Trusted world.

When a memory access violation occurs, such as when the Non-Trusted world tries to access a Trusted asset, a *security exception* or *security interrupt* is raised.

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

These requirements permit a Non-Trusted world request to a Trusted operation to deliver a Trusted Interrupt to a Non-Trusted target, which signals the end of the operation. The configuration of the interrupt is performed by the Trusted world before or during the Trusted operation. These operations need to be handled carefully. Arm recommends that compliant designs ensure that, if a requirement allows the Non-Trusted world to trigger Secure interrupts, the hardware arrangement only allows the dedicated Secure interrupt to be triggered from the Non-Secure side. The Secure interrupt handler must be written carefully, in order to avoid denial of service attacks.

5.7 Debug

A processor typically supports at least two types of debug modes:

- Self-hosted debug: The processor itself hosts a debugger. Developer software and debugger 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).

All debug mechanisms need to be access controlled to prevent abuse. Access must be prevented 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 can include one or more DPMs. A DPM authenticates each requestor using one of the following methods:

- Token-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.
- Password-based authentication. A device does not store the password itself, but a cryptographically-strong hash of the password. The device also limits the number of attempts to prevent a brute force attack.
- Challenge/response protocol. If the communication channel between the device and the external debugger is not secure, then this is the preferred option.

Which method to use often depends on the trade-off between complexity on the device and complexity on an external server. For example, it is more complicated to implement signature checking on a device than to compare passwords, but managing a database of unique passwords is more complicated than one or two private keys on a server.

To prevent the leak of a secret from affecting multiple devices, tokens or passwords used to authenticate to DPMs should be unique for each instance.

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

Complex SoCs often include extra debug functionality beyond the main processor. Examples of this are initiators on the interconnect, which are controlled directly from an external debug interface, and system trace modules. Care must be taken to 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.

A scan chain is a mechanism to test all the flip-flops in an SoC. Scan chains are a form of debug and need to be governed by a security lifecycle to ensure they can never be accessed after a certain point. While scan chains are expected to be disabled in the factory, the specific requirements will be determined by the product.

	All scan chains must be lifecycle aware.

5.8 Peripherals and subsystems

A peripheral or subsystem is hardware that is not part of the PE. It can be a part of the SoC or connected via an off-chip bus. In many cases the hardware is an isolated system with its own local resources, configuration and firmware. It has an interface to receive commands and data from one or more PEs and might be capable of direct memory access (DMA).

5.8.1 Interfaces and policy

Peripherals offer an interface to operate on assets. These assets might be assets from the Trusted world or the Non-trusted world depending on the functionality and security provided. A Trusted peripheral is one that operates on assets belonging to the Trusted world. A few types of peripherals are possible:

- A simple peripheral can have its operations mapped exclusively into one world or the other by the wider system depending on its role.
- A simple Trusted peripheral can only act on Trusted world assets.
- A more complex Trusted peripheral might operate on assets within both worlds, supporting both Trusted and Non-trusted operations, as illustrated in Figure 2. An implementation-specific policy manages the separation, which might be fixed in hardware or configurable by a Trusted service.

Interfaces can be implemented fully in hardware or mediated by a service in the Trusted world. These interfaces permit software to request operations on data. Care must be taken care by the interface designer to ensure that Trusted assets and operations are isolated from the Non-trusted assets and operations.

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

When data is processed on behalf of multiple worlds, a policy is needed to constrain privileges based on the accessor. An example policy for a cryptographic accelerator peripheral would cover at least:

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

Figure 2 shows an illustration with a policy in place, where requests can be rejected if they do not comply with the policy.

