# Learn the architecture - Arm Confidential Compute Architecture software stack

Version 2.0

**Non-Confidential**

**Issue 05**
den0127_0200_05_en

# Learn the architecture - Arm Confidential Compute Architecture software stack

## Release information

### Document history

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| 0100-02 | 20 September 2021 | Non-Confidential | Initial release |
| 0100-03 | 13 September 2022 | Non-Confidential | Minor update |
| 0100-04 | 9 May 2023 | Non-Confidential | Minor update |
| 0200-05 | 29 June 2023 | Non-Confidential | Change the title |

## Proprietary Notice

## 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.

## Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

## Inclusive language commitment

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

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

# Contents

# 1. Overview

This guide describes the firmware and software components which are part of the Arm Confidential Compute Architecture (Arm CCA).

In this guide, you will learn how to:

- List the set of components which make up the Arm CCA software stack
- Understand the reasons why Arm CCA introduces new software components
- Understand the roles of the Monitor and the Realm Management Monitor (RMM)
- Understand how Realms are created and managed

## 1.1 Before you begin

We assume that you are familiar with the AArch64 Exception model, AArch64 memory management, AArch64 virtualization, and the fundamentals of Arm CCA. Information on these topics can be found in the following guides:

- AArch64 exception model introduces the exception and privilege model of AArch64.
- AArch64 memory management introduces the Arm Virtual Memory System Architecture (VMSA).
- AArch64 virtualization describes virtualization support in Armv8-A AArch64. Topics include stage 2 translation, virtual interrupts, and trapping.
- Introducing Arm Confidential Compute Architecture describes the motivation and requirements for confidential compute and provides an overview of how these requirements are solved by Arm CCA.

Some features of the Realm Management Extension (RME), which is the hardware component of Arm CCA, can also be used for purposes other than hosting Realms. Non-Realm use cases are out of scope of this document. For further details about how RME can enable more flexible management of Secure memory, see Introducing Arm's dynamic TrustZone technology.

## 1.2 Arm CCA goals

The primary goal of Arm CCA is to retain the ability of existing system software such as hypervisors to manage hardware resources required by virtual machines (VMs), while providing protection to data in use inside the VMs. To enable this protection, Arm CCA introduces the concept of a protected VM, called a Realm, and prevents the hypervisor and other privileged software and hardware agents from observing or modifying the contents of a Realm. In this way, Arm CCA separates the right of management from the right of access, and grants only the first of these to the hypervisor with respect to Realms.

The removal of right of access means that the owner of a Realm does not need to trust the hypervisor. However, the owner does still need to trust a carefully defined set of other software components, which is described in Software components. A goal of Arm CCA is to enable these components to be as small and simple as possible, thereby simplifying the task of reasoning about the correctness of their implementation.

It is important to point out that while Arm CCA provides confidentiality and integrity guarantees to a Realm, there is no corresponding guarantee of availability. While a hypervisor cannot access the internal state of a Realm, it can still decide not to schedule its virtual CPUs (VCPUs).

Arm CCA aims to avoid the imposition of arbitrary limits on the resources which can be allocated for use by Realms. At a first approximation, if a system is capable of hosting a given set of normal, unprotected VMs, then it should be possible instead to host a corresponding set of Realms.

Finally, Arm CCA aims to minimize the effort required to port an existing workload from a normal VM to a Realm. Some amount of enlightenment and hardening is however necessarily required, to adapt the software to the environment of a Realm and to take full advantage of the attestation mechanisms that Arm CCA provides.

## 1.3 Hardware-software split

A key consideration in the design of a technology such as Arm CCA is the split between hardware and software. At one extreme, solutions could exist which make no change to existing hardware architecture and deliver the required functionality entirely in software. At the other, designing all the necessary primitives as hardware architecture extensions would also be a solution.

Looking at the existing hardware architecture as of Armv8.4-A, the zero hardware change extreme can be ruled out. At a minimum, an architected mechanism for dynamically protecting memory by moving it from the Non-secure to the Secure physical address space is required. With this feature added, then the Secure virtualization extensions would allow us to create an execution environment in Secure EL1 / EL0. This would have hardware-backed protection from software in the Non-secure state, including the hypervisor. However, on many platforms, the Secure state is already used to provide a range of platform services. The requirement for mutual isolation between Secure state software and confidential compute environments means that this approach is not acceptable.

