TrustZone for AArch64

Revision: 0101

Non-Confidential

Issue 1.1

102418

TrustZone for AArch64

Release information

Document history

Issue	Date	Confidentiality	Change
1.0	08 January 2020	Non-Confidential	1
1.1	21 Decemeber 2021	Non-Confidential	Armv9-A updates

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

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

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at **https://www.arm.com/company/policies/trademarks**.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please email **terms@arm.com**.

Contents

1 Overview

In this guide, we introduce TrustZone. TrustZone offers an efficient, system-wide approach to security with hardware-enforced isolation built into the CPU.

We cover the features that TrustZone adds to the processor architecture, the memory system support for TrustZone, and typical software architectures. We also introduce the resources that are available from Arm to aid system and software developers who are working with TrustZone.

At the end of this guide, you will be able to:

- Give an example use case for TrustZone, describing how TrustZone is used to fulfill a security need
- List the number of Security states and physical address spaces in a TrustZone system
- State the purpose of a Secure Monitor and give examples of the state that it is required to save or restore
- Name the components in a typical TrustZone enabled memory system and describe their purpose
- Explain the purpose of the Trusted Base System Architecture and Trusted Board Boot Requirements specifications from Arm
- Explain how a chain of trust is used to secure the boot of a device

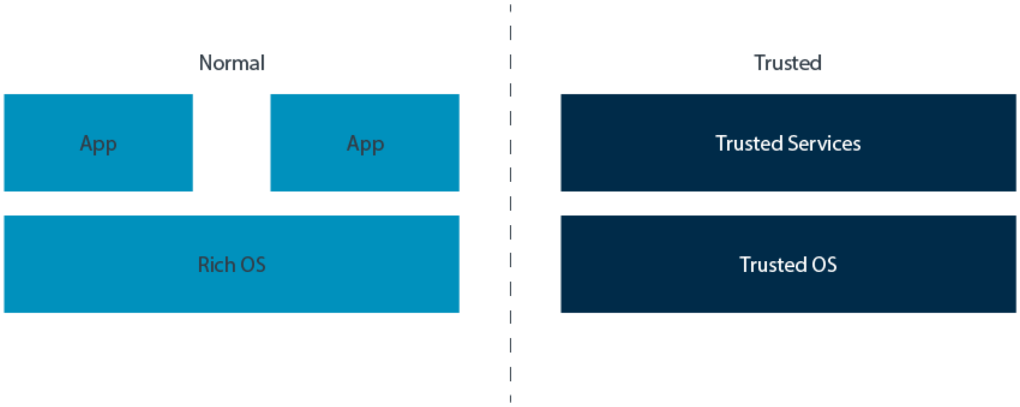
1.1 Before you begin

This guide assumes that you are familiar with the Arm Exception model and memory management. If you are not familiar with these subjects, read our Exception model and Memory management guides.

If you are not familiar with security concepts, we also recommend that you read our Introduction to security guide before reading this guide.

2 What is TrustZone?

TrustZone is the name of the Security architecture in the Arm A-profile architecture. First introduced in Armv6K, TrustZone is also supported in Armv7-A and Armv8-A. TrustZone provides two execution environments with system-wide hardware enforced isolation between them, as shown in this diagram:



The Normal world runs a rich software stack. This software stack typically includes a large application set, a complex operating system like Linux, and possibly a hypervisor. Such software stacks are large and complex. While efforts can be made to secure them, the size of the attack surface means that they are more vulnerable to attack.

The Trusted world runs a smaller and simpler software stack, which is referred to as a Trusted Execution Environment (TEE). Typically, a TEE includes several Trusted services that are hosted by a lightweight kernel. The Trusted services provide functionality like key management. This software stack has a considerably smaller attack surface, which helps reduce vulnerability to attack.

Note

You might sometimes see the term Rich Execution Environment (REE) used to describe the software that is running in the Normal world.

TrustZone aims to square a circle. As users and developers, we want the rich feature set and flexibility of the Normal world. At the same time, we want the higher degrees of trust that it is possible to achieve with a smaller and more restricted software stack in the Trusted world. TrustZone gives us both, providing two environments with hardware-enforced isolation between them.

2.1 TrustZone for Armv8-M

TrustZone is also used to refer the Security Extensions in the Armv8-M architecture. While there are similarities between TrustZone in the A profile architecture and the M profile architecture, there are also important differences. This guide covers the A profile only.

2.2 Armv9-A Realm Management Extension

The Armv9-A Realm Management Extension (RME) extends the concepts supported by TrustZone. This guide does not cover RME, but you can find more information in the Realm Management Extension Guide.

3 TrustZone in the processor

In this topic, we discuss support for TrustZone within the processor. Other sections cover support in the memory system and the software story that is built on the processor and memory system support.

3.1 Security States

In the Arm architecture, there are two Security states: Secure and Non-secure. These Security states map onto the Trusted and Normal worlds that we referred to in What is TrustZone?

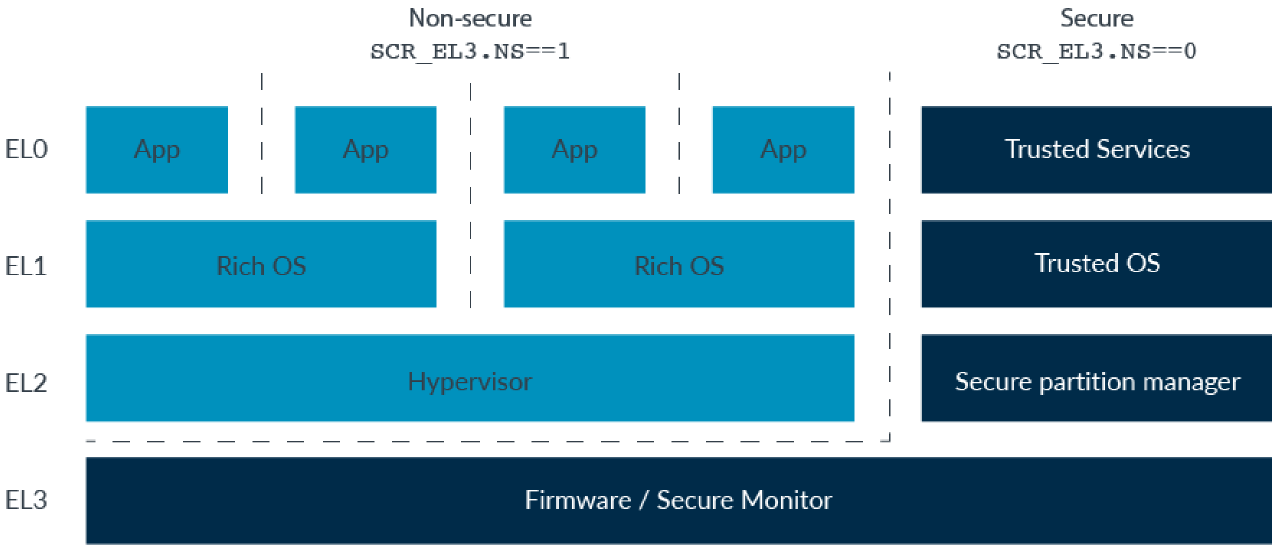
Note

In Armv9-A, if the Realm Management Extension (RME) is implemented, then there are two extra Security states. This guide does not cover the change introduced by RME, for more information on RME, see Realm Management Extension Guide.

At EL0, EL1, and EL2 the processor can be in either Secure state or Non-secure state, which is controlled by the `SCR_EL3.NS` bit. You often see this written as:

- NS.EL1: Non-secure state, Exception level 1
- S.EL1: Secure state, Exception level 1

EL3 is always in Secure state, regardless of the value of the SCR_EL3.NS bit. The arrangement of Security states and Exception levels is shown here:



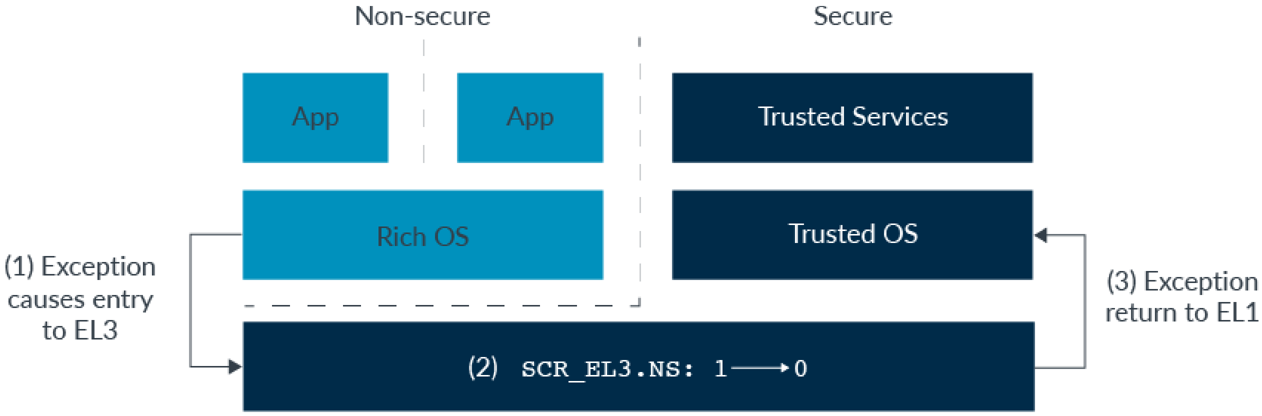
Note

Support for Secure EL2 was first introduced in Armv8.4 - A and support remains optional in Armv8-A.

3.2 Switching between Security states

If the processor is in NS.EL1 and software wants to move into S.EL1, how does it do this?

To change Security state, in either direction, execution must pass through EL3, as shown in the following diagram:

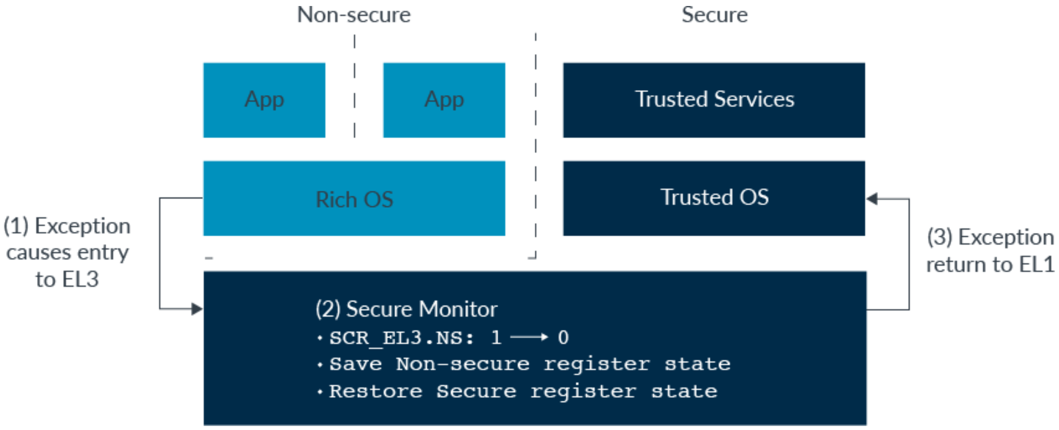


The preceding diagram shows an example sequence of the steps that are involved in moving between Security states. Taking these one step at a time:

1. Entering a higher Exception level requires an exception. Typically, this exception would be an FIQ or an SMC (Secure Monitor Call) exception. We look at interrupt handling and SMCs in more detail later.
2. EL3 is entered at the appropriate exception vector. Software that is running in EL3 toggles the `SCR_EL3.NS` bit.
3. An exception return then takes the processor from EL3 to S.EL1.