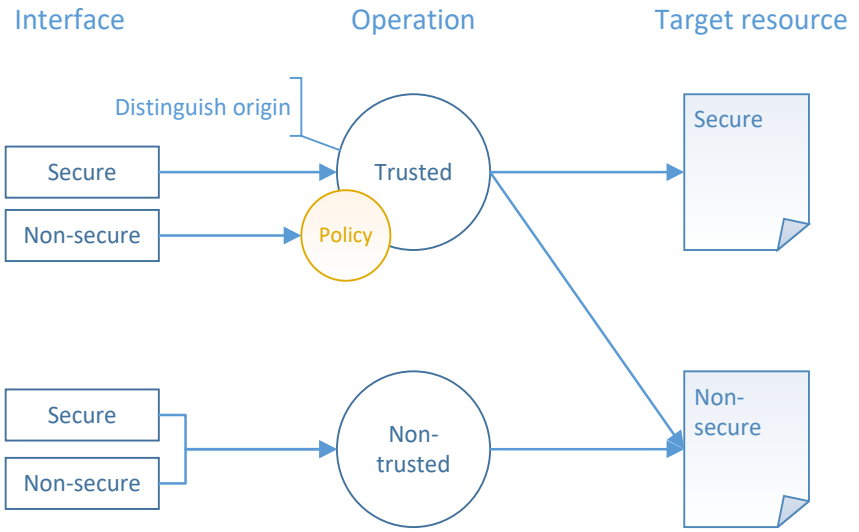

Figure 2: Peripheral operations

5.8.2 External peripherals

SoCs will often need to communicate with external peripherals to receive and transmit data. Examples of these external peripherals include network controllers, secure elements, displays, and, interface controller hubs, such as PCIe and USB chips. Some interfaces are simple connections via SPI or UART whereas others can embed high bandwidth controllers within the SoC itself.

Since external interface peripherals can easily be replaced by local attackers, it is important to protect SoC assets from DMA and PCIe transactions that originate from outside the SoC. Therefore, transactions must be constrained using an on-chip mechanism. The configuration of such a constraining mechanism can be fixed in hardware or configurable by firmware. The precise constraints will vary depending on the context. For instance, the boot process must configure the constraining mechanism to protect its own assets, while the runtime firmware or OS will have need to reconfigure the mechanism to protect a different set of assets.

Some designs are subject to threat models in which particular hardware IP blocks could have unknown or undesirable behaviors. In these cases, additional master side filters should be implemented under sole control of the Trusted world to ensure that such IP cannot access Trusted world assets beyond that authorized by a Trusted world *policy*. Arm recommends that peripherals embedded within the SoC are also constrained.

	All DMA transactions from an external peripheral must be constrained using an on-chip mechanism.
	When an external peripheral can receive commands from an external system, for example PCIe, then the system must enforce a policy to check that those commands do not breach the security of the SoC.
	If an external peripheral is used to send or receive clear or unauthenticated Trusted world assets, then it must meet the requirements for Trusted operations.

5.8.3 Security subsystems

A security subsystem is hardware that is used to operate on or store high value Trusted assets. The security services provided by a security subsystem might include one or more of the following services:

- A key for a unique, unclonable identity that is bound to hardware.
- Counters
- Key storage and management.
- Secure cryptography where keys are never visible.
- Key derivation.
- True random number generation.
- Secure storage for boot measurements, forming the basis for a system to be able to perform secure attestation.
- Transparent encryption of RAM or storage

The security hardware must be managed by the Trusted world. This ensures that the security subsystem is always available for use by the Trusted world.

If the Trusted world does not intend to use a particular security subsystem, it might choose to delegate the subsystem to the Non-trusted world, subject to the threat model of the final product.

If the security subsystem is off-chip then it is susceptible to bus interposition attacks or physical replacement. A compliant platform must ensure that the communication path is protected from eavesdropping. Communication may also require replay protection to ensure that an attacker cannot record and replay bus traffic. To ensure that an off-chip attack does not reduce security, there are two methods to consider:

- The SoC is physically tied to the security subsystem during manufacture. For example, the subsystem can be placed within the same physical packaging as the SoC, depending on the physical attacks within the threat model.
- The SoC detects replacement of the security subsystem by means of cryptographic authentication. For example, during device assembly an off-chip security subsystem is connected to the host system to establish unique shared keys. Similarly, the security subsystem must authenticate the host in order to prevent extraction of stored secrets.