Extending the architecture of the Non- secure state could be considered, adding features which allow a VM to be protected from the hypervisor. However, this approach suffers from two problems. The first is that it would require a feature-by-feature study of the interaction between the confidential compute extensions, and the existing architecture. As an example, debug registers are explicitly designed to allow the hypervisor to peek inside a VM. This behavior conflicts with the requirements of confidential computing and would therefore need to be redesigned. Scaling this up over the entire Arm A-class architecture would be a hugely complex task.

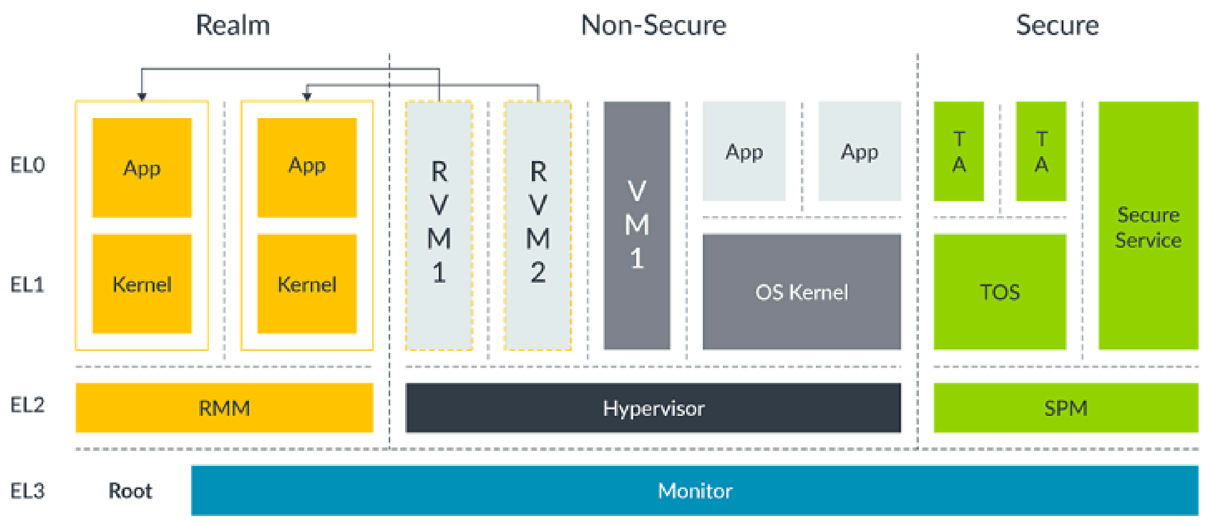
The second problem is that the complex nature of many of the primitives required for confidential computing goes against the RISC design of the Arm architecture. These instructions would likely need to be implemented using microcode, which does not fit well with the implementation style of most Arm processors.

Arm CCA chooses a middle ground between these two extremes. The Realm Management Extension (RME) provides hardware primitives in the form of new Processing Element (PE) security states and mechanisms for protecting memory in a fine-grained and dynamic manner. The software components described in this guide make use of the hardware primitives to create and protect confidential compute environments. By keeping the hardware changes to a minimum, and by re-using existing architectural concepts such as Exception levels and security states, the task of reasoning about the correctness of the RME is simplified. By implementing most of the Realm management logic in software, this is enabled to be delivered in a transparent and auditable way and make it easier to deploy bug fixes and errata workarounds.

# 2. Software components

In this section, you will learn about the software components of Arm CCA, including the Realm World and the Monitor Root world. The following diagram shows the software components in an Arm CCA system:

**Figure 2-1: Arm CCA software components**



In this diagram, boundaries between worlds are shown as thick dashed lines. Boundaries between lower-privileged software components, which are enforced by software at higher privilege, are shown as thin dashed lines. For example, the Hypervisor at Non-secure EL2 enforces isolation between VMs at Non-secure EL1/0.