There is more to changing Security state than just moving between the Exception levels and changing the `SCR_EL3.NS` bit. We also must consider processor state.

There is only one copy of the vector registers, the general-purpose registers, and most System registers. When moving between Security states it is the responsibility of software, not hardware, to save and restore register state. By convention, the piece of software that does this is called the Secure Monitor. This makes our earlier example look more like what you can see in the following diagram:


Non-secure
Secure
App
App
Trusted Services
Rich OS
Trusted OS
(1) Exception causes entry to EL3
(3) Exception return to EL1
(2) Secure Monitor
• SCR_EL3.NS: 1 → 0
• Save Non-secure register state
• Restore Secure register state

Trusted Firmware, an open-source project that Arm sponsors, provides a reference implementation of a Secure Monitor. We will discuss Trusted Firmware later in the guide.

A small number of registers are banked by Security state. This means that there are two copies of the register, and the core automatically uses the copy that belongs to the current Security state. These registers are limited to the ones for which the processor needs to know both settings at all times. An example is `ICC_BPR1_EL1`, a GIC register that is used to control interrupt preemption. Banking is the exception, not the rule, and will be explicitly called out in the Architecture Reference Manual for your processor.

When a System register is banked, we use (S) and (NS) to identify which copy we are referring to. For example,

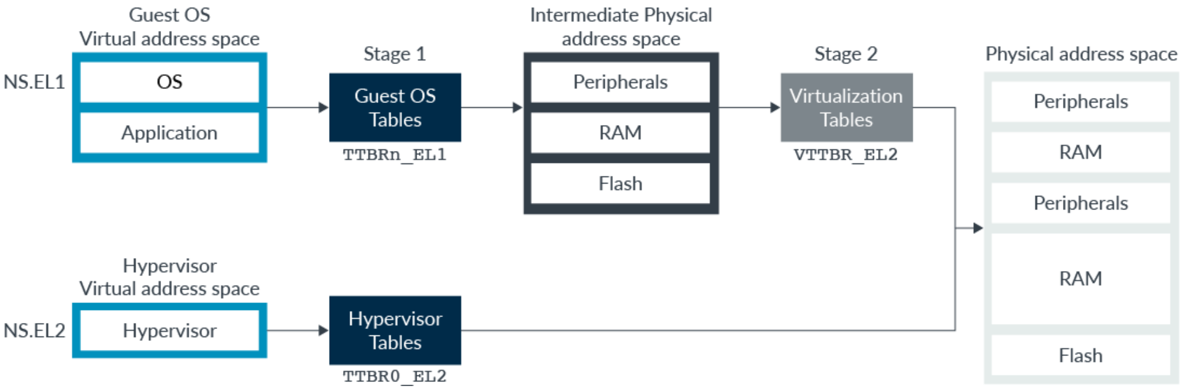
`ICC_BPR1_ EL1` (S) and `ICC_BPR1_EL1` (NS).

Note

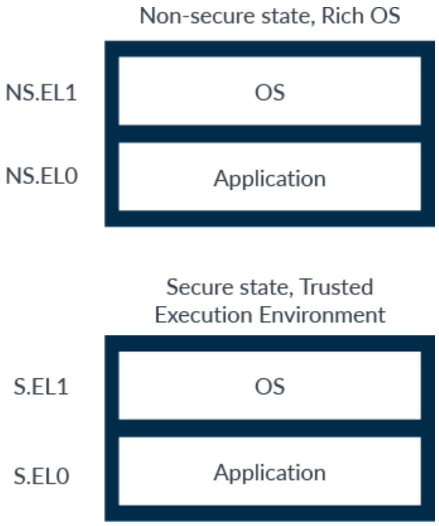
In Armv6 and Armv7 - A most System registers are banked by Security state, but general- purpose registers and vector registers are still common.

3.3 Virtual address spaces

The memory management guide in this series introduced the idea of multiple virtual address spaces, or translation regimes. For example, there is a translation regime for EL0/1 and a separate translation regime for EL2, shown here:



There are also separate translation regimes for the Secure and Non-secure states. For example, there is a Secure EL0/1 translation regime and Non- secure EL0/1 translation regime, which is shown here:


Non-secure state, Rich OS
NS.EL1
OS
NS.EL0
Application
Secure state, Trusted Execution Environment
S.EL1
OS
S.EL0
Application

When writing addresses, it is convention to use prefixes to identify which translation regime is being referred to:

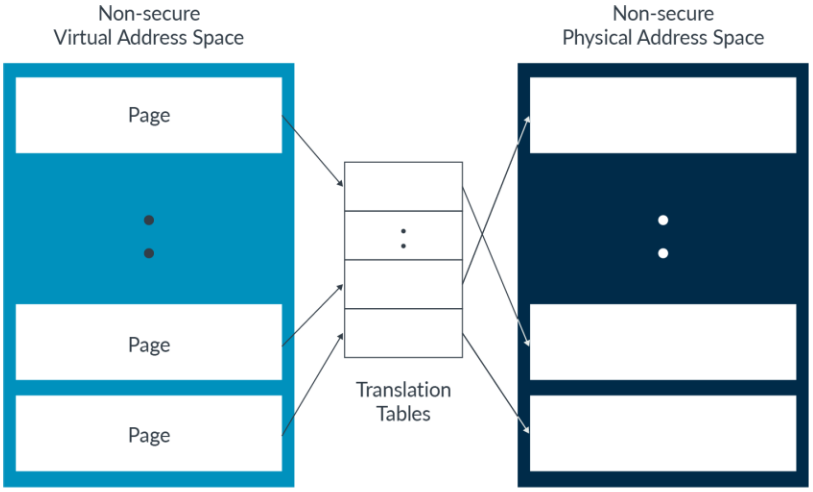
- `NS.EL1:0x8000` - Virtual address 0x8000 in the Non-secure EL0/1 translation regime
- `S.EL1:0x8000` - Virtual address 0x8000 in the Secure EL0/1 translation regime

It is important to note that `S.EL1:0x8000` and `NS.EL1:0x8000` are two different and independent virtual addresses. The processor does not use a NS.EL1 translation while in Secure state, or a S.EL1 translation while in Non-secure state.

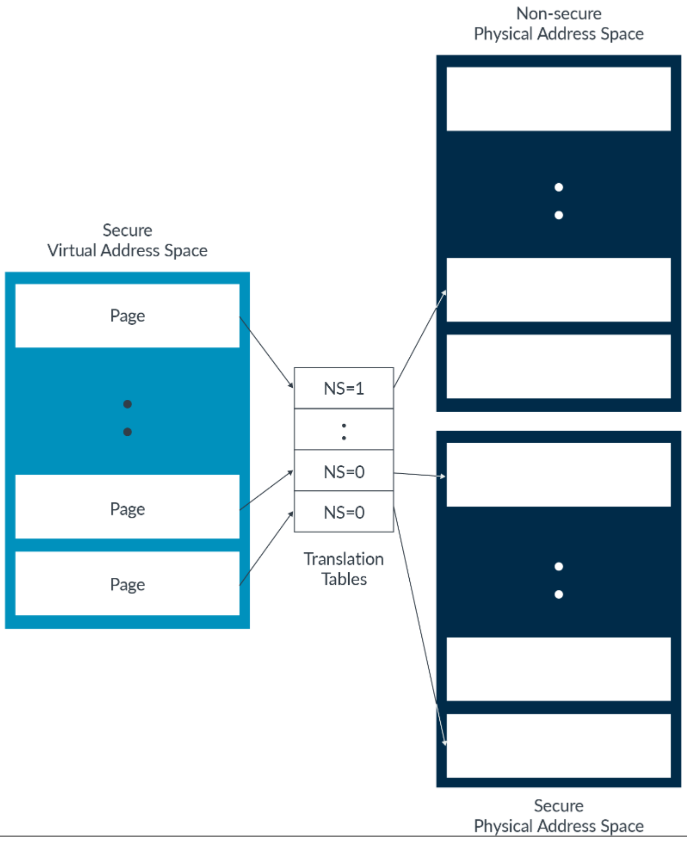
3.4 Physical address spaces

In addition to two Security states, the architecture provides two physical address spaces: Secure and Non-secure.

While in Non-secure state, virtual addresses always translate to Non-secure physical addresses. This means that software in Non-secure state can only see Non-secure resources, but can never see Secure resources. This is illustrated here:


Non-secure Virtual Address Space
Page
Page
Page
Translation Tables
Non-secure Physical Address Space

While in Secure state, software can access both the Secure and Non-secure physical address spaces. The NS bit in the translation table entries controls which physical address space a block or page of virtual memory translates to, as shown in the following diagram:



Note

In Secure state, when the Stage 1 MMU is disabled all addresses are treated as Secure.

Like with virtual addresses, typically prefixes are used to identify which address space is being referred to. For physical addresses, these prefixes are NP: and SP:.For example:

- `NP:0x8000` – Address 0x8000 in the Non-secure physical address space
- `SP:0x8000` – Address 0x8000 in the Secure physical address space

It is important to remember that Secure and Non-secure are different address spaces, not just an attribute like readable, or writable. This means that `NP:0x8000` and `SP:0x8000` in the preceding example are different memory locations and are treated as different memory locations by the processor.

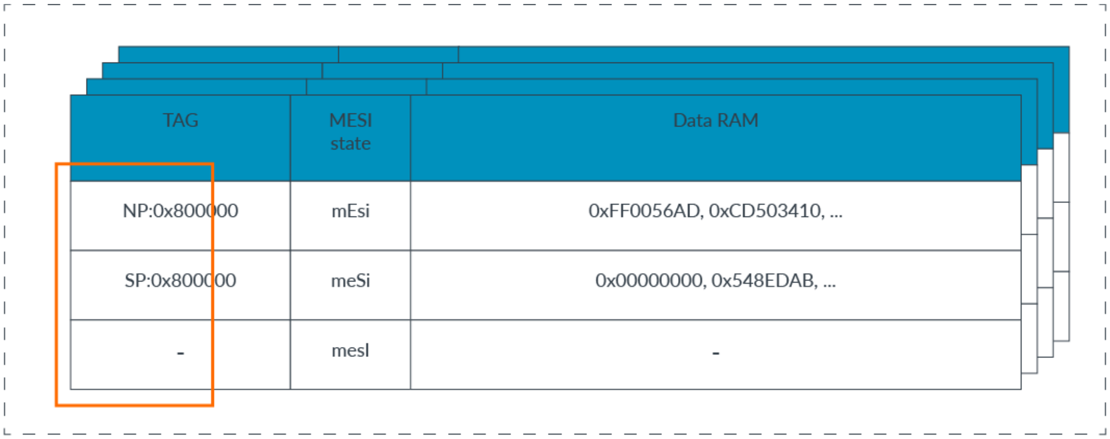
Note

It can be helpful to think of the address space as an extra address bit on the bus.

If the Armv9-A Realm Management Extension (RME) is implemented, the number of physical address spaces increases to four. The extra physical address spaces are Root and Realm. Software running in Secure state can still only access the Non-secure and Secure physical address spaces. For more information on RME, see Realm Management Extension Guide.

