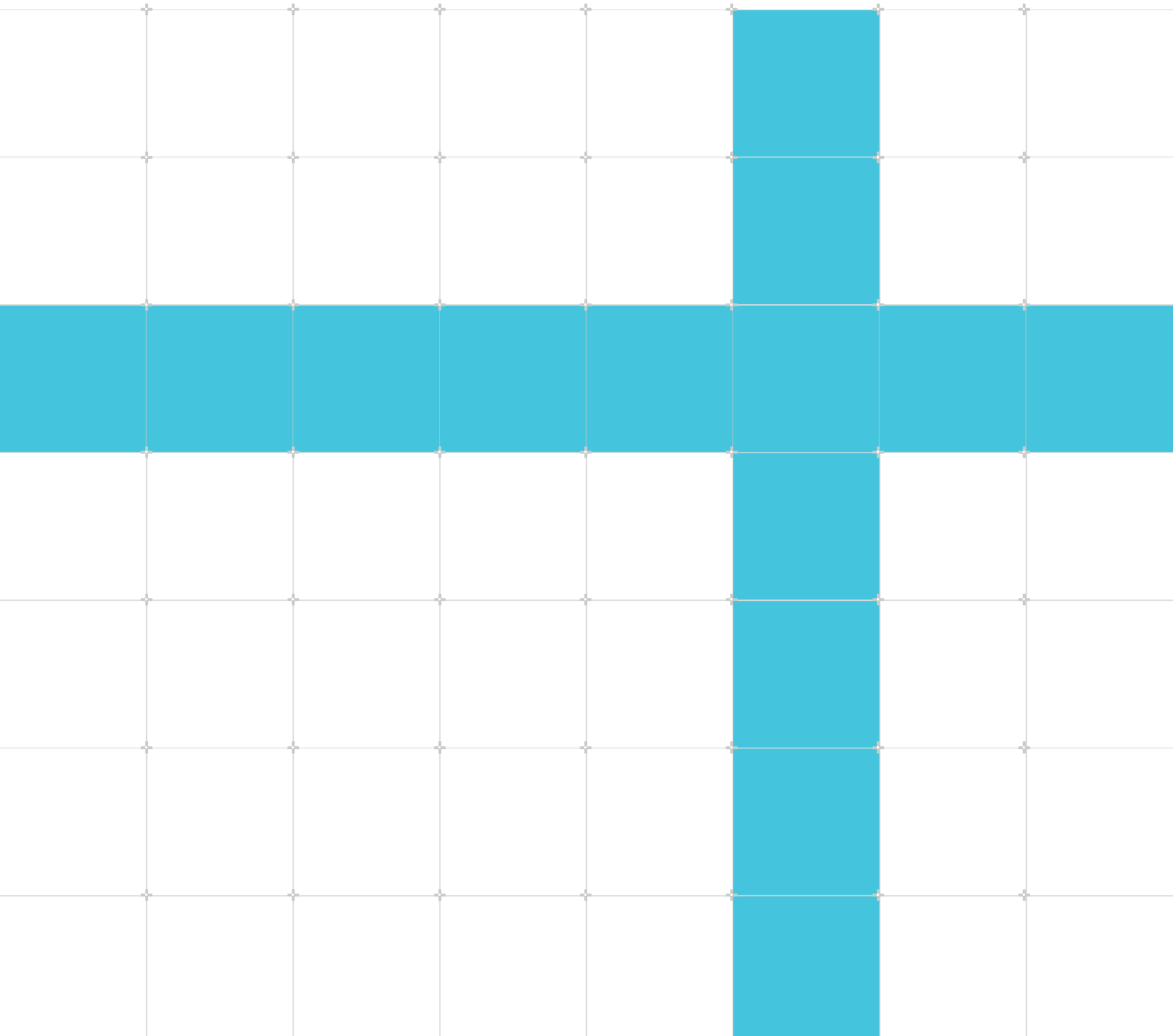
Learn the architecture - Arm Confidential Compute Architecture software stack

Version 3.0

Non-Confidential

Issue 06
den0127_300_06_en



Learn the architecture - Arm Confidential Compute Architecture software stack

Release information

Document history

Issue	Date	Confidentiality	Change
0200-06	30 June 2025	Non-Confidential	Minor update
0200-05	29 June 2023	Non-Confidential	Change the title
0100-04	9 May 2023	Non-Confidential	Minor update
0100-03	13 September 2022	Non-Confidential	Minor update
0100-02	20 September 2021	Non-Confidential	Initial release

Proprietary Notice

with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. 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, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

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.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted 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 this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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. Introduction **7**
1.1 Before you begin 7
1.2 Arm CCA goals 7
1.3 Hardware-software split 8

2. Software components **9**
2.1 Realm Management Extension 9
2.2 Monitor 9
2.3 Realm Management Monitor 10

3. Realm management **12**
3.1 Resource management 12
3.2 Realm creation and attestation 13
3.3 Creating a Realm 13
3.4 Destroying a Realm 14
3.5 Realm memory management 14
3.6 Realm context switching 15
3.7 Realm interrupts 15
3.8 Realm Translation Table (RTT) 16
3.8.1 Lifecycle of RTT 16