A product threat model shall dictate more specific requirements of a security subsystem. For instance, there might be requirements necessitated by the chosen operating system vendor, by market, or by region in which the system is to operate in.

	An off-chip security subsystem must be physically or logically inseparable from the host system. Separation must not reduce system security.
	Communication to and from an off-chip security subsystem must be protected against eavesdropping.
	Communication to and from an off-chip security subsystem must be able to detect tampering and replay attacks
	A security subsystem key must not be directly accessible by any software unless a policy explicitly allows the key to be exported.

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

Examples of security subsystems include, but are not limited to, Security Enclaves (SEn), Secure Elements (SE) or Trusted Platform Modules (TPMs), DRAM protection subsystems, and security sensitive accelerators.

It is recommended that security subsystems are managed by the Trusted world to ensure that Trusted services can safely use them.

Some security subsystems can also offer increased tamper resistance against a variety of side channel attacks. Arm recommends increasing protection for cryptographic keys in the system by providing a security subsystem with a hardware key store that prevents the keys from being read by both Non-trusted and Trusted software.

5.9 Invasive subsystems

Invasive subsystems include any hardware system feature or interface which could be used to compromise security properties, such as:

- JTAG debug interface.
- Boundary scan interface.
- I2C interface with access to on-chip resources.
- RAS and other fault detection and recovery technologies.
- Interfaces to a power management subsystem.

	An Invasive subsystem must only be controllable from the Trusted world

5.10 Platform identity

To meet the Attestation security goal in Section 2.3, the system must include an attestation key. An attestation key is a cryptographic key that proves identity, and therefore trustworthiness, to the external world. The attestation key might be used to initially provision credentials of a market specific attestation scheme, such as one defined by the FIDO Alliance or the Trusted Computing Group (TCG), and is therefore called the Initial Attestation Key. An Initial Attestation Key might also be known as an *endorsement key* on some systems. Arm strongly recommends using public key cryptography, whereby the attestation key is a keypair consisting of a (secret) private key and a public key.

The manufacturer is expected to issue information about the key for the purposes of proving that a platform is genuine during remote attestation. The manufacturer vouches that the key is protected in a platform that they have manufactured. For example, the manufacturer can produce a public key certificate signed by their own certificate authority. The manufacturer in this context is the company who provisioned or generated the key. The certificate should be made available to the platform owner in order to participate with remote attestation services.

The Initial Attestation Key must be protected against cloning. If an attacker can copy the key to another platform then they will be able to impersonate the device. This means that the key must be safely stored in the Trusted world.

For privacy sensitive deployments, such as personal devices, it is permitted for a group of devices to share the same attestation key. This provides a degree of anonymity for device owners. However, keys should only be shared within small groups to reduce the impact of a leaked key. The particular group size might depend on the size of production batches or industry standards.

	The SoC must include an Initial Attestation Key that is either held within secure storage controlled by the Trusted world or held within a Security subsystem.
	The Initial Attestation Key must be unique per instance or per batch of devices
	In an implementation that uses a Security subsystem for cryptographic identities, the Initial Attestation Key must only be visible to the Security subsystem.
	The Initial Attestation Key must be protected by a security lifecycle.

Additional keys for firmware decryption and provisioning may also be included. These keys are either unique to the device or are class keys that are common across a family of devices.

5.11 Random number generation

Many cryptographic protocols depend on challenge response mechanisms that need truly random numbers. This makes a *true random number generator* (TRNG) an important element of a secure system.

Where a TRNG is required there is normally an associated requirement that specifies the quality of the source or, more commonly, a set of tests that must be passed. The quality of a random source is normally described in terms of entropy. For any string of bits provided by a TRNG, the maximum entropy is achieved if all bit combinations are equally probable. Recommendations can be found in NIST Special Publication 800-90B [3].

A hardware realization of a TRNG consists of two main components: an entropy source and a digital post processing block, as illustrated in Figure 3.