3.5 Data, instruction and unified caches

In the Arm architecture, data caches are physically tagged. The physical address includes which address space the line is from, shown here:



TAG	MESI state	Data RAM
NP:0x800000	mEsi	0xFF0056AD, 0xCD503410, ...
SP:0x800000	meSi	0x00000000, 0x548EDAB, ...
-	mesI	-

Example 4-way data-cache

A cache lookup on `NP:0x800000` never hits on a cache line that is tagged with `SP:0x800000`. This is because `NP:0x800000` and `SP:0x800000` are different addresses.

This also affects cache maintenance operations. Consider the example data cache in the preceding diagram. If the virtual address `va1` maps to physical address `0x800000`, what happens when software issues `DC IVAC, va1` (Data or unified Cache line Invalidate by Virtual Address) from Non-secure state?

The answer is that in Non-secure state, all virtual addresses translate to Non-secure physical addresses. Therefore, `va1` maps to `NP:0x800000`. The cache only operates on the line containing the specified address, in this case `NP:0x800000`. The line containing `SP:0x800000` is unaffected.

Check your knowledge

1. If we performed the same operation from Secure state, with va1 still mapping to NP:0x800000, which caches lines are affected?
2. Is it possible to perform a cache operation by virtual address from Non-secure targeting a Secure line?

Answers

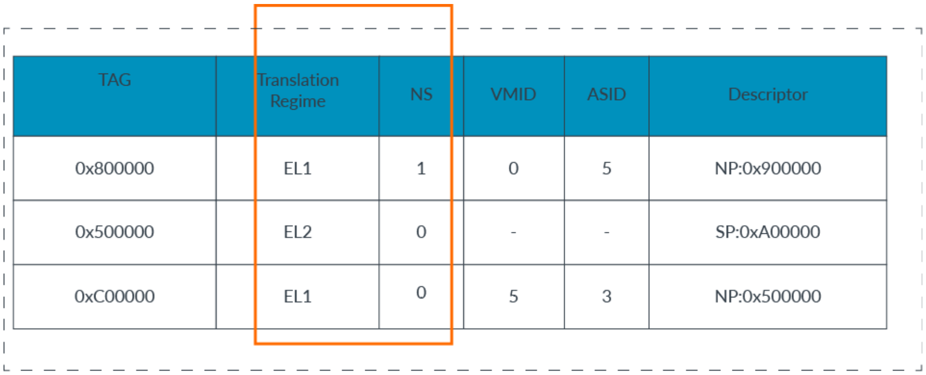
1. Like in the earlier example, the cache invalidates the line containing the specified physical address, NP:0x800000. The fact that the operation came from Secure state does not matter.
2. No. In Non-secure state, virtual addresses can only ever map to Non-secure physical addresses. By definition, a cache operation by VA from Non-secure state can only ever target Non-secure lines.

For set/way operations, for example DC ISW or Xt, operations that are issued in Non-secure state will only affect lines containing Non-secure addresses. From Secure state set/way operations affect lines containing both Secure and Non-secure addresses.

This means that software can completely invalidate or clean the entire cache only in Secure state. From Non-secure state, software can only clean or invalidate Non-secure data.

3.6 Translation Lookaside Buffer

Translation Lookaside Buffer (TLBs) cache recently used translations. The processor has multiple independent translation regimes. The TLB records which translation regime, including the Security state, an entry represents. While the structure of TLBs is implementation defined, the following diagram shows an example:



TAG	Translation Regime	NS	VMID	ASID	Descriptor
0x800000	EL1	1	0	5	NP:0x900000
0x500000	EL2	0	-	-	SP:0xA00000
0xC00000	EL1	0	5	3	NP:0x500000

Translation Look-aside Buffer (TLB)

When software issues a TLB invalidate operation (`TLBI` instruction) at EL1 or EL2, the software targets the current Security state. Therefore, `TLBI ALLE1` from Secure state invalidates all cached entries for the S.EL0/1 translation regime.

EL3 is a special case. As covered earlier in Security states, when in EL0/1/2 the `SCR_EL3.NS` bit controls which Security state the processor is in. However, EL3 is always in Secure state, regardless of the SCR_EL3.NS bit. When in EL3, `SCR_EL3.NS` lets software control which Security state `TLBIs` operate on.

For example, executing `TBLI ALLE1` at EL3 with:

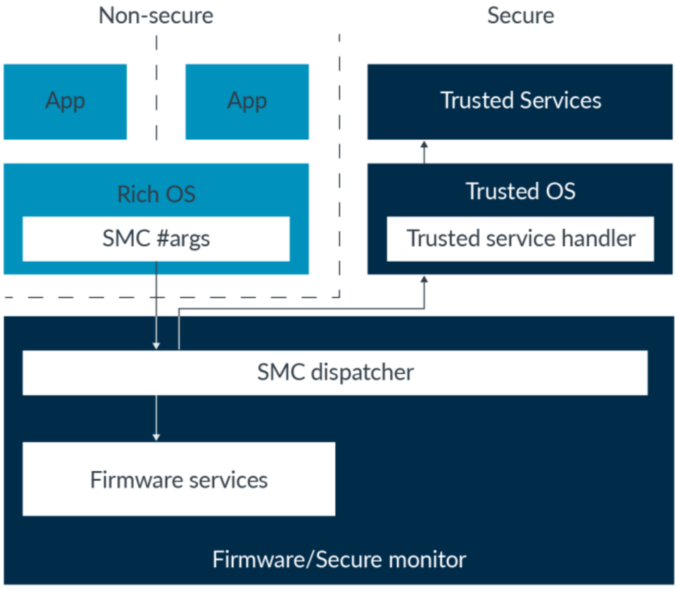
- `SCR_EL3.NS==0`: Affects Secure EL0/1 translation regime
- `SCR_EL3.NS==1`: Affects Non-secure EL0/1 translation regime

3.7 SMC exceptions

As part of the support for two Security states, the architecture includes the Secure Monitor Call (SMC) instruction. Executing SMC causes a Secure Monitor Call exception, which targets EL3.

SMCs are normally used to request services, either from firmware resident in EL3 or from a service that is hosted by the Trusted Execution Environment. The SMC is initially taken to EL3, where an

SMC dispatcher determines which entity the call will be handled by. This is shown in the following diagram:



In a bid to standardize interfaces, Arm provides the SMC Calling Convention (DEN0028) and Power State Coordination Interface specification (DEN0022). These specifications lay out how SMCs are used to request services.

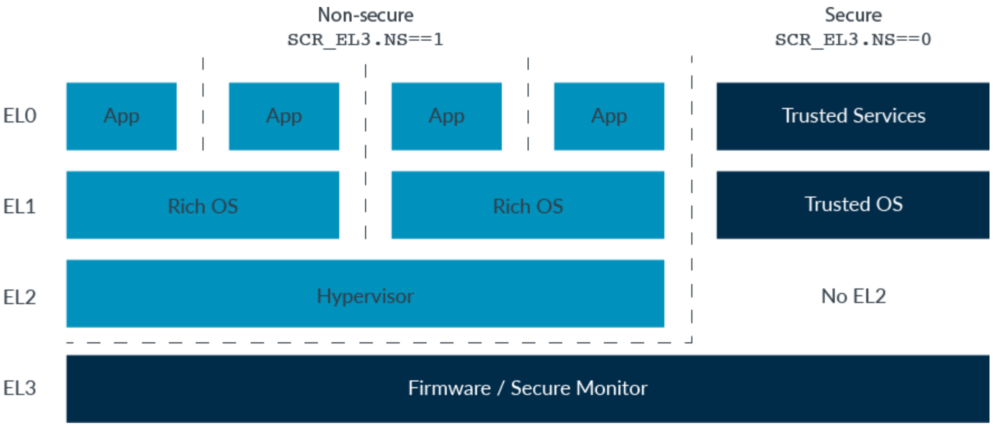
Execution of an SMC at EL1 can be trapped to EL2. This is useful for hypervisors, because hypervisors might want to emulate the firmware interface that is seen by a virtual machine.

Note

The SMC instruction is not available at EL0 in either Security state. We discuss exceptions later in Interrupts when we look at the interrupt controller.

3.8 Secure virtualization

When virtualization was first introduced in Armv7-A, it was only added in the Non-secure state. Until Armv8.3, the same was true for Armv8 as illustrated in the following diagram:

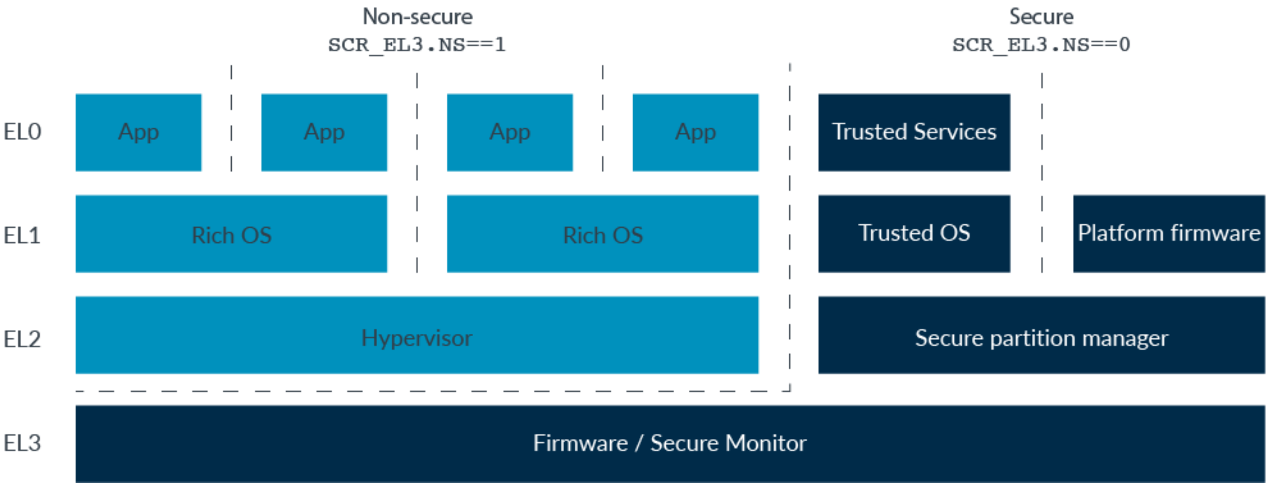


As previously described in Switching between Security States, EL3 is used to host firmware and the Secure Monitor. Secure EL0/1 host the Trusted Execution Environment (TEE), which is made up of the Trusted services and kernel.

There was no perceived need for multiple virtual machines in Secure state. This means that support for virtualization was not necessary. As TrustZone adoption increased, several requirements became apparent:

- Some Trusted services were tied to specific Trusted kernels. For a device to support multiple services, it might need to run multiple Trusted kernels.
- Following the principle of running with least privilege, moving some of the firmware functionality out of EL3 was required.

The solution was to introduce support for EL2 in Secure state, which came with Armv8.4-A, as you can see in this diagram:



Rather than a full hypervisor, S.EL2 typically hosts a Secure Partition Manager (SPM). An SPM allows the creation of the isolated partitions, which are unable to see the resources of other partitions. A system could have multiple partitions containing Trusted kernels and their Trusted services.