4. Realm communication APIs **17**
4.1 Realm Management Interface 17
4.1.1 RMI commands 17
4.2 Realm Services Interface 18
4.2.1 Security Model 18
4.3 PSCI support 18
4.3.1 Realms and PSCI 18
4.4 Realm Host Interface 19
4.4.1 Self discovery 19
4.4.2 Security 19
4.4.3 Host Session 20
4.5 Firmware Activity Log 20

4.6 Device Assignment....20

5. Arm CCA reference software stack....21
5.1 TF-RMM....22
5.1.1 TF-RMM booting....22
5.1.2 TF-RMM runtime service....23
5.1.3 Realm interrupts and timers....24
5.2 TF-A for Arm CCA....24
5.2.1 Boot flow for Arm CCA....24
5.2.2 RMM Dispatcher....25
5.2.3 GPT handling....26
5.3 KVM support for Arm CCA....26
5.3.1 Realm VM lifecycle....27
5.3.2 Realm memory management....28
5.3.3 RMI interface....28
5.3.4 Device virtualization....28
5.4 kvmtool for KVM hosting....29
5.5 Guest Linux in Realm....30
5.6 EDKII guest firmware for Arm CCA....31
5.7 How to set up a Realm....32
5.8 Hands-on with Arm CCA example Stack....33
5.8.1 Debug Arm CCA software stack....34

6. Related information....35

7. Next steps....36

1. Introduction

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

This guide describes 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.

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 (VM), while providing protection to data in use inside the VMs. To enable this protection, Arm CCA introduces the concept of a confidential 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, making it easier to reason 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 deny the Realm availability. The hypervisor can choose to not to schedule its Virtual CPUs (VCPUs) or reclaim a resource, such as a device or memory, required by the Realm.

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

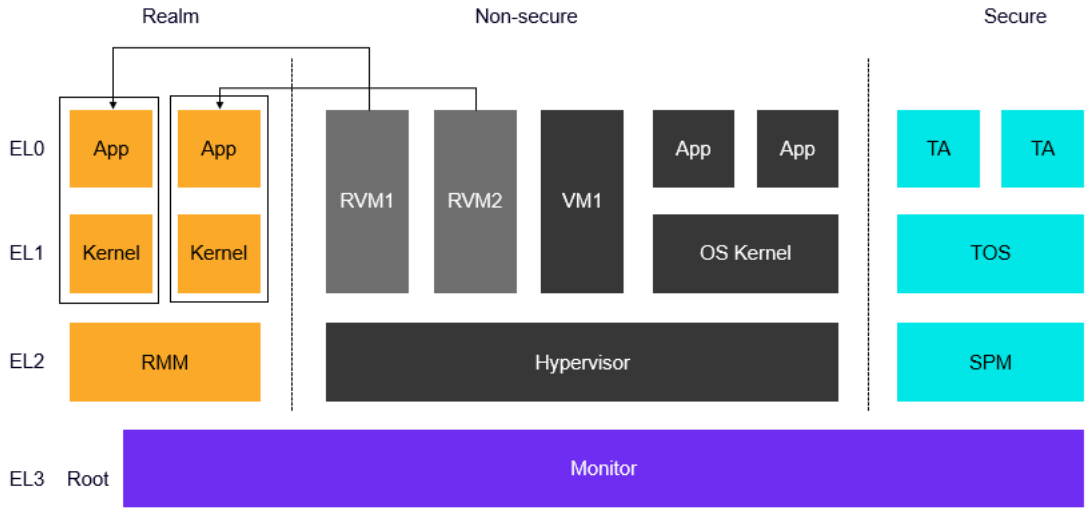
A key consideration in the design of a technology such as Arm CCA is the split between hardware and software. At one extreme, solutions can exist which make no change to existing hardware architecture and deliver the required functionality entirely in software. At the other, designing all the necessary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

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

This section describes 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 is an architecture extension which provides the following primitives:

- Two new security states, Root and Realm, are added to the Non-secure and Secure states
- For each of the new security states, a corresponding Physical Address Space (PAS)

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, 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 described in Software components interact during the creation and execution of Realms.

3.1 Resource management

The main 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. 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 fist content is measured by the RMM and the result,the Realm Initial Measurement (RIM), is stored in the Realm Descriptor.

When the Realm has been fully constructed, the Host executes an RMI command to activate the Realm. This step constitutes a temporal boundary. When 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 enable the Realm owner to decide whether to trust the Realm which it describes.

3.3 Creating a Realm

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 RMI commands into the RMM via SMC to the Monitor.

In response to each RMI 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.

3.4 Destroying a Realm

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, scrubs the Realm's memory and transitions 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 is later 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.