## 2.1 Realm Management Extension (RME)

RME is an architecture extension which provides the following primitives: * Two new security states (Root and Realm), in addition to the Non-secure and Secure states * For each of the new security states, a corresponding Physical Address Space (PAS) The following sections describe the software components which run in Root and Realm security states.

## 2.2 Monitor

The software component which runs at EL3 in Root security state is called the Monitor. The responsibilities of the Monitor include:

- Context switching of PE execution between security states

- Managing the assignment of memory to different Physical Address Spaces (PAS). This is performed by writing to the Granule Protection Table (GPT), which is only accessible from Root security state.

## 2.3 Realm Management Monitor

The Realm Management Monitor (RMM) executes at EL2 in Realm security state (R-EL2). Its responsibilities are to: * Provide an execution environment for Realms, which run at R-EL1 / R-EL0. * Isolate Realms from one another.

The RMM acts on requests received from the Non-secure hypervisor to execute and manage Realms. Many of the operations performed by the RMM in response to these requests are similar to the operations performed by the hypervisor when managing Non-secure VMs, such as manipulation of stage 2 translation tables and execution of register save / restore sequences. At the same time, the RMM is much simpler than a typical hypervisor because it does not do any of the following: * Dynamic resource allocation * Make scheduling decisions * Manage interrupts * Provide complex device emulation

Instead, the RMM relies on the Non-secure virtual machine monitor (VMM) and hypervisor (hereafter collectively referred to as “the Host”) to provide this functionality, and its own activities are limited to only those required to protect the confidentiality and integrity of Realms. As a result, its implementation can be much smaller than a typical bare-metal hypervisor.

The RMM provides services to the Host through the Realm Management Interface (RMI), which is an SMCCC-compliant set of commands. The RMI is the interface between the Host and the RMM. The RMI allows the Host to manage the lifecycle of Realms, allocate and reclaim Realm resources, and execute Realm VCPUs.

The RMM also provides services to the Realm, through the Realm Services Interface (RSI). One such service is attestation. Using an RSI command, the Realm can request an attestation report from the RMM, which describes the initial Realm state and the state of the platform. The attestation report is a critical part of the initial establishment of trust in a Realm and is discussed further in Realm management .

Other functionality provided by RSI relates to memory management. The Realm can determine which parts of its address space are private, and which are shared with the Host, and can manage memory sharing dynamically during runtime.

In addition to RSI, the RMM also provides the Realm with trusted implementations of existing firmware standards such as the Power State Coordination Interface (PSCI). This enables existing software written against those standards to be used inside a Realm, while at the same time being provided with additional security guarantees.

# 3. Realm management

This section describes how the software components introduced in Software components interact during the creation and execution of Realms.

## 3.1 Resource management

The fundamental principle of Realm resource management is that the Host remains in control. This means that the Host decides which physical memory is used to back a given Realm Intermediate Physical Address (IPA), or to store a given piece of Realm metadata used by the RMM.

The Host can always reclaim this physical memory, without requiring consent from the Realm. Similarly, the Host remains in control of CPU resources: it decides when to run a Realm VCPU and can cause that VCPU to stop running.

Physical memory is managed in units of a Granule, which is the size of the smallest implemented translation granule. In a CCA system, the granule size must be 4KB. Allocation of memory to a Realm consists of two steps. First, the Host executes an RMI command to perform an operation called delegation. This causes a Granule, chosen by the Host, to transition from the Non-secure PAS to the Realm PAS. Then the Host executes another RMI command to request the RMM to use that Granule as an "RMM object", each of which has a specific associated purpose: * Realm Data – Memory which is mapped into the Realm's address space. * Realm Translation Tables (RTT) – Describes the properties of the Realm's IPA space. * Realm Descriptor (RD) – Stores the Realms attributes. * Realm Execution Contexts (REC) – Stores the Realm VCPU state.

To reclaim the Granule, the Host performs the reverse. First, the Host requests the RMM to free the Granule from its current usage. Then the Host requests the RMM to undelegate the Granule, causing its contents to be scrubbed and the Granule to transition back to the Non-secure PAS.

In each case, the RMM checks whether the request is valid, and only modifies system state if it meets a set of specified pre-conditions. For example, a request to delegate a Granule which is not in the Non-secure PAS, or to free a Granule which is in active use by the RMM, is rejected with an error code.