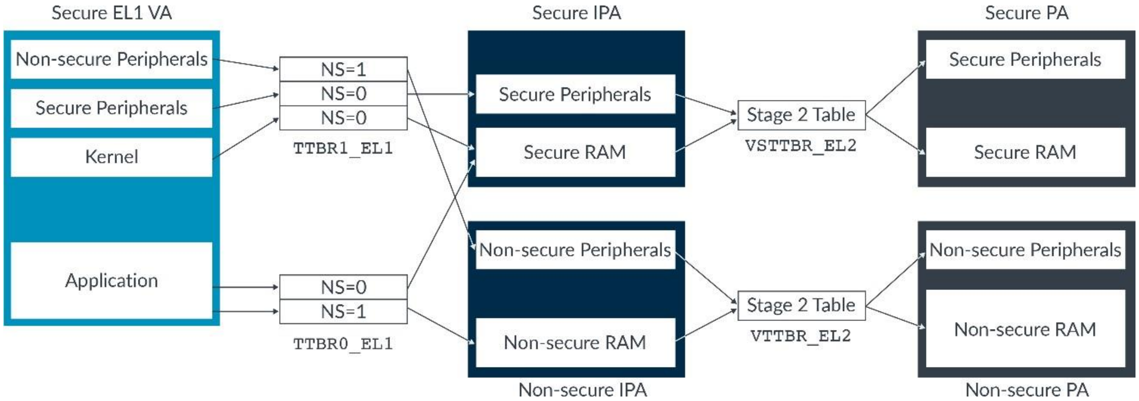
A partition can also be created to house platform firmware, removing the need to have that code that is run at EL3.

- Enabling Secure EL2

  When S.EL2 is supported, it can be enabled or disabled. Whether S.EL2 is enabled is controlled by the `SCR_EL3.EEL2` bit:

  - 0: S.EL2 disabled, behavior is as on a processor not supporting S.EL2
  - 1: S.EL2 enabled

- Stage 2 translation in Secure state

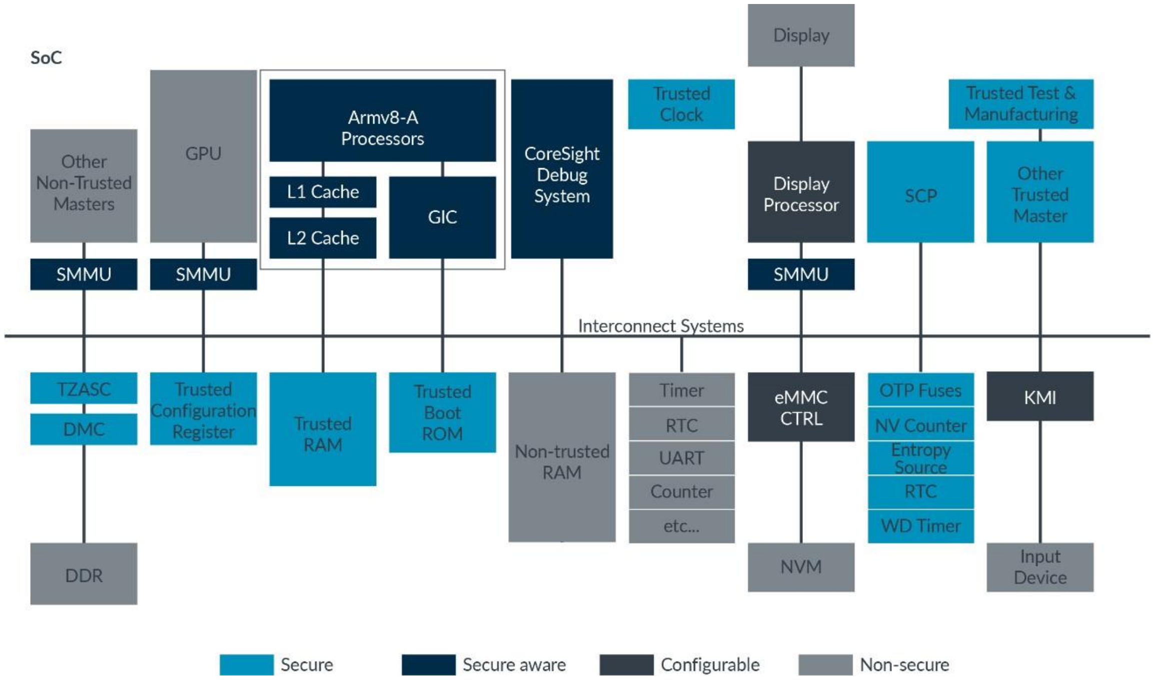


Unlike the Stage 1 tables, there is no NS bit in the Stage 2 table entries. For a given IPA space, all the translations either result in a Secure or Non-secure physical address, which is controlled by a register bit. Typically, the Non-secure IPAs translate to Non-secure PAs and the Secure IPAs translate to Secure PAs.

4 System Architecture

So far in this guide, we have concentrated on the processor, but TrustZone is much more than just a set of processor features. To take advantage of the TrustZone features, we need support in the rest of the system as well.

Here is an example of a TrustZone-enabled system:



This section explores the key components in this system and their role in TrustZone.

4.1 Completers: peripherals, and memories

Earlier in the Physical address spaces section we introduced the idea of two physical address spaces, Secure and Non-secure. The processor exports the address space that is being accessed to the memory system. The memory system uses this information to enforce the isolation.

In this topic, we refer to bus Secure and bus Non-secure. Bus Secure means a bus access to the Secure physical address space. Bus Non-secure means a bus access to the Non-secure physical address space. Remember that in Secure state software can access both physical address spaces. This means that the security of the bus access is not necessarily the same as the Security state of the processor that generated that access.

Note

In AMBA AXI and ACE, the `AxPROT[1]` signal is used to specify which address space is being accessed. Like with the NS bit in the translation tables, 0 indicates Secure and 1 indicates non-secure.

In theory, a system could have two entirely separate memory systems, using the accessed physical address space (`AxPROT`) to select between them. In practice this is unlikely. Instead, systems use the physical address space like an attribute, controlling access to different devices in the memory system.

In general, we can talk about two types of memories and peripherals, and bus completers.

4.2 Enforcing Isolation

TrustZone is sometimes referred to as a completer-enforced protection system. The requestor signals the security of its access and the memory system decides whether to allow the access. How is the memory system-based checking done?

In most modern systems, the memory system-based checking is done by the interconnect. For example, the Arm NIC-400 allows system designers to specify for each connected completer:

- Secure:

  Only Secure accesses are passed to device. Interconnect generates a fault for all Non-secure accesses, without the access being presented to the device.

- Non-secure:

  Only Non-secure accesses are passed to device. Interconnect generates a fault for all Secure accesses, without the access being presented to the device.

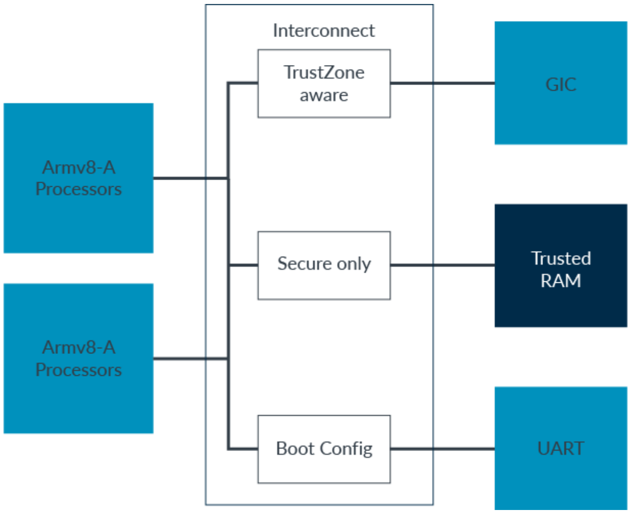
- Boot time configurable

  At boot time, system initialization software can program the device as Secure or Non-secure. The default is Secure.

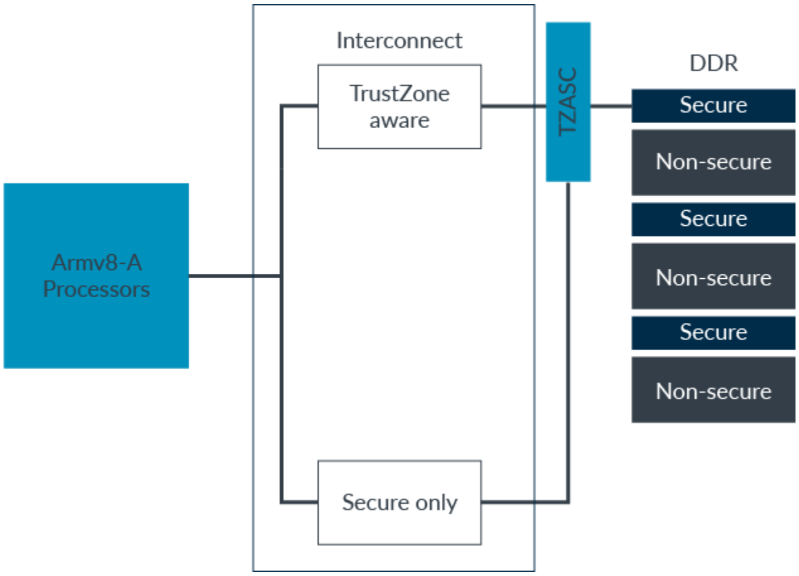
- TrustZone aware

  The interconnect allows all accesses through. The connected device must implement isolation.

For example:



This approach works well for either TrustZone-aware devices or those devices that live entirely within one address space. For larger memories, like off-chip DDR, we might want to partition the memory into Secure and Non-secure regions. A TrustZone Address Space Controller (TZASC) allows us to do this, as you can see in the following diagram:



The TZASC is similar to a Memory Protection Unit (MPU), and allows the address space of a device to split into several regions. With each region specified as Secure or Non-secure. The registers to control the TZASC are Secure access only, permitting only Secure software to partition memory.

An example of a TZASC is the Arm TZC-400, which supports up to nine regions.

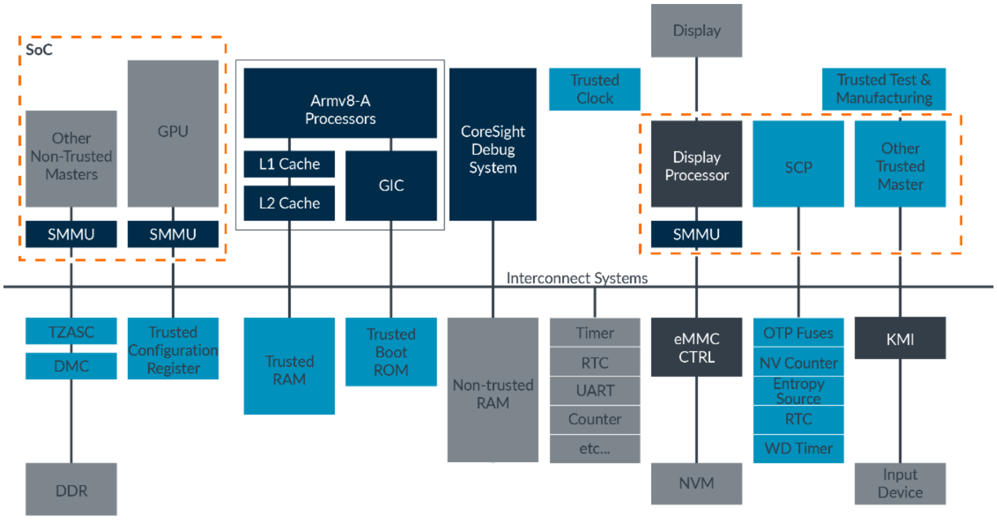
Note

Off-chip memory is less Secure than on-chip memory, because it is easier for an attacker to read or modify its contents. On-chip memories are more secure but are much more expensive and of limited size. As always, we must balance cost, usability, and security. Be careful when deciding which assets you want in off -chip memories and which assets need to be kept on-chip.