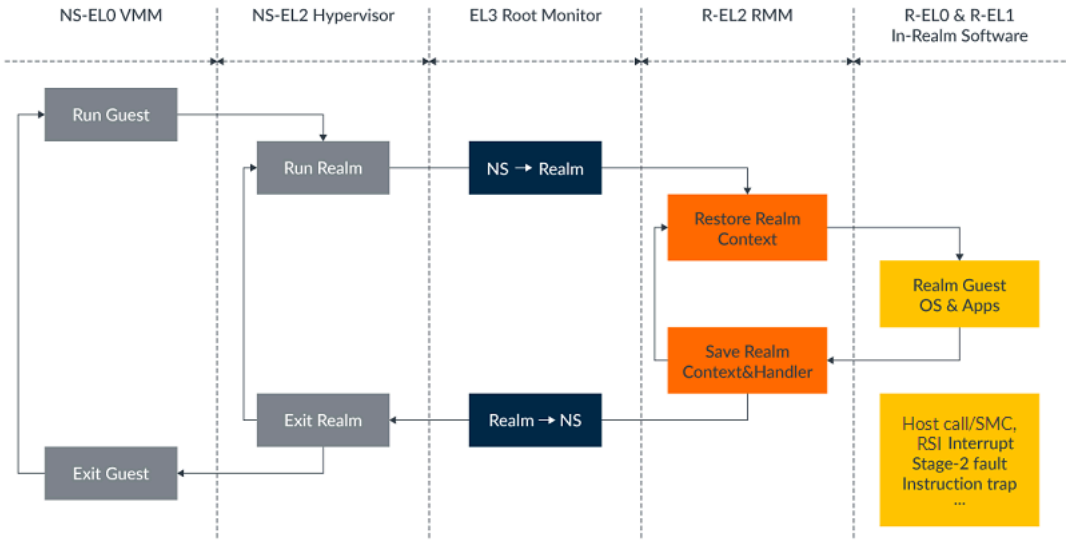
The RMM supports block mappings for both Protected and Unprotected IPAs. The process of establishing block mappings is designed to fit in with existing hypervisor memory management flows.

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_REC_ENTER 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 a non-secure interrupt.

The flow for scheduling a Realm VCPU and then handling an exit is designed to fit 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.

3.8 Realm Translation Table (RTT)

The RTT is the page table structure that manages Realm address translation and memory ownership.

The RTT is used by the Realm World but it is controlled and managed by the RMM, not the Realm itself. This enforces strict separation between control by the management plane and execution by the Realm.

- RTT defines Intermediate Physical Address (IPA) to Physical Address (PA) mappings within a Realm.
- Each Realm has its own RTT, similar in concept to a stage-2 page table used by a hypervisor. The RTT separation is enforced by hardware and RMM. The Realm cannot modify the RTT directly.

3.8.1 Lifecycle of RTT

The RMM creates the RTT in response to a request from the Host:

1. Host requests Realm creation.
2. RMM creates an empty RTT structure.
3. Host, via RMM maps pages into the Realm.
4. RTT enforces memory ownership and isolation at granule level.
5. Realm uses the RTT implicitly during execution for IPA → PA lookups.
6. RMM tears down RTT on Realm destruction.

4. Realm communication APIs

Communication between each CCA software componant is through well-defined APIs at the interface boundary.

4.1 Realm Management Interface

The Realm Management Interface (RMI) is a software interface implemented by the Realm Management Monitor (RMM), a trusted, privileged software layer, to create, manage, and destroy Realms. RMI is used by the host.

The Realm Management Interface provides calls for:

- Realm creation and destruction
- Memory assignment and management for example assigning non-secure memory to Realms
- Entering and exiting Realms
- State transitions for example suspend/resume, booting Realms
- Monitoring and attestation

4.1.1 RMI commands

The RMI defines a set of commands that enable the host to manage Realms via the Realm Management Monitor (RMM) using Secure Monitor Calls (SMCs).

RMI commands are able to:

- Create a Realm amd allocate its metadata
- Destroy a Realm and deallocates all resources
- Create and destroy a Realm Execution Context (REC), similar to a Realm vCPU
- Convert a memory page into Realm-owned secure data
- Convert Realm data back to non-secure memory
- Enter a Realm and begin execution
- Exit a Realm and return control back to the host
- Query available features and interface version

All RMI calls are made using SMC call from the host. The RMM validates each request and enforces memory and Realm isolation.

4.2 Realm Services Interface

The Realm Services Interface (RSI) allows Realm payloads, software running inside a Realm, to request services from the RMM via SMC calls.

4.2.1 Security Model

RSI calls are strictly controlled by the RMM and cannot access host or secure world memory. All calls are non-preemptive and return directly to the caller like system calls. Realm payloads use `SMC` instructions with RSI function IDs to invoke these services:

- RSI provides minimal, well-defined services to reduce the attack surface.
- These calls are Realm-initiated and do not expose internal state unless authorized.

4.3 PSCI support

The Power State Coordination Interface (PSCI) is the standard Arm interface for managing power state transitions for example CPU suspend, power-off, reset.