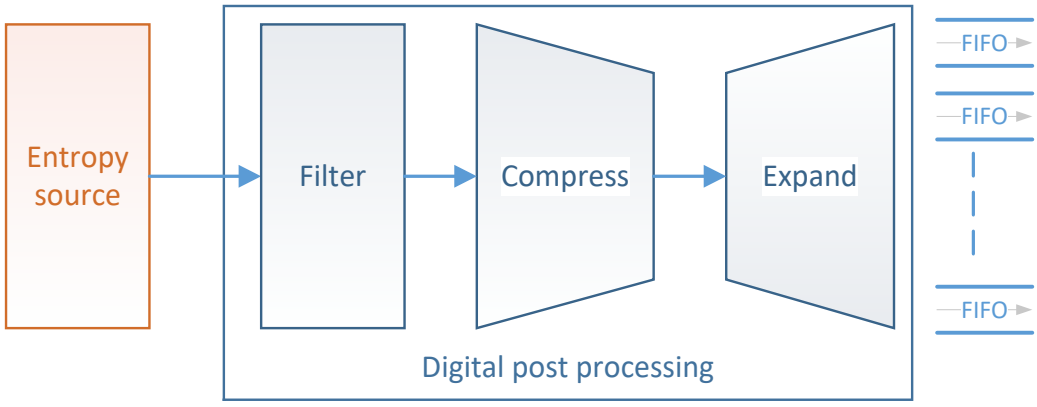

Figure 3: Entropy source top level

The entropy source incorporates the non-deterministic, entropy-providing circuitry. Constructing an on-chip entropy source might exploit die thermal noise or manufacture process variations.

Digital post-processing is responsible for collecting entropy from the source, for monitoring the quality of the data, and for filtering it to ensure a high level of entropy. For example, repeated periodic sequences are predictable and must be rejected. This is important because fault injection techniques can be used to induce predictable behavior in a TRNG.

Although a filtering scheme can remove predictable patterns in an entropy source, other more complex patterns might degrade the available entropy. The extent of any such degradation depends on the quality of the source, and in some cases additional digital processing might be required to compensate for it. A common compensation technique utilizes a cryptographic hash function to compress a large bit string of lower entropy into a smaller bit string of higher entropy. However, this comes at the expense of available data rate. To counter this, a digital post processing stage can expand the entropy source to provide a greater number of bits per second by using the filtered or compressed source to seed a cryptographically strong pseudo random sequence generator with a large period. Recommendations can be found in [4].

Each random bit generated should be used no more than once, which ensures statistical independence between samples. This applies to consecutive reads on any one interface, and for reads via different interfaces.

One or more suitably sized FIFOs might be implemented to ensure short-term peak demands are met. Each such FIFO should act like a shift-register such that the oldest un-used random number is lost as each new one is generated.

	The entropy source and post processing must be an integrated hardware block.
	It must not be possible to monitor the entropy source output on production parts.
	It must not be possible to halt the entropy source output on production parts.
	Each bit from the entropy source must be used no more than once.
	Each bit derived in post-processing must be used no more than once.

There are many possible choices for measuring entropy. The required methods will be determined by the applicable certification scheme protection profile, industry or government regulations. NIST 800-22 test suite is commonly referenced.

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

5.12 Timers

An SoC requires various forms of timer for scheduling, watchdogs and wall clock time. Any clock source that the Trusted world depends on must be resistant against tampering and must only be configured by the Trusted world.

To ensure tamper resistance for clocks that drive Trusted timers and watchdogs, two recommended strategies are possible:

- Internal clock source: the clock source is an integrated autonomous oscillator within the die and cannot be easily altered or stopped without deploying invasive techniques.
- External clock source: the clock source is connected from outside the SoC. In this case, an attacker can easily stop the clock or alter its frequency. If this is the case, then the main SoC must implement monitoring hardware that can detect when the clock frequency is outside its acceptable range.

Trusted timers are needed to provide time-based triggers to Trusted world services. The SoC must support one or more Trusted timers.