When the Armv9-A Realm Management Extension (RME) is implemented, memory can by dynamically moved between physical address spaces via the Granule Protection Table. For more information, see Introducing Arm's Dynamic TrustZone technology blog.

4.3 Bus requesters

Next, we will look at the bus requesters in the system, as you can see in the following diagram:



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, we can roughly divide the requester devices in the system into groups:

- TrustZone aware

  Some requesters are TrustZone aware, and like the processor, provide the appropriate security information with each bus access. Examples of this include System MMUs (SMMUs) that are built to the Arm SMMUv3 specification.

- 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.

What system resources do non-TrustZone-aware requesters need to access? Based on the answer to this question, we could pick one of several approaches:

- Design time tie-off

  Where the requesters only needs to access a single physical address space, a system designer can 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

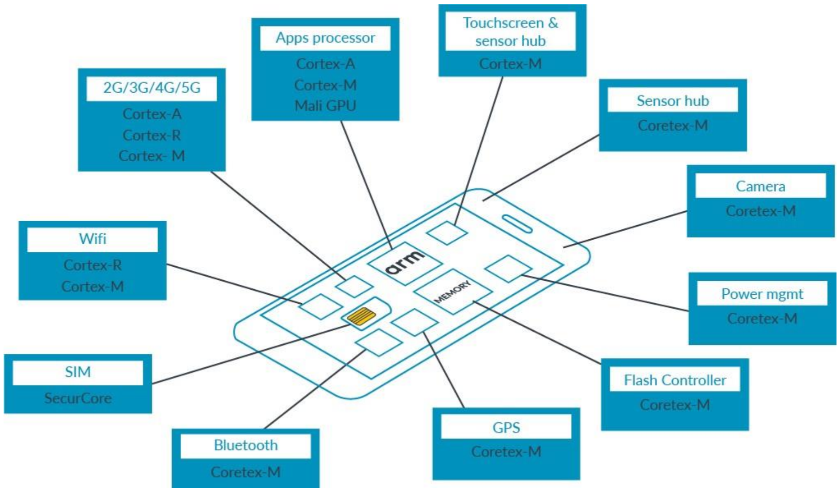
  Logic is provided to add the security information to the requester's bus accesses. Some interconnects, like the Arm NIC-400, provide registers that Secure software can use at boot time to set the security of an attached requester accesses. This overrides whatever value the requester provided itself.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

- SMMU

  A more flexible option is an SMMU. 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.

4.4 M and R profile Arm processors

Many modern designs include a mixture of A-profile, R-profile, and M-profile processors. For example, a mobile device might have an A-profile processor to run the mobile OS, an R-profile processor for the cellular modem, and an M-profile processor for low-level system control. The following diagram shows an example mobile device and the different processors that you might find:

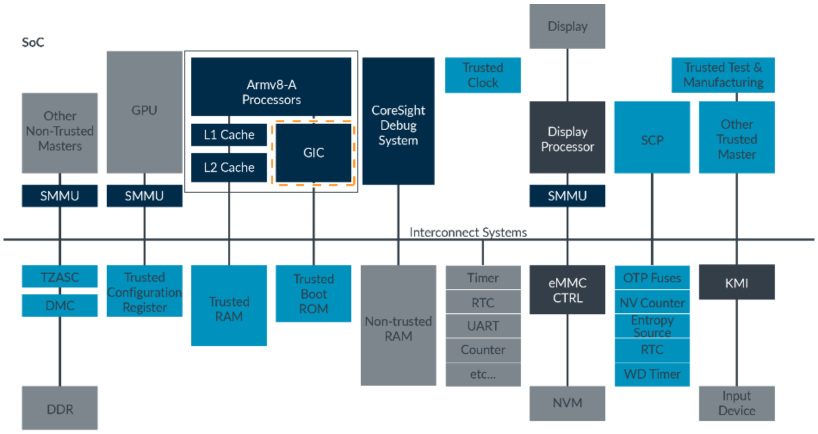


The R profile does not support the two Security states in the way that the A profile does. This means that software running on those processors cannot control the outputted physical address space. In this way, they behave much like other non-TrustZone aware bus requesters. The same is true for M profile processors that do not implement TrustZone for Armv8-M.

Often these processors only need to access a single physical address space. Using our example of a mobile device, the processors typically include an M-profile processor for low-level system control. This is sometimes called a System Control Processor (SCP). In many systems, the SCP would be a Secure-only device. This means that it only needs the ability to generate bus secure accesses.

4.5 Interrupts

Next, we will look at the interrupts in the system, as you can see in the following diagram:



The Generic Interrupt Controller (GIC), supports TrustZone. Each interrupt source, called an INTID in the GIC specification, is assigned to one of three Groups:

- Group 0: Secure interrupt, signaled as FIQ
- Secure Group 1: Secure interrupt, signaled as IRQ or FIQ
- Non-secure Group 1: Non-secure interrupt, signaled as IRQ or FIQ

This is controlled by software writing to the `GIC[D|R]_IGROUPR<n>` and `GIC[D|R]_IGRPMODR<n>` registers, which can only be done from Secure state. The allocation is not static. Software can update the allocations at run-time.

For INTIDs that are configured as Secure, only bus secure accesses can modify state and configuration. Register fields corresponding to Secure interrupts are read as 0s to Non-secure bus accesses.

For INTIDs that are configured as Non-secure, both Secure and Non-secure bus accesses can modify state and configuration.

Why are there two Secure Groups? Typically, Group 0 is used for interrupts that are handled by the EL3 firmware. These relate to low-level system management functions. Secure Group 1 is used for all the other Secure interrupt sources and is typically handled by the S.EL1 or S.EL2 software.

4.6 Handling interrupts

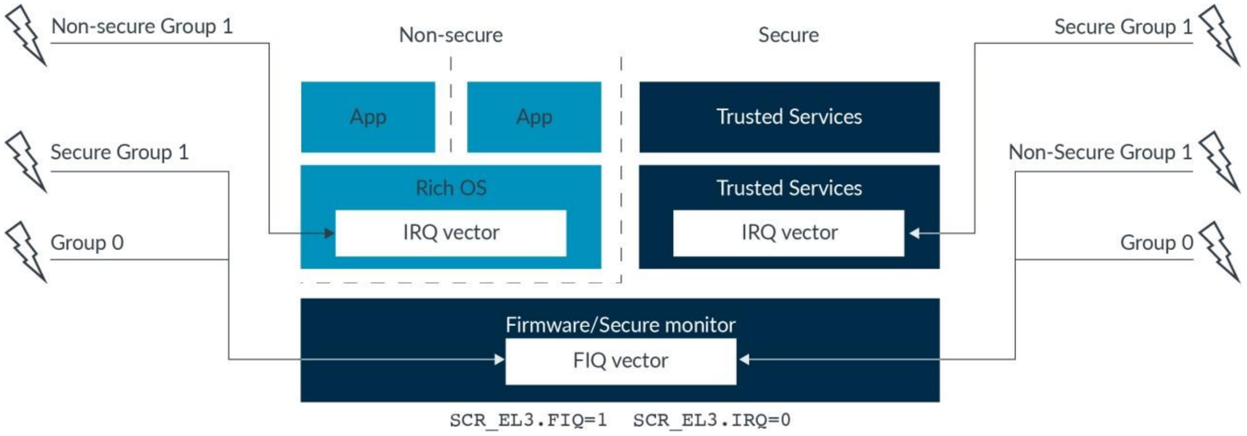
The processor has two interrupt exceptions, IRQ and FIQ. When an interrupt becomes pending, the GIC uses different interrupt signals depending on the group of the interrupt and the current Security state of the processor:

- Group 0 interrupt
    - Always signaled as FIQ exception
- Secure Group 1
    - Processor currently in Secure state – IRQ exception
    - Processor currently in Non-secure state – FIQ exception
- Non-secure Group 1
    - Processor currently in Secure state – FIQ exception
    - Processor currently in Non-secure state – IRQ exception

Remember that Group 0 interrupts are typically used for the EL3 firmware. This means that:

- IRQ means a Group 1 interrupt for the current Security state.
- FIQ means that we need to enter EL3, either to switch Security state or to have the firmware handle the interrupt.

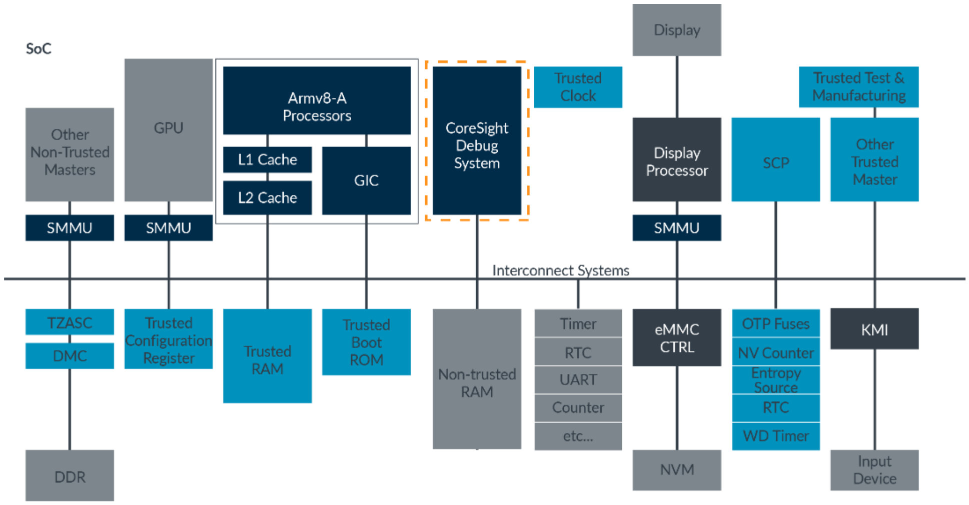
The following example shows how the exception routing controls could be configured:



The preceding diagram shows one possible configuration. Another option that is commonly seen is for FIQs to be routed to EL1 while in Secure state. The Trusted OS treats the FIQ as a request to yield to either the firmware or to Non-secure state. This approach to routing interrupts gives the Trusted OS the opportunity to be exited in a controlled manor.

4.7 Debug, trace, and profiling

Next, we will look at the debug and trace components in the system, as you can see in the following diagram:



Modern Arm systems include extensive features to supporting debugging and profiling. With TrustZone, we must ensure that these features cannot be used to compromise the security of the system.

Regarding debug features, consider the development of a new SoC. Different developers are trusted to debug different parts of the system. The chip company engineers need, and are trusted to, debug all parts, including the Secure state code. Therefore, all the debug features should be enabled.

When the chip ships to an OEM, they still need to debug the Non-secure state software stack. However, the OEM might be prevented from debugging the Secure state code.

In the shipping product containing the chip, we might want some debug features for application developers. But we also want to limit the ability to debug the code of the silicon provider and the OEM.

Signals to enable the different debug, trace, and profiling features help us deal with this situation. This includes separate signals to control use of the features in Secure state and Non-secure state.