PSCI is a firmware-level interface provided by the platform (usually via EL3 firmware or Secure Monitor) that offers:

- CPU on/off
- Suspend/resume
- System reset/shutdown

PSCI is invoked via SMC calls by higher-level software.

4.3.1 Realms and PSCI

Realm and Host software interract with PSCI in different ways.

The Realm uses PSCI to manage the power state of what it thinks of as physical CPUs. These are virtual CPUs, so the requests need to be handled by the host.

4.4 Realm Host Interface

The Realm Host Interface (RHI) is a communications channel that enables the realm kernel or firmware to communicate with the Host Hypervisor. This enables the realm software to request data from the Non-secure host or request access to services that are managed by the host.

Some Realm flows involve steps which have one or both of the following properties: * Implementation can vary between platforms * Requires storage of large amounts of data

Implementing such functionality in the RMM is undesirable. Therefore the functionality is implemented by the host, and provided to the Realm as a service. RHI is a set of standard interfaces for interacting with these services.

Examples of such flows are: * Obtaining a log of events which have modified the Realm's Trusted Computing Base (TCB), for example by updating the platform firmware * Obtaining artifacts which are used to determine the trustworthiness of a Realm-assigned device.

RHI commands are split into groups based on use-case and, depending on the platform, not all use-cases are valid. This means that not all commands are available on every platform.

As realms have no direct mechanism to communicate with the Non-secure host, RHI operation relies on the Realm Management Monitor (RMM) to provide a communication channel. RHI commands are communicated over the Realm Services Interface (RSI) using `RSI_HOST_CALL` commands.

4.4.1 Self discovery

There are three RHI Protocol sets for the currently supported use-cases:

- Host Session
- Firmware Activity Log
- Device Assignment

Realm software can query the available protocol using an `RHI_IMPLEMENTATION_FEATURES` call. It is can then discover which individual calls are available within that protocol.

4.4.2 Security

Any data that is received through these interfaces is provided by the Non-secure Host and is therefore outside the Trusted Computing Base for Realms. Extra security checks are needed on a per-use-case basis to ensure the safe usage of any data.

4.4.3 Host Session

The Host Session protocol creates a generic communication channel between the Realm and the Non-secure world. The channel is created by the Non-secure host, using an agent such as kvmtool The host can then link the channel to the appropriate Non-secure client.

Clients can support a blocking or non-blocking implementation of the interface, this is known as the connection mode. The API can query the version, supported connection mode and if sending or receiving messages are supported.

This protocol provides no security guarantees for any messages. If this is necessary then extra security should be implemented for that use case. For example, session keys can be exchanged through a non-authenticated key exchange protocol, such as Diffie-Hellman.

4.5 Firmware Activity Log

If a platform supports Live Firmware Activation, then the platform Root-of-trust creates a log of firmware activation measurements, known as the Firmware Activity Log (FAL). The FAL is maintained by the Non-secure software but can be validated against a trusted source. The FAL can then be used by a consumer, the Realm in this case, to check the security of the platform fimrware.

The Realm can use this RHI protocol to obtain the FAL. It can then use the CCA Platform attestation token to check the integrity of the data.

4.6 Device Assignment

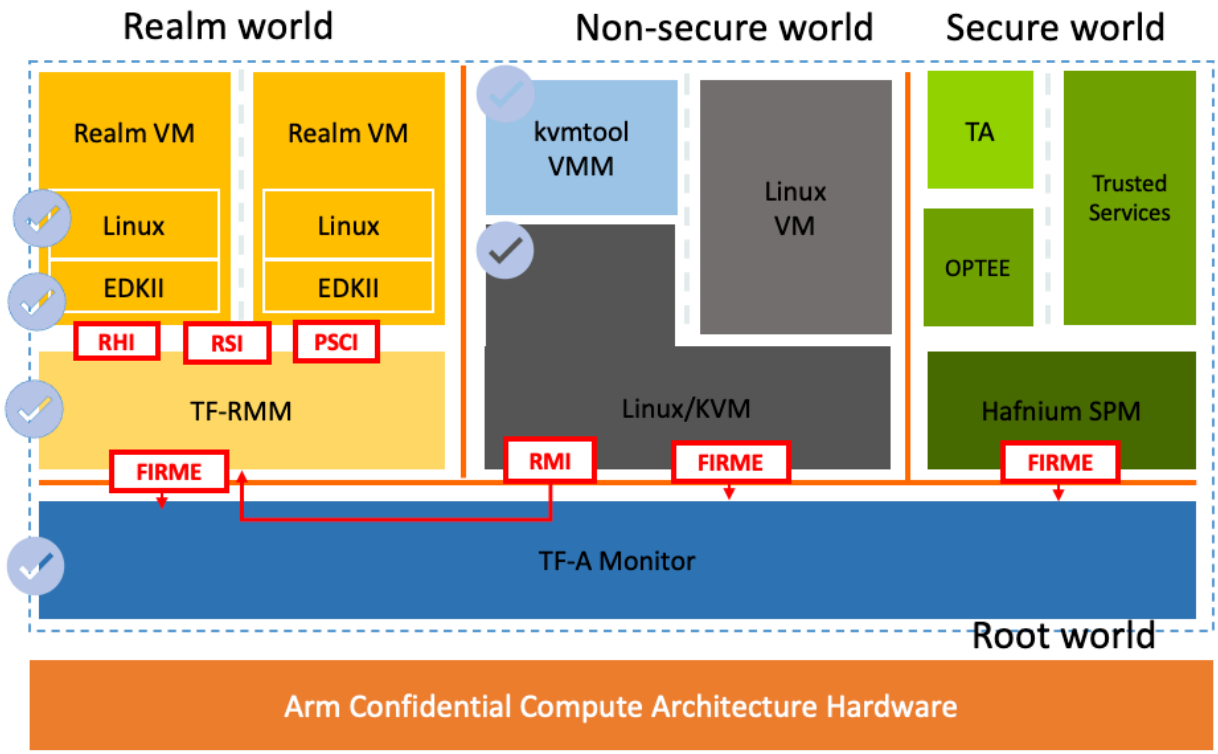
PCIe Device Assignment relies on the trustworthiness of a device to be used by the Realm. This is attested by a certificate which the RMM obtains from the device. The RMM gets an interface report and perform some device measurements. This information is passed to the Non-secure host to be cached.