This pattern of checking system state and then either performing a discrete action or failing with an error is widely used in the RMM ABIs and allows the RMM to remain in overall control of the consistency of the system.

## 3.2 Realm creation and attestation

The life cycle of a Realm begins with it being created and populated with content provided by the Host. This includes both memory contents and the register state of all Realm VCPUs. This initial content is measured by the RMM and the result (the Realm Initial Measurement, or RIM) is stored in the Realm Descriptor.

Once the Realm has been fully constructed, the Host executes an RMI command to activate the Realm. This step constitutes a temporal boundary. Once activated, Realm contents are no longer modifiable by the Host, and the RIM is immutable, and its VCPUs can be scheduled. At this point the integrity of the Realm is protected, but its contents by definition do not include any confidential information because they were provided by the untrusted Host.

Before provisioning the Realm with any secrets, the Realm owner first needs to establish trust in the Realm. To do this, the Realm owner needs to know that the Realm has been correctly constructed and is hosted on a robust implementation of the Arm CCA hardware architecture. This knowledge is obtained through an attestation process. The Realm requests an attestation report from the RMM and returns it to the Realm owner. This report contains the measurement of the initial state of the Realm, measurements of firmware components including the Monitor and RMM, and the identity of the hardware platform. The report can also contain measurements taken by the Realm at runtime. The report is cryptographically bound to the underlying physical platform. The report provides sufficient evidence to allow the Realm owner to decide whether to trust the Realm which it describes.

## 3.3 Creating a Realm - Realm lifecycle flow

At boot the firmware is measured. Memory is reserved by the boot firmware to be used by the RMM. The VMM then sends a Launch Realm request to the Hypervisor.

The Hypervisor then makes a series of SMC calls into the RMM via the Monitor. In response to each SMC command, the RMM performs appropriate actions, including: * Creates a Realm instance. * Copies data into Realm PAS memory and maps it into the Realm's IPA space. * Initializes register values stored in RECs. * Updates the RIM to reflect the initial state of the Realm.

The Realm guest is then activated. The Hypervisor cannot directly affect the Realm content, execution state or security configuration after the Realm guest has been activated. After activation, the Realm guest can then be executed.

## 3.4 Destroying a Realm - Realm lifecycle flow

The VMM sends a request to destroy the Realm to the Hypervisor which then makes an SMC call into the RMM via the Monitor. The RMM then destroys the Realm instance and sends a request to the Monitor to scrub the Realms memory and transition the memory from Realm PAS back to Non-secure PAS.

## 3.5 Realm memory management

The IPA space of a Realm is defined by stage 2 translation tables, referred to as Realm Translation Tables (RTTs). An RTT uses the standard format and radix tree format described in the VMSA. The contents of an RTT are directly accessible only to the RMM. The Host uses RMI commands to request modifications to the RTT.

To add memory to a Realm, the Host issues RMI commands, first requesting the RMM to create the RTT, and then requesting the RMM to map a physical Granule at a specified Realm IPA. When this process is performed during Realm creation, the Host provides the content of the Granule, which is measured by the RMM and which will later be observable to memory accesses from the Realm.

After Realm activation, the Host can no longer control the content of memory which is added to a Realm, but it can still provide Granules to back previously unpopulated parts of the Realm's address space, for example in response to a fault. This avoids the need for the Host to fully populate a Realm with memory during creation, which can be time- consuming, and instead provide memory on demand during Realm execution.

The Realm IPA space is divided into two halves. In the "Protected" half, the RMM guarantees that only memory owned by and private to the Realm will be mapped. Confidentiality and integrity of this memory is guaranteed.

In the "Unprotected" half of the IPA space, the RMM permits the Host to create mappings to Non-secure PAS locations. This can be used, for example, for virtual I/O between the Realm and the Host. Alternatively, the Host can emulate access to unprotected memory. This enables the Host to present emulated devices to the Realm.

This division of IPA space enables software in the Realm to easily reason about whether a given memory access crosses the boundary between itself and the untrusted Host. If it does, the Realm should take suitable measures to protect itself, such as sanitizing input values and not writing any confidential data.