Continuing with the debug example, these signals include:

- `DBGEN` – Top-level invasive debug enable, controls external debug in both Security states
- `SPIDEN` – Secure Invasive Debug Enable, controls external ability to debug in Secure state

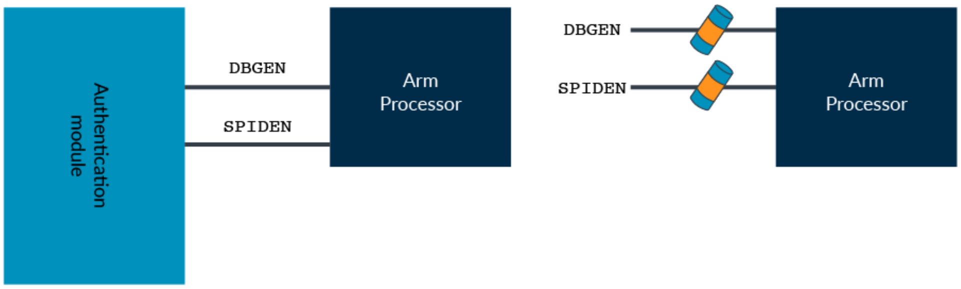
Note

These two signals are examples. There are other debug authentication signals. Refer to the Technical Reference Manual of your processor for a complete list.

Here is an example of how we might use these signals:

- Early development by chip designer
  - `DGBEN==1` and `SPIDEN==1`, enabling full external debug
- Product development by OEM
  - DBGEN==1, enabling external debug in Non-secure state
  - SPIDEN==0, disabling debug in Secure state
- Shipping product
  - DGBEN==0 and SPIDEN==0, disabling external debug in both Security states
  - Debug of applications still possible

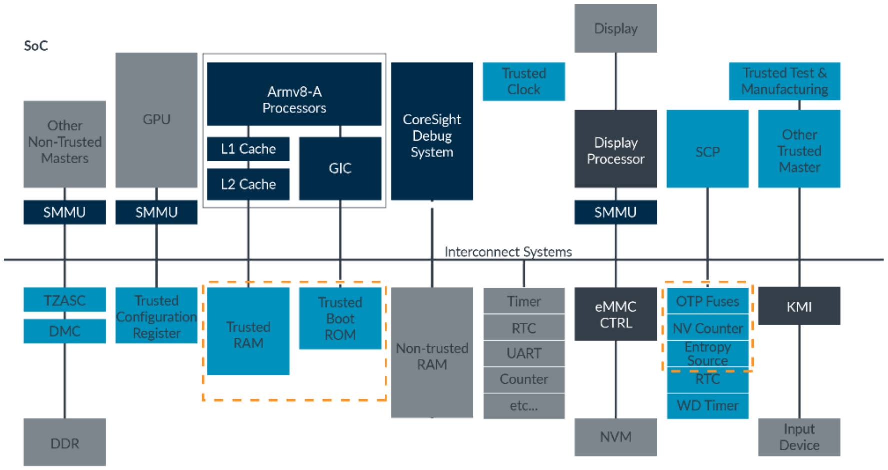
Because we want different signal values at different stages of development, it is common to connect the signals using e-fuses or authentication blocks. Here is an example:



By blowing fuses during manufacture, external debug can be permanently disabled. Using fuses does make in-field debug more difficult. When the fuses are blown, they cannot be unblown. An authentication module is more flexible.

4.8 Other devices

Finally, we will look at the other devices in the system, as you can see in the following diagram:



Our example TrustZone-enabled system includes several devices which we have not yet covered, but which we need to build a practical system.

- One-time programmable memory (OTP) or fuses

  These are memories that cannot be changed once they are written. Unlike a boot ROM which contains the same image on each chip, the OTP can be programmed with device unique values and possibly OEM unique values.

  One of the things that is stored in OTP is a device unique private key. When each chip is manufactured, a randomly generated unique key is written to the OTP. This device unique private key is used to tie data to the chip.

  The advantage of a device unique private key is that it prevents class attacks. If each chip had the same key, if one device is compromised then all similar devices would also be vulnerable.

  OTP is also often used to store hashes of OEM public keys. OTP is relatively expensive compared to other memories. For public keys, only storing the hash and not storing the full key saves cost.

- Non-volatile counter

  Non-volatile (NV) counter, which might be implemented like more fuses. This is a counter that can only increase and can never be reset.

  NV counters are used to protect against rollback attacks. Imagine that there is a known vulnerability in version 3 of the firmware on a device. The device is currently running version 4, on which the vulnerability is fixed. An attacker might try to downgrade the firmware back to version 3, to exploit the known vulnerability. To protect against this, each time the firmware is updated the count is increased. At boot, the version of the firmware is checked against the NV counter. If there is mismatch, the device knows that it is being attacked.

- Trusted RAM and Trusted ROM

  These are on-chip Secure access only memories.

The Trusted ROM is where the first boot code is fetched from. Being on-chip means that an attacker cannot replace it. Being a ROM means that an attacker cannot reprogram it. This means that we have a known, trusted, starting point of execution, which will be discussed in the Software architecture section of this guide.

The Trusted RAM is typically an SRAM of a couple of hundred kilobytes. This is the working memory of the software that is running in Secure state. Again, being on-chip makes it difficult for an attacker to gain access to its content.

4.9 Trusted Base System Architecture

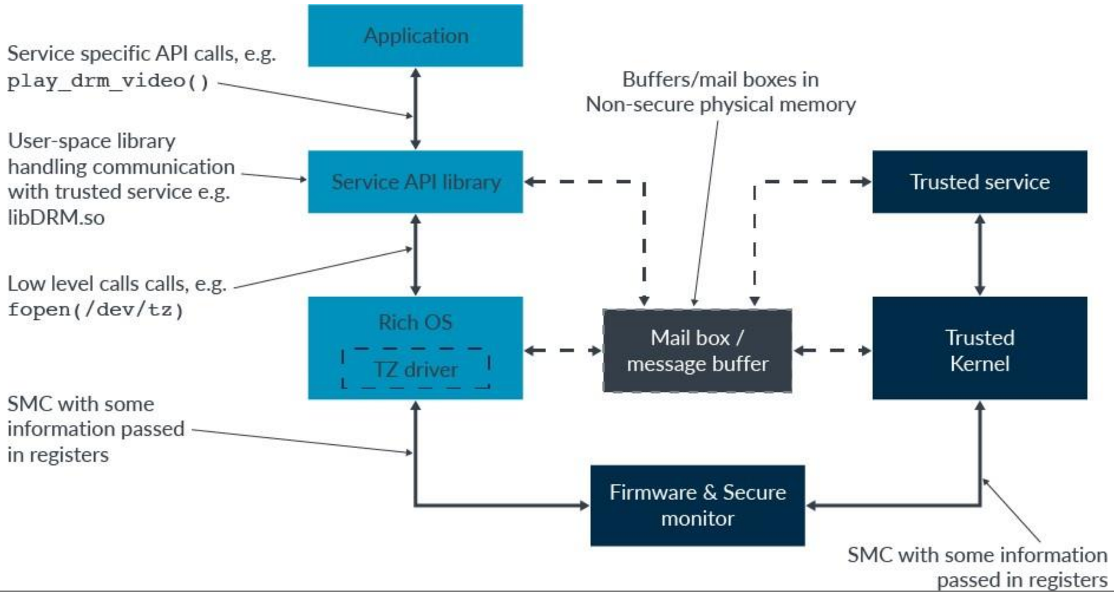
The Trusted Base System Architecture (TBSA) is a set of guidelines from Arm for system designers. TBSA provides recommendations on what resources different use cases require, for example, how many bits of OTP are required.

5 Software architecture

In TrustZone in the processor and System architecture, we explored TrustZone support in hardware, both the Arm processor and wider memory system. This topic looks at the software architecture that is found in TrustZone systems.

5.1 Top-level software architecture

The following diagram shows a typical software stack for a TrustZone enabled system:


Application
Service specific API calls, e.g. play_drm_video()
Buffers/mail boxes in Non-secure physical memory
User-space library handling communication with trusted service e.g. libDRM.so
Service API library
Trusted service
Low level calls calls, e.g. fopen(/dev/tz)
Rich OS
TZ driver
Mail box / message buffer
Trusted Kernel
SMC with some information passed in registers
Firmware & Secure monitor
SMC with some information passed in registers

Note

For simplicity, the diagram does not include a hypervisor, although they might be present.

The Trusted kernel in Secure state hosts services, like key management or DRM. Software running in Non-secure state needs to have controlled accesses to those services.

A user-space application is unlikely to be directly aware of TrustZone. Instead it would use a high-level API that is provided by a user-space library. That library handles communication with the Trusted service. This is similar to how, for example, a graphics API provides abstraction from the underlying GPU.

Communication between the service library and the Trusted service is typically handled using message queues or mailboxes in memory. The term World Shared Memory (WSM) is sometimes used to describe memory that is used for this communication. These queues must be in memory that both sets of software can see, which means Non-secure memory. This is because Non-secure state can only see Non-secure memory.

The service library places a request, or requests, in the mailbox and then invokes a driver in kernel space. The driver is responsible for low-level interactions with the Trusted Execution Environment

(TEE), which could include allocating the memory for the message queues and registering them with the TEE. Remember that the two worlds are operating in different virtual address spaces, therefore they cannot use virtual addresses for communication.

The driver would call Secure state, typically using an SMC. Control would pass through the EL3 Secure Monitor to the Trusted Kernel in the TEE. The kernel invokes the requested service, which can then read the request from the queue.

5.2 Trusting the message?

In the flow that we have just described, the requests sit in a queue that is located in Non-secure memory. What if:

- The application that made the initial request is malicious?
- Other malicious software substituted the messages in the queue?

The TEE must assume that any request or data that is provided from Non-secure state might be malicious or in some other way invalid. This means that authenticating the request, or requestor, needs to be done in Secure state.

What this looks like will depend on the Trusted service being provided and its security requirements. There is no one single answer.

5.3 Scheduling

In a TrustZone system there are two software stacks, one for Non-secure state and another for Secure state. A processor core can only be in one state at a time. Who decides when each world is allowed to run?

Explicit calls to the EL3 firmware, like power management requests using Power State Coordination Interface (PSCI), are typically blocking. This means that control will only be returned to Non-secure state when the requested operation is complete. However, these calls tend to be short and infrequent.

The TEE typically runs under the control of the Non-secure state OS scheduler. A possible design is to have a daemon running under the OS as a place holder for the TEE. When the daemon is scheduled by the OS, the daemon hands control to the TEE through an SMC. The TEE then runs, processing outstanding requests, until the next scheduler tick or interrupt. Then control returns to the Non-secure state OS.