The Device Assignment protocol provides a series of commands which enable the request the host's copies of the device information that were previously cached. The Realm can then use RSI to request a digest of the same information from the RMM and can compare the two before mapping the device in to the Realm.

5. Arm CCA reference software stack

This section describes the software components of the Arm CCA reference stack. It highlights a newly developed component and enhancements made to most existing components to support the RME hardware extension.

Figure 5-1: Software stack for Arm Confidential Compute Architecture



The software stack includes following components:

- TF-A, which serves as EL3 monitor by doing context switching of CPU execution between security states, and manages Granule Protection Table (GPT). The EL3 Monitor also performs realm attestation and device assignment supports.
- TF-RMM, which is fully compliant with RMM specification, and implements RSI and RMI interfaces.
- Linux KVM
  - KVM is a type-2 hypervisor. Due to FEAT_VHE architecture feature, both the host kernel and KVM module run on Non-secure EL2, which can reduce extra context switch.
  - KVM for CCA manages Realm VMs, and communicates with TF-RMM via RMI interface to delegate pages to the Realm world. The delegated pages are inaccessible in the Non-secure world.
- kvmtool Virtual Machine Monitor (VMM)

    - It creates and schedules VM with the help of Linux KVM. VMM also provides device virtualization, by emulation or direct assignment of physical devices.
    - It uses new KVM UAPIs and capabilities to support Realm VM.
    - It is one prototype. Real-world Arm CCA deployments are likely to use a production-quality VMM.
- Realm Software
    - EDKII firmware is enhanced to boot guest Linux kernel in Realm world.
    - Guest Linux running in Realm world, supports RSI to communicate with TF-RMM to manage the Realm IPA State and to discover the configuration of the realm.

This document describes the software stack blocks that are checked in above diagram.

Note: Those software stack are keeping evolving:

- CCA v1.0 software stack is foundation for CPU-only realms.
- On CCA v1.1 software stack, there are more use cases.

Future document releases will describe these more cases.

There are several reference hardware platforms with corresponding software stack to help you adopt Arm CCA for confidential computing:

- Arm's Base AEM FVP platform
- RD-V3 Platform
- QEMU including RME enablement provided by Linaro

5.1 TF-RMM

TF-RMM is an OSS reference implementation hosted on trustedFirmware.org and is suitable for use by all Arm CCA deployment. You can get it from upstream or can contribute to this open governance project, and there is an open technical forum to join. The software code is organized via CMake build system. See file-org-and-config for more details.

5.1.1 TF-RMM booting

TF-RMM firmware is part of the system firmware as packed in Firmware Image Package (FIP) format. It is loaded into Realm EL2, activated by the EL3 Root firmware and measured in platform attestation token. The load address is determined in TF-A BL2 firmware, see the TF-A chapter for the TF-RMM boot flow.

The firmware runs with the memory statically carved out from DRAM. There are build options which can impact memory usage. The primary and secondary CPU booting flow on TF-RMM is different, because the first needs to initialize BSS, fix up GDT and relocate TF-RMM to run as

Position Independent Executable (PIE) image. See the boot flow on the TF-RMM page cold-and-warm-boot-design for more details.

During boot, the initialization code sets up Realm EL2 translation region based on Arm v9-A VMSA architecture and exception model. The page table is handled by the xlat library, which can support up to 52-bit addresses and 5 levels of translation when FEAT_LPA2 is enabled.

TF-RMM enables FEAT_VHE to split the 64-bit VA space with low and high VA regions. The low VA range creates static mappings for the TF-RMM executable memory and the EL3 Shared memory region. The high VA range creates dynamic per-CPU mappings to manage run-time data including Realm metadata, NS data and Realm data, and this VA range is also used for EL0 App. See the chapter memory management and EL0 apps in RMM for more details.