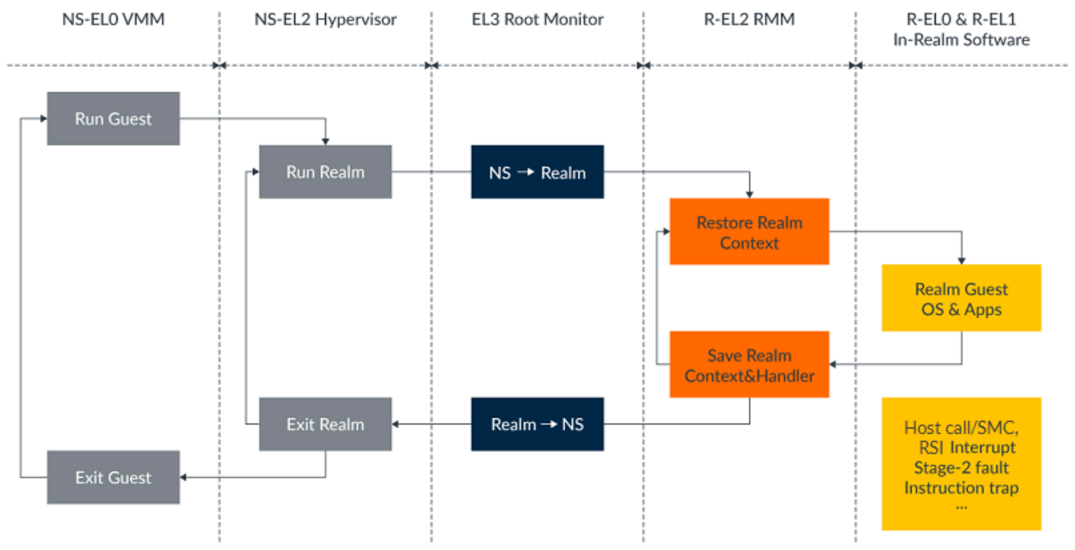
The RMM supports block mappings for both Protected and Unprotected IPAs. The process of establishing block mappings fits in with existing hypervisor memory management flows.

## 3.6 Realm context switching

This section describes how a Realm is scheduled by the Host, and how exceptions taken during Realm execution are delivered to the Host.

The following diagram shows the process of entering and exiting a Realm:

**Figure 3-1: Realm entry and exit**



The decision to run a particular Realm VCPU is taken by the Host and enacted by execution of an RMI "run" command, passing the address of the corresponding REC. This is shown in the diagram as "Run Realm". In response, the RMM restores register state from the REC and then passes control to the Realm. This is shown in the diagram as "Restore Realm Context".

On an exception taken to EL2, the RMM either: * Performs an operation requested by the Realm, and then returns control to the Realm, or * Saves Realm register state to the REC and then returns from the "run" command, providing sufficient information to enable the Host to handle the exception. For example, the Host can respond to a page fault or handle an interrupt.

The flow for scheduling a Realm VCPU and then handling an exit fits in with existing hypervisor scheduling frameworks.

## 3.7 Realm interrupts

The Host presents an emulated Generic Interrupt Controller (GIC) to the Realm. The emulated GIC's MMIO regions are mapped in unprotected IPA space, allowing the Realm's GIC driver to interact with it. On Realm entry, the Host uses the RMI to inject virtual interrupts into the Realm, which are presented using the GIC CPU interface. Usage of the GIC CPU interface accelerates processing of interrupts by avoiding traps to the Host on a set of interrupt management operations.

Because the GIC emulation is provided by the untrusted Host, the Realm cannot rely on correct interrupt routing and prioritization, or on the validity of an incoming interrupt. For these reasons, the Realm must protect itself by hardening its own interrupt processing subsystem, based on the assumption that the GIC emulation is malicious.

# 4. Related information

The following resources contain material related to the contents of this guide:

- Arm Community
- Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A
- Arm Realm Management Extension (RME) System Architecture
- Arm System Memory Management Unit Architecture supplement - The Realm Management Extension (RME), for SMMUv3

# 5. Next steps

This guide introduced you to the software components of Arm CCA.

The following guides provide more information about Arm CCA:

- Introducing Arm Confidential Compute Architecture guide
- Arm Realm Management Extension guide