	All clock sources used by the Trusted world must be resistant against physical tampering.
	At least one Trusted timer must exist.
	A Trusted timer must only be modified by a Trusted access. Examples of modifications are the timer being refreshed, suspended, or reset.
	A Trusted timer must only produce Trusted interrupts
	The clock source that drives a Trusted timer must be exclusively controlled by the Trusted world.

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

Trusted watchdog timers might be needed to protect against denial of service, for example where secure services depend on the non-secure scheduler. In such cases, if the Trusted world is not entered before a pre-defined time limit, a reset is issued and the SoC is restarted.

It is desirable for a Trusted watchdog timer to have the ability to signal an interrupt in advance of the reset, permitting some state save before a reboot.

Arm recommends that a clock speed of at least 1Hz is used when the system is not in a power-saving cycle.

	A Trusted watchdog timer must only be modified by a Trusted access. Examples of modifications are the timer being refreshed, suspended, or reset.
	A Trusted watchdog must only produce Trusted interrupts
	Before needing a refresh, a Trusted watchdog timer must be capable of running for a time period that is long enough to complete critical sections.
	A Trusted watchdog timer must be able to trigger a reset of the SoC, after a pre-defined period of time. This value is fixed in hardware or programmed by a Trusted access.
	A Trusted watchdog timer must implement a flag that indicates the occurrence of a timeout event that causes a warm reset, to allow post-reset software to distinguish this from a powerup cold boot.
	The clock source driving a Trusted watchdog timer must be exclusively controlled by the Trusted world.

After a system reset, Arm recommends that a Trusted watchdog timer should be started before execution of the immutable boot code transfers control to the next firmware stage.

Trusted services may rely on the availability of Trusted time. Trusted time is typically implemented using an on-chip real-time counter that is synchronized securely with a remote time server.

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

A suitable solution for dealing with power outages can be realized by implementing a counter together with a status flag that indicates whether a valid time has been loaded.

An SoC that deploys this solution implements Trusted time using a TRTC that consists of a Trusted hardware timer that is associated with a status flag that indicates whether the current time is valid, and receives a Trusted clock source. The valid flag is set when the Trusted timer has been updated by a Trusted service and is cleared when power is removed from the timer. Arm recommends that the Trusted timer and valid flag reside in a power domain that remains on as much as possible.

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

	A TRTC must be configured only by a Trusted world access.
	All components of a TRTC must be implemented within the same power domain.
	On initial power up, and following any other outage of power to the TRTC, a validity mechanism must indicate that the TRTC is not Trusted.
	The TRTC must be exclusively controlled by the Trusted world.

5.13 Cryptography

The cryptographic algorithms that are used must be strong against networked adversaries and local attackers. The specific choice of algorithms depends on the target market and local regulations.

The strength of a cryptographic algorithm is determined by the number of operations that is required to break it in some way. If the security strength associated with an algorithm or system is S bits, then it is expected that (roughly) 2^S basic operations are required to break it.

It must be noted that S bits does not refer to a key length. For example, to meet 128 bits of security strength:

- An RSA-based key must be at least 3072 bits.
- An elliptic-curve-based key must be at least 256 bits.

Further information can be found in externally published documents from the cryptographic community and governments. 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. Arm strongly recommends vendors verify that the implemented algorithms execute in constant time, in order to prevent timing-based attacks.

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

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

	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 cryptographic scheme provides one or more security services and is based on a purpose and an algorithm requiring specific key properties and key management.
Keys are characterized depending on their classification as private, public, or symmetric keys and according to their use.

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

1. Limiting the uses of a key limits the potential harm if the key is compromised.
2. The use of a single key for two or more different cryptographic schemes can reduce the security provided by one or more of the processes.
3. Different uses of a single key can lead to conflicts in the way each key should be managed. For example, the different lifetime of keys used in different cryptographic processes may result in keys having to be retained longer than is best practice for one or more uses of that key.

In cases where a scheme can provide more than one cryptographic service, this principle does not prevent use of a single key. For instance, when a symmetric key is used both to encrypt and authenticate data in a single operation or when a digital signature is used to provide both authentication and integrity.