When the TF-RMM boot succeeds, there is handshake between TF-RMM and TF-A EL3 firmware. It issues one SMC call with value {*SMC_RMM_BOOT_COMPLETE*, *E_RMM_BOOT_SUCCESS*} to notify EL3 firmware, then returns to the handler *rmm_handler*, which is ready to handle RMI requests within its loop.

5.1.2 TF-RMM runtime service

The role of TF-RMM is to serve SMC calls including RMI, RSI, RHI and PSCI, and issue FIRME calls like Granule Management Interface (GMI). The function calling flow is enabled by AArch64 exception model.

RMI SMC calls originate from KVM on Non-secure EL2. They are relayed to TF-RMM via EL3 TF-A monitor via exception return. Those host requests are handled by RMI SMC calls. One SMC handler named *smc_handlers[]* array is introduced to handle RMI requests from non-secure world. It's much easier to add new entries to support new RMI requests. One notable merit is that the array structure can enable execution log and error log. It is useful to check one Realm creation flow or do trouble-shooting with the execution log by simply changing the third value to enable or disable the logging.

The RMI runtime handlers are under rmi. Those RMI handlers are called via one common handler *rmm_handler* through exception return from EL3 firmware TF-A.

RSI ABI calls are made from the Realm to the TF-RMM and serviced by the TF-RMM. Those APIs are used to query Realm configuration, manage Realm IPA State (RIPAS), communicate with the Host via *RSI_HOST_CALL*, and execute attestation & measurement for platform and Realm.

RSI handler software are under rsi. The Realm exit into TF-RMM is carried out via exception entry. TF-RMM supports exception vector *el2_vectors* from lower EL with AArch64 entry. TF-RMM handles RSI calls during the REC running flow because a Realm is only scheduled by the *RMI_REC_ENTER* call.

PSCI ABI calls via SMC are made from the Realm to the TF-RMM. PSCI calls via HVC conduit are not supported for Realms, that's not same as Normal VM. TF-RMM injects Undefined Synchronous Exception into vCPU when handling HVC exception.

PSCI calls are related to power management of vCPU or the whole Realm VM. TF-RMM serves and terminates query PSCI calls like *PSCI_AFFINITY_INFO*, *PSCI_FEATURES* and *PSCI_VERSION* within its context, while for other PSCI calls, TF-RMM need route them to Linux KVM for further handling.

FIRME GMI calls originate from TF-RMM to TF-A firmware to request granule transition delegate services, as TF-RMM serves *GRANULE_DELEGATE* and *GRANULE* UNDELEGATE_ RMI calls to request the change of GPT. There are other EL3–RMMD SMC calls to request RMM Dispatcher (RMMD) runtime services like realm attestation and Integrity and Data Encryption (IDE) key management.

5.1.3 Realm interrupts and timers

Virtual GIC is presented to Realm VM via KVM host emulation and TF-RMM coordination, so delivery of virtual interrupts to a Realm is supported. GIC MMIO interfaces are emulated in KVM. Compared to the PV GIC mechanism, the GIC virtual CPU interface presented to Realm VMs is designed for better performance.

TF-RMM does not make use of arch timers by itself as Non-secure EL2 timers are configured by the host, and EL1 timer are configured by Realm. However, it needs to handle EL1 timer state during REC entry and exit for Realm.

5.2 TF-A for Arm CCA

TF-A firmware is the most privileged software component executing in EL3 Root state, it manages security state switching and assigns resources between security states. For Realm state, additional contexts in TF-A are added for security state transitions. During run-time, it handles SMC calls from TF-RMM to configure PAS for data (un)delegation, update GPT and execute TLB maintenance to Point of Physical Aliasing (PoPA).

TF-A includes BL1 boot ROM image, BL2 image and BL31 Run-time firmware for Arm CCA enablement.

5.2.1 Boot flow for Arm CCA

TF-A BL2 loads FIP image, which includes TF-RMM image. Since the image needs be loaded into Realm PAS, the boot flow is modified to run BL2 at EL3 as typically BL2 runs under Secure-EL1. As on RME architecture, Stage 1 translation table on EL3 with NS and NSE bits can map different physical address spaces, while the NSE bit is **RES0** in Secure translation regime. See the page boot-flow-changes for further details.

BL1 image is in boot ROM, which serves as Root of Trust. It boots BL2 image with EL3 exception level via *MODEL_EL3* setting. See function bl1_run_bl2_in_root for more details.

BL2 image only runs on primary CPU boot flow. It loads all firmware images in one FIP package. It initializes the entire protected space with *GPT_GPI_ANY* to permit all PAS accesses, then carves out defined PAS ranges to enable granule protection check. See chapter GPT handling for more details.