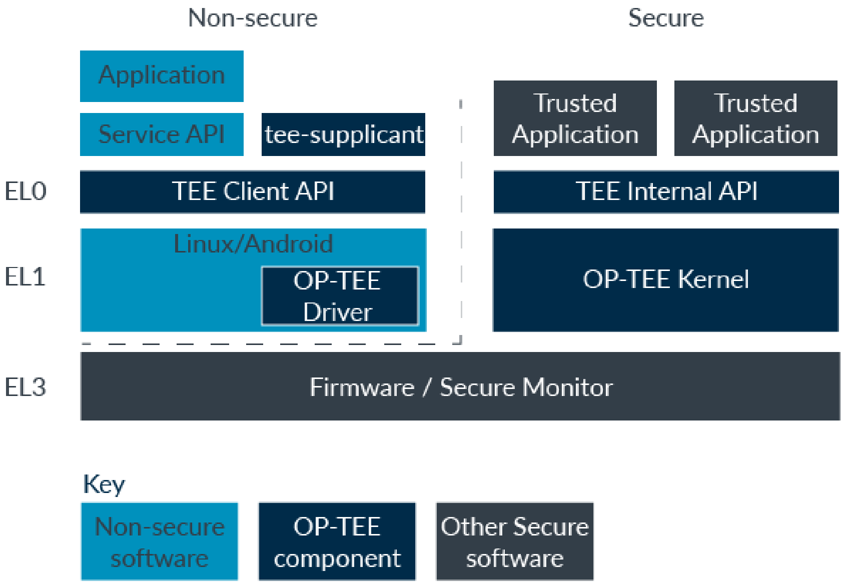
This might seem odd, because this approach gives the untrusted software control over when Trusted software can execute, which could enable denial of service attacks. However, because the TEE provides services to Non-secure state, preventing it from running only prevents those services from being available. For example, an attacker could prevent a user from playing a DRM-protected video. Such an attack does not cause any information to be leaked. This type of design can ensure confidentiality but not availability.

We could design the software stack to also give availability. The GIC allows Secure interrupts to be made higher priority than Non-secure interrupts, preventing Non-secure state from being able to block the taking of a Secure interrupt.

5.4 OP-TEE

There are many Trusted kernels, both commercial and open source. One example is OP-TEE, originally developed by ST-Ericsson, but now an open- source project hosted by Linaro. OP-TEE provides a fully featured Trusted Execution Environment, and you can find a detailed description on the OP-TEE project website.

The structure of OP-TEE is shown in the following diagram:



The OP-TEE kernel runs in S.EL1, hosting Trusted applications in S.EL0. The Trusted applications communicate with the OP-TEE kernel through the TEE Internal API. The TEE Internal API is a standard API developed by the GlobalPlatform group. GlobalPlatform work to develop standard APIs, which are supported by many different TEEs, not just OP-TEE.

Note

In the preceding diagram, the Trusted applications are not shown as OP -TEE components. This is because they are not part of the core OP-TEE OS. The OP-TEE project does provide some example Trusted Applications for people to experiment with.

In Non-secure state, there is a low-level OP-TEE driver in kernel space. This is responsible for handling the low-level communication with the OP- TEE kernel.

In the Non-secure user space (EL0), there is a user-space library implementing another GlobalPlatform API. The TEE Client API is what applications use to access a Trusted application or service. In most cases, we would not expect an application to use the TEE Client API directly. Rather there would be another service-specific library providing a higher-level interface.

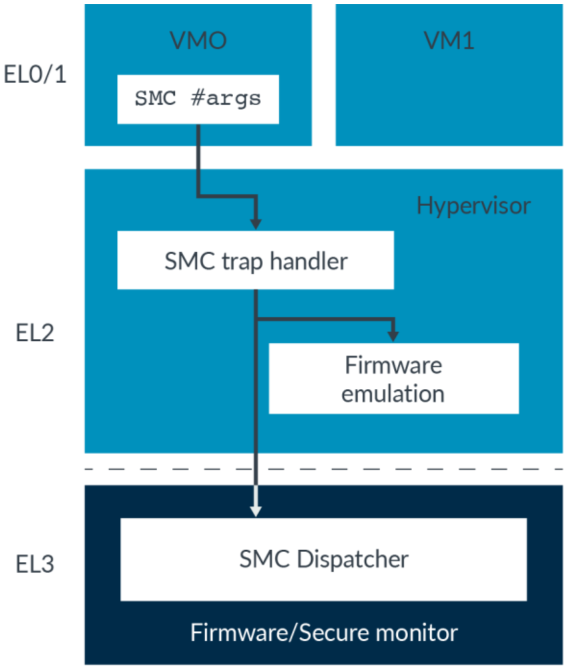
OP-TEE also includes a component that is called the tee-supplicant. The tee-supplicant handles services that are supported by OP-TEE and require some level of rich OS interaction. An example is secure storage.

5.5 Interacting with Non-secure virtualization

In the examples that we have covered so far, we have ignored the possible presence of a hypervisor in Non-secure state. When a hypervisor is present, much of the communication between a VM and Secure state will be through the hypervisor.

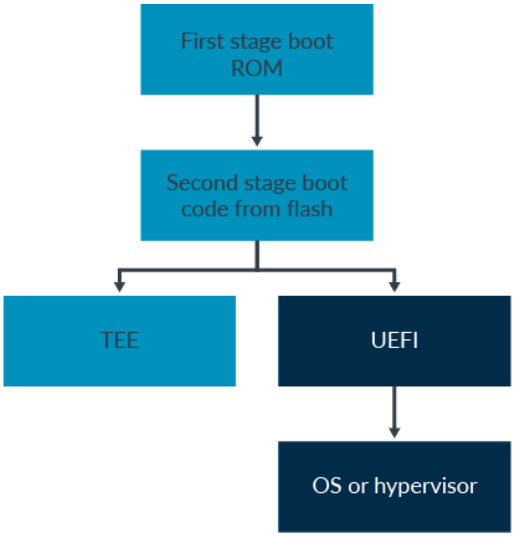
For example, in a virtualized environment SMCs are used to access both firmware functions and Trusted services. The firmware functions include things like power management, which a hypervisor would typically not wish to allow a VM to have direct access to.

The hypervisor can trap SMCs from EL1, which allows the hypervisor to check whether the request is for a firmware service or a Trusted service. If the request is for a firmware service, the hypervisor can emulate the interfaces rather than passing on call. The hypervisor can forward Trusted service requests to EL3. You can see this in the following diagram:



5.6 Boot and the chain of trust

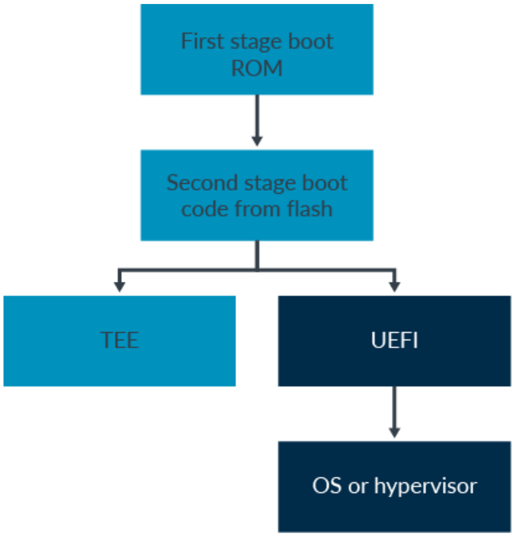
Boot is a critical part of any TrustZone system. A software component can only be trusted if we trust all the software components that ran before it in the boot flow. This is often referred to as the chain of trust. A simplified chain of trust is shown in the following diagram:



In our example, the first code that runs is the boot ROM. We must implicitly trust the boot ROM, because there are no earlier stages of boot to verify its contents. Being in ROM protects the initial boot code from being rewritten. Keeping the initial boot code on-chip prevents it from being replaced, so we can implicitly trust it. The boot ROM code is typically small and simple. Its main function is to load and verify the second stage boot code from flash.

The second stage boot code performs system initialization of the platform, like setting up the memory controller for off-chip DRAM. This code is also responsible for loading and verifying the images that will run in Secure and Non-secure state. Examples include loading a TEE in Secure state and higher-level firmware like UEFI in Non-secure state.

Earlier we introduced the System Control Processor (SCP). An SCP is a microcontroller that performs low-level system control in many modern SoCs. Where an SCP, or similar, is present it also forms part of the chain of trust. The following diagram shows this:



5.7 Boot failures

In a Trusted boot system, each component verifies the next component before it loads, forming a chain of trust. Let us look now at what happens when verification fails.

There is no one answer for this situation. It depends on the security needs of the system and which stage of the boot processor the failure occurs at. Consider the example of an SoC in a mobile device. If the verification failed at:

- Second stage boot image

  The second stage boot image is required for initialization of the SoC and processor. If verification fails at this stage, we might not be sure that the device can boot safely and function correctly. Therefore, if verification fails at this stage it is usually fatal and the device cannot boot.

- TEE

  The TEE provides services, like key management. The device can still function, perhaps at a limited level, without the TEE being present. Therefore, we could choose to not load the TEE, but still allow the Non-secure state software to load.

- Non-secure state firmware or Rich OS image

  The Non-secure state software is already at a lower level of trust. We might choose to allow it to boot, but block accesses to advanced features provided via the TEE. For example, a TrustZone-enabled DRM might not be available with an untrusted OS image.

These are just examples. Each system needs to make its own decisions based on its security requirements.

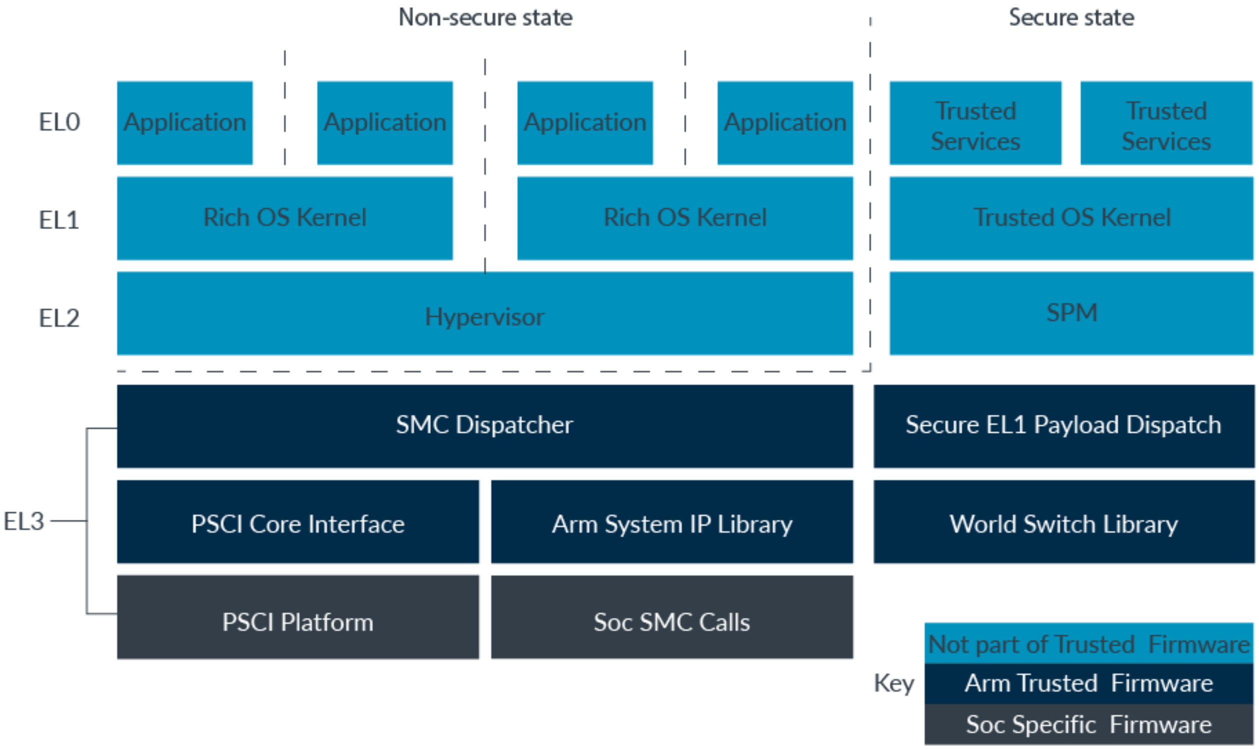
5.8 Trusted Board Boot Requirements

Earlier we introduced the Trusted Base System Architecture (TBSA), which is guidance for system designers. The Trusted Board Boot Requirements (TBBR) are a similar set of guidelines for software developers. TBBR gives guidance on how to construct a Trusted boot flow in a TrustZone-enabled system.