Re-using part of a larger key in a scheme that uses a shorter key, or using a shorter key in a larger algorithm and padding the key input, can leak information about the key. This is prohibited.

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

An SoC includes a number of different keys during its operation. Different keys in the same the system can have different lifespans. Some keys are programmed during SoC manufacture and never change, while others exist only during a communication session. These include static keys, ephemeral keys, and hardware keys:

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

- An *ephemeral key* is a key that has a short lifespan. In many cases, they only exist between power cycles of the device. Ephemeral keys are created in the device in several ways, such as key derivation. Ephemeral keys can also be sourced from a TRNG at boot time. This method is preferred because it gives better protection by generating keys that are unique for every boot cycle.
- A *hardware key* is a key that is not visible to software and is either static or ephemeral. It is used for Trusted world cryptographic operations, but its usage in a Non-Trusted world must be subject to a policy. An example key usage policy would cover at least the following:
    - The world the input data is permitted to be read from.
    - The world the output data is permitted to be written to.
    - Permitted operations.
- A *temporally isolated* key is a key that is only available at a specific point in time. For example, a bootloader can derive a key from source material on each system reset, use the key, erase the key, and finally triggers a hardware mechanism to hide the key (or its source material) until the next system reset. This allows the bootloader to have a secret that cannot be derived or used by software that is later loaded.

	When a key is no longer required by the system, it must be put beyond use to prevent a hack at a later time from revealing it.
	A static key must be stored in an immutable structure, for example a ROM or a set of bulk-lockable fuses.
	To prevent the re-derivation of previously used keys, the source material must be hidden either by Trusted code or Trusted hardware.
	If an ephemeral key is stored in memory or in a register in clear text form, the storage location must be scrubbed before being used for another purpose.
	A key that is accessible to, or generated by, the Non-trusted world must only be used for Non-Trusted world cryptographic operations, which are operations that are either implemented in Non-trusted world software, or have both clear text and cipher text in the Non-trusted world.
	A key that is accessible to, or generated by, the Trusted world can be used for operations in both Non-Trusted and Non-trusted worlds, and even across worlds, if: • The Non-Trusted world cannot access the key directly. • The Trusted world can control the use of the key through a policy.
	A Trusted hardware key must not be directly accessible by any software.
	The Trusted world must be able to enforce a usage policy for any Trusted hardware key that can be used for Non-Trusted world cryptographic operations.

5.14 Secure storage

Trusted assets, such as firmware images and sensitive data, often need to be stored in external storage. The external storage needs to be protected from attackers who may try to read, modify or clone the contents. Therefore, a compliant SoC must provide a secure storage solution by embedding:

- A *hardware unique key (HUK)* for encrypting and decrypting external storage. The key is hardware unique so that assets cannot be cloned or decrypted on another platform.
- An on-chip Trusted non-volatile counter, which is required for version control of firmware and trusted data held in external storage. An important property of these counters is that it must not be possible to roll them back, to prevent replay attacks. There must be at least one counter for Trusted firmware use and at least one counter for Non-trusted firmware use.

An implementation of secure storage can be made with a Trusted service or by using a hardware approach. A hardware implementation of secure storage can be transparent to software and provide increased throughput compared to a software solution. However, a hardware implementation must conform to the rules described in the Security Subsystems section.

	Any sensitive data that needs to be stored must be stored in Secure storage.
	The SoC must embed 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.
	An on-chip non-volatile Trusted firmware version counter implementation must provide a counter range of 0-63.
	An on-chip non-volatile Non-trusted firmware version counter implementation must provide a counter range of 0-255.
	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.

Ideally, an SoC implementation implements secure storage and version counters using on-chip *multiple time programmable* (MTP) storage; for example, floating gate (EE ROM) or phase transition technology. It is recognized that an MTP-based approach is currently not economically scalable for smaller process nodes because the overhead is costly compared to a standard bulk CMOS process. By contrast, *one-time programmable* (OTP) storage, which is based on anti-fuse technology, is widely available and cost- effective.