BL31 controls the firmware booting order according to BL2 setup. Before booting TF-RMM image, BL31 loads TF-RMM manifest, sets up TF-RMM context and registers one deferred initialization function during setup of standard runtime services, then enters TF-RMM via exception return (refer to *rmmd_rmm_sync_entry()*). TF-A needs to know the TF-RMM initialization result. If it is not successful, TF-A marks a boot failure flag for all the CPUs, and RMM-EL3 interface SMC are treated as unknown. BL31 also subscribes to *PSCI CPU_ON* call to initialize TF-RMM for secondary CPU booting.

In Non-secure world, BL33 firmware usually is EDKII or U-Boot, which loads Linux kernel. BL32 is Secure Partition Manager (SPM) Hafnium, which is not mandatory in the Arm CCA reference software stack, for example, the cca-3world configuration.

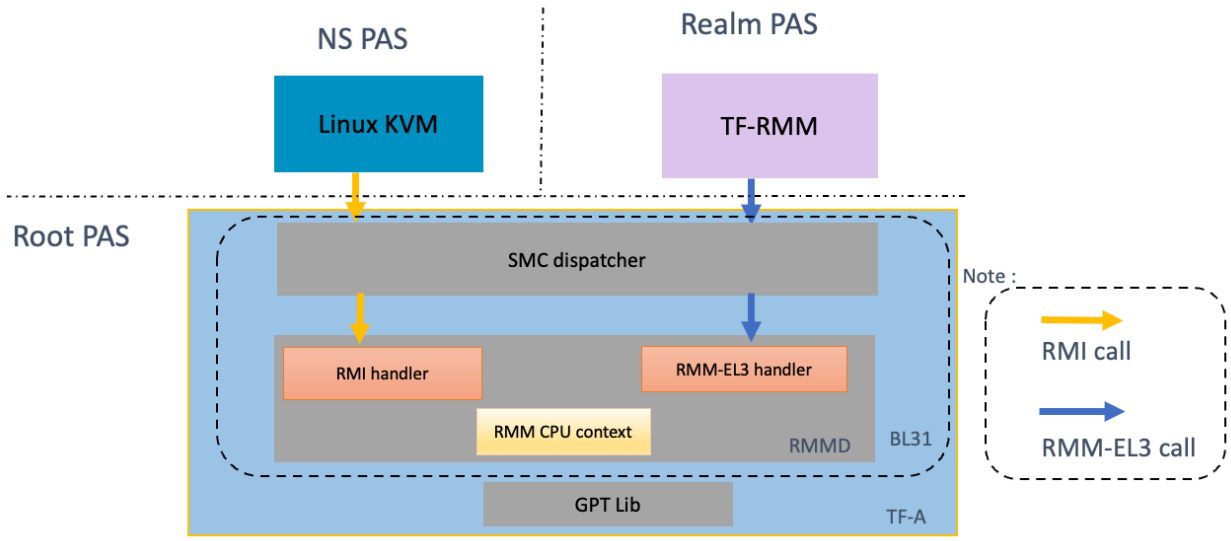
5.2.2 RMM Dispatcher

RMMD is created to handle the switch from EL3 to the Realm world. RMI calls originated from Non-secure world are forwarded into TF-RMM on Realm EL2 by the RMMD. RMM-EL3 handler serves GMI requests, attestation requests and other requests from Realm world.

The PSCI power management operations are registered and notified to RMMD via publish-subscribe mechanism. Currently *PSCI_CPU_ON* finish event needs to be propagated to TF-RMM.

Realm CPU context is added based on the context management library. It is the data structure that the RMMD uses in EL3 to track context of the RMM at R-EL2 and keeps all physical CPU contexts. See psci-cpu-context-management for more information.

Figure 5-2: TF-A run-time handler for Arm CCA software stack



5.2.3 GPT handling

GPC enables dynamic PAS access control. To implement GPC protection, the GPT must be created and enabled during the boot process. TF-A provides a GPT library that initializes the GPT based on the system's memory layout. It configures key GPC system registers such as *GPCCR_EL3* and *GPTTR_EL3* to activate granule protection checks in the BL2 firmware.

GPT has two levels of GPT tables: level 0 GPT are GPT Block or GPT Table descriptors, and level 1 GPT are GPT Contiguous or GPT Granules descriptors. The GPT library makes use of these formats. The GPT lookup is done according to the values of *GPCCR_EL3.{PPS, PGS,L0GPTSZ}* for *{T,P,S}*. Notably *L0GPTSZ* defines Level 0 GPT entry size and sets the size of the level 0 GPT in conjunction with the programmed PPS. Each entry in the level 0 table covers a larger granule size and therefore requires a larger level 1 table size. Limiting the size of the level 0 table can ensure that it fits into on-chip memory for faster responses for level 0 fetches.

During boot, the GPT library first initializes the entire protected space to allow all PAS accesses using level 0 block entries. It then configures the carve-out PAS ranges based on the PAS regions defined for each platform. For example, they are differences on Base FVP and RD-V3 platforms. The GPT library might modify the GPT region's structure due to runtime requests, so those data cannot be constant.