5.9 Trusted Firmware

Trusted Firmware is an open-source reference implementation of Secure world software for Armv8-A devices. Trusted Firmware provides SoC developers and OEMs with a reference Trusted code base that complies with the relevant Arm specifications, including TBBR and SMCC.

The following diagram shows the structure of the Trusted Firmware:



The SMC dispatcher handles incoming SMCs. The SMC dispatcher identifies which SMCs should be dealt with at EL3, by Trusted Firmware, and which SMCs should be forwarded the Trusted Execution Environment.

The Trusted Firmware provides code for dealing with Arm system IP, like interconnects. Silicon providers need to provide code for handling custom or third-party IP. This includes SoC-specific power management.

6 Example use cases

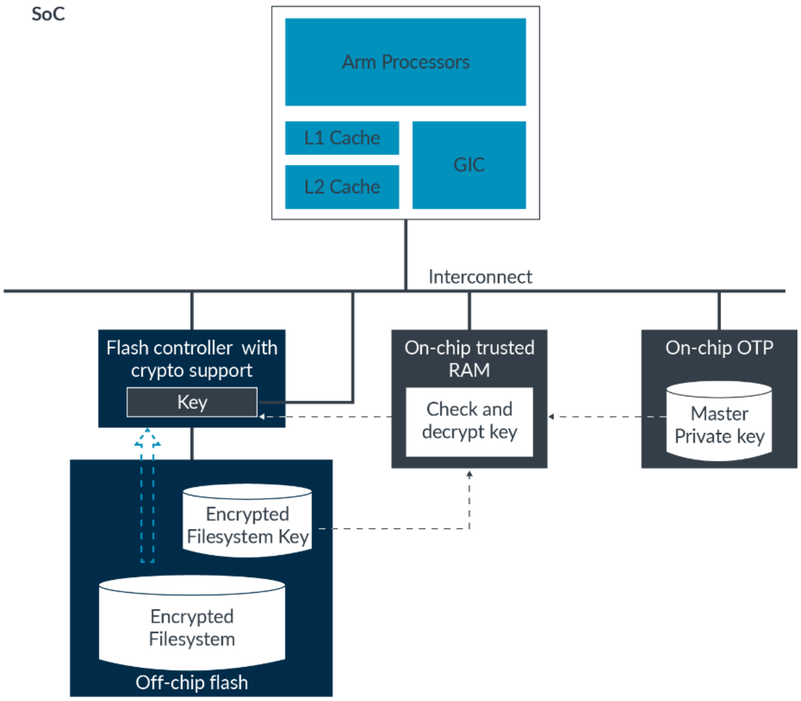
In TrustZone in the processor and System Architecture, we introduced the TrustZone features in hardware and discussed the typical software stack that uses those features. In this topic, let's pull together this knowledge and look at some example use cases.

6.1 Encrypted filesystem

Mobile devices like smartphones contain a lot of personal data. Users care about the confidentiality of that data if the device is lost or stolen. This is why most recent devices support file system encryption. TrustZone can be used part of the solution for protecting this data.

Data stored in the external flash is encrypted. On boot, the device authenticates the user and then provisions the key to decrypt the filesystem. Decryption might be handled by an accelerator or might be integrated into the flash controller.

The key for the file system also needs to have its confidentiality protected. If the key is compromised, an attacker could decrypt the filesystem. The processes after authentication are illustrated in the following diagram:



In Secure state:

- After authentication, the encrypted filesystem key is read into on-chip secure memory. The key is decrypted and checked, using the requester device unique key, which is stored on-chip.
- The filesystem key is provisioned into a secure access only register in a crypto engine or memory controller.

- Subsequent bus accesses to the filesystem in flash will be encrypted or decrypted using the provisioned key.

By performing these operations in Secure state, TrustZone allows us to never expose the filesystem keys to Non-secure state software. This means that malicious code in Non-secure cannot extract those keys for later attacks. Using what we have discussed so far, think about the following questions:

Q: Why is the filesystem key stored off-chip?

A: On-chip memory tends to be limited in size and expensive, compared to off-chip flash. Keeping the filesystem key off-chip can reduce cost. Having it encrypted means that we ensure confidentiality. There is a risk that malicious software could corrupt the key, which would be a breach of integrity, but that does not expose data.

Q: Why do we use a separate filesystem key in this example, and not the requester device unique private key?

A: In theory, we could use the device unique key. But that means that we can never change the key, because the requester device unique private key is stored in OTP. That might be a problem if, for example, we sell the phone. Instead we generate a new random filesystem key. If you want to format or reset the device, we can delete the filesystem key and generate a new one. Any data that is encrypted with the old key is now irretrievable.

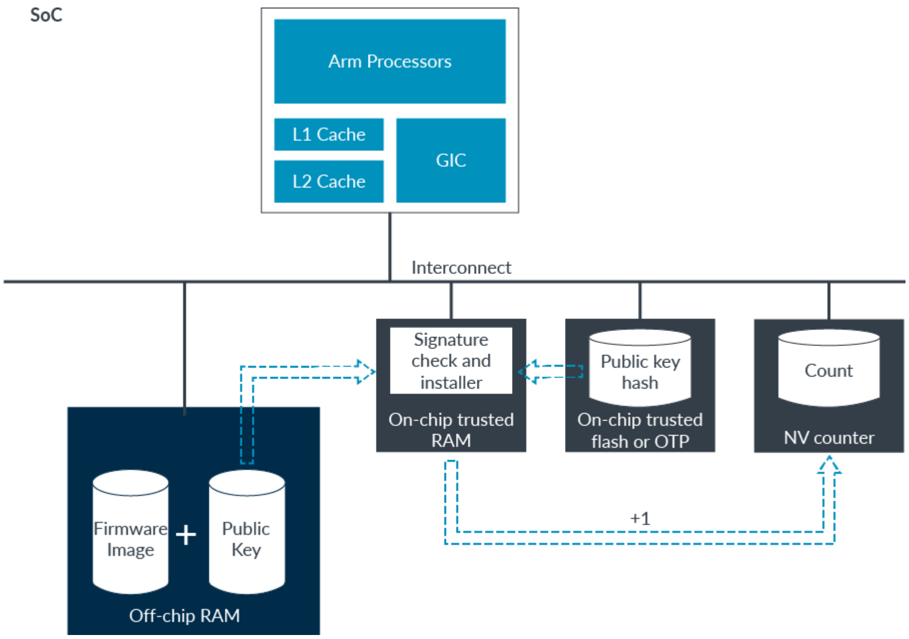
6.2 Over the air firmware update

This second example relates to updating the boot firmware. The requirements for our system are:

- New firmware image is provided over a network.
- Only authentic images can be installed.
- Firmware version cannot be rolled back.

To achieve these aims, the OEM signs the image with its private key. The downloading device is provisioned with the public key, which it can use to verify the signature. A non-volatile counter is incremented when the firmware is updated, allowing detection of roll-back.

Our system is shown in the following diagram:



The downloading of the image is carried out in Non-secure state. The image itself is a not a secret, so we do not protect its confidentiality. The downloaded image is placed in memory and a request is issued to Secure state to install it.

The Secure state software is responsible for authentication. It does this using the public key of the OEM, typically stored in off-chip flash. This key is not a secret, so we do not need to ensure confidentiality. We do need to ensure authenticity of the key and detect attempts to substitute the key.

We achieve this by keeping a hash of the key on-chip, which can be used to check the key when needed. A hash requires fewer bits and on-chip memory is expensive.

When the public key is loaded and checked, the new firmware image can be checked. We want to ensure that it is authentic (the signature matches) and that it is a newer version of the firmware than what is installed.

Assuming that these checks pass, the image is installed, and the NV counter incremented. Incrementing the NV counter means that, if an attacker tries to install an older firmware, the device will detect that attempt.

In this example, TrustZone allows us to ensure that the keys that are used to authenticate firmware images are protected and that firmware images cannot be rolled back.

7 Check your knowledge

1. What are the Security states and physical address spaces in the Arm architecture?

   A. The Security states in the Arm architecture are Secure state and Non-secure state. The physical address spaces in the Arm architecture are the Secure physical address space and the Non-secure physical address space.

2. For each Exception level, what determines whether the processor is in Secure state or Non-secure state?

   A. For EL0/1/2, the `SCR_EL3.NS` bit. EL3 is always in Secure state.

3. While in Non-secure state, can software access the Secure physical address space?

   A. No. While in Non-secure state, virtual addresses always map to Non-secure physical addresses.

4. Can an access to SP:0x80000 hit on a cache line containing NP:0x80000?

   A. No. SP:0x80000 are NP:0x80000 are different locations, so there is no cache hit.

5. What do Trusted Base System Architecture (TBSA) and Trusted Board Boot Requirements (TBBR) provide guidance on?

   A. TBBR gives guidance on booting, and TBSA gives guidance on system architecture.

6. What is the purpose of a TrustZone Address Space Controller (TZASC)?

   A. A TZASC allows a memory to be partitioned into Secure and Non-secure regions.

8 Related information

Here are some resources related to material in this guide:

- Arm architecture and reference manuals: Find technical manuals and documentation relating to this guide and other similar topics.
- Arm Community: Ask development questions, and find articles and blogs on specific topics from Arm experts
- Isolation using virtualization in the Secure world: Learn more about Secure virtualization in this white paper
- Arm CoreLink Generic Interrupt Controller v3 and v4 Guide
- Silicon IP Security: Find more information on Trusted Base System Architecture.
- TrustZone for Cortex-A
- TrustZone for Cortex-M
- Introducing Arm's Dynamic TrustZone technology

Here are some resources related to topics in this guide:

OP-TEE

- OP-TEE is an example of a trusted execution environment. OP-TEE is an open-source project.
- OP-TEE implements industry standard APIs that are developed and maintained by the Global Platform group. For information on these APIs, see the Global Platform Specification Library.
- You can experiment with the Trusted Firmware and OP-TEE on the free Arm Foundation model, or on the FVP models that are provided with Arm Development Studio. Here is more information about building and running Arm Reference Platforms.

SMC Exceptions

The following specifications describe how SMCs are used to request services:

- SMC Calling Convention (DEN0028)
- Power State Coordination Interface specification (DEN002)

Trusted Board Boot Requirements

- Trusted Firmware: Find some example code for dealing with different types of Arm System IP, for example interconnects.

Trusted Firmware

- Trusted Firmware: Find some example code for dealing with different types of Arm System IP, for example interconnects.

9 Next steps

This guide has introduced the TrustZone security architecture, which provides isolation between two worlds or execution environments. The Arm architecture also has features for providing robust security within a given environment. To learn more, read our guide on Security – Providing robust protection for complex software.

In this guide we have mentioned the topics of virtualization and GICs without fully exploring them. To learn more about these topics, read these guides in our series:

- Arm CoreLink Generic Interrupt Controller v3 and v4 Guide
- Armv8-A Virtualization
- This guide does not cover the Armv9-A Realm Management Extension (RME). For more information on RME, see Realm Management Extension guide.