5.15 Main memory

Trusted code is expected to execute from Secure RAM. The Trusted code also stores high value assets within the Secure RAM. In the context of this document, Secure RAM refers to one or more dedicated regions that are mapped onto one or more physical RAMs. When a physical RAM is not entirely dedicated to Secure storage, it can be configured to be shared between worlds. However, the underlying locations are not classified as shared volatile storage unless they are reallocated from Secure to Non-secure. The mapping of Secure regions can be static and fixed by design, or programmable at runtime.

Example Secure RAM use cases are:

- Secure boot code and data.
- Secure Monitor code.
- A Trusted OS or a Secure Partition Manager.
- Cryptographic services.
- Trusted services.

The size of the Secure RAM depends on the target requirements and is therefore not specified in this document. Arm recommends the use of on-chip RAM, but it is acceptable to use SRAM on a separate die if it is within the same package as the main SoC.

	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.

Many SoC designs rely on external DRAM for operating on assets. External memory is vulnerable to probing attacks, which can be used to:

- Directly recover sensitive assets.
- Subvert the behavior of the system to extract assets, or to use the system for illegitimate purposes.

To mitigate these risks, various protections can be applied to an asset before it is stored in DRAM. The amount of protection that is required depends on the deployment of the target system.

There are three general tiers of protection, each of which provide increased resistance to physical attack. In general, each increase in protection requires more complex hardware. The appropriate level should be determined by a product's requirements and deployment threat model.

5.15.1 Confidentiality protection

An attacker that can freeze external memory or use a battery-backed DRAM module can directly recover any asset in main memory. This is known as a "cold boot" attack.

DRAM encryption ensures that assets in main memory cannot be read by physical attackers. Encryption can be transparently provided through performance-optimized on-chip cryptographic hardware blocks, each of which receives a symmetric key. Alternatively, the encryption might be provided by software that executes in on-chip

Trusted SRAM, at the expense of performance and coverage. The required level of cryptographic protection depends on the target requirements and is not specified here.

When DRAM encryption is implemented, it must not be possible to decrypt a copy of the memory contents on a different device. Therefore, the keys used for encryption and authentication must be unique to the SoC. Arm recommends that the keys are randomly generated on each system reset.

5.15.2 Integrity protection

In addition to encryption, external modification of DRAM data can be detected, enabling execution to be halted to prevent an attacker from exploiting any such modifications. This is known as integrity protection.

The most economical solution might involve using a hardware IP block that transparently creates and manages cryptographic hashes of memory blocks.

The required level of cryptographic protection depends on the target requirements and is not specified here.

5.15.3 Replay protection

An attacker with highly specialized equipment might be able to capture and reproduce memory content, either by directly altering the contents in physical memory, or by interposing on bus transactions between the SoC and the external memory.

With this capability an attacker can force an SoC to accept a piece of captured memory that passes integrity checks and correctly decrypts. For an attacker to exploit this vulnerability:

1. They must capture memory content at an address that is known to contain an insecure value or insecure configuration.
2. The attacker then “replays” this memory content when the SoC requests for this memory and at a point in time that is suitable for the attacker. This is a form of “timing attack”.
3. Once the SoC receives the memory transaction, it may operate on it, which may reduce or deactivate software defenses. For example, the memory content might contain system configuration that was previously safe but is now unsafe, which is then written into a privileged configuration register.

With the addition of replay protection, attackers cannot used captured memory to mount a timing attack. The degree of specialty required depends on the speed and complexity of bus transactions.

5.15.4 Usage

The choice of confidentiality, integrity and replay protection depends on the threat model of the final system. However, there are some common rules that apply in all cases, which are described below.

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

The addition of such cryptographic hardware in the data path to the memory system often carries performance penalties that typically increase with the cryptographic strength.

Designers must consider if their threat model indicates that DRAM integrity protection is required. A row hammer attack exploits unintended physical side-effects of DRAM memory to change the values stored in other memory cells. Target Row Refresh (TRR) as defined by JEDEC, should be implemented to add resistance to such attacks.