The secure PAS are used for the secure world images like SPM. The Realm world PAS will be used for TF-RMM. The Root world carve-out memory is used for L1 GPT tables. If the TF-A firmware needs to store any data affecting the Security model of Arm CCA on DDR, then that data needs to have additional integrity protection implemented in software.

During runtime, the GPT library on BL31 firmware handles transition requests from TF-RMM and Secure EL2 SPM firmware. The transition delegate is related to change GPT on-the-fly, so the GPT library needs do TLBI and change GPT.

The library may either shatter contiguous blocks or fuse adjacent GPT entries to form a contiguous block. Depending on the maximum block size, the fuse operation might propagate to higher block sizes as allowed by RME Architecture. There is fine-grained lock implementation in the software. See GPT design for more information.

5.3 KVM support for Arm CCA

Our design principle is to reuse as much Linux KVM infrastructure as possible to enable Arm CCA. The actions performed by KVM to manage a Realm are like those required to manage a normal Non-secure VM, which includes UABI call, guest enter and exit handling flow. The difference is that most of those actions are indirected via the TF-RMM to uphold the security guarantees of the Realm.

5.3.1 Realm VM lifecycle

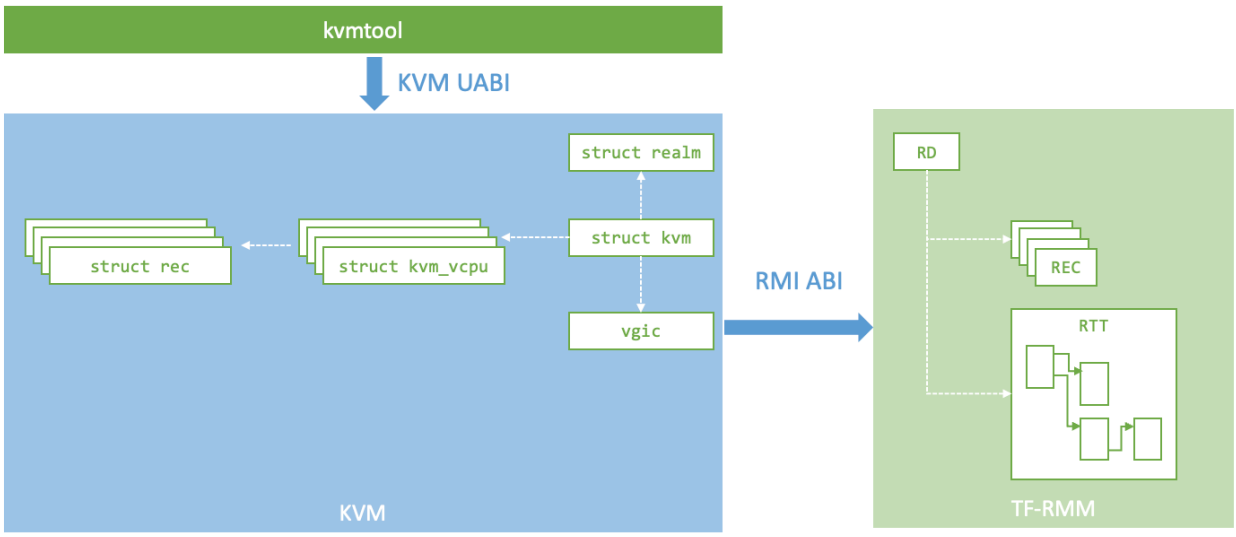
Realm creation is initiated through the *KVM_CAP_ARM_RME* capability using the parameter *KVM_CAP_ARM_RME_CREATE_RD*. In this flow, the VM ioctl allocates memory for Realm Descriptors (RD) and RTTs, notably VMID values are assigned separately from those used by normal Non-secure VMs.

The *KVM_CAP_ARM_RME_CONFIG_REALM* capability allows for the configuration of Realm parameters such as IPA size, PMU counter count, and the maximum SVE vector length. These parameters, when measured, play a role in attestation. Understanding the differences between how the TF-RMM and KVM handle them is key to grasp the setup complexities.

Realm vCPU creation is triggered via *KVM_ARM_VCPU_REC* feature. These vCPUs require extra granules to store their register states. When all configurations for the Realm VM are complete, a finalizing *ioctl(KVM_ARM_VCPU_REC)* KVM UABI call from VMM commits the Realm setup.

In a normal VM, kernel objects stores the VM state, and KVM directly interacts with hardware to manage exceptions, memory mapping and context switches. However, when hosting a Realm VM, these kernel objects act as proxies for TF-RMM-managed objects. KVM then communicates with TF-RMM via RMI ABIs to handle exceptions, map memory, perform context switches, and manage the overall Realm state.

Figure 5-3: KVM manage a Realm VM



Entry and Exit of a Realm VM attempts to reuse the KVM normal VM entry and exit flow, but the final mechanism is different.

Entering a realm is done using a SMC call *RMI_REC_ENTER* to the TF-RMM. Before entering Realm, KVM synchronizes the General-Purpose Registers (GPRs) and vGIC state into a structure called *rec_entry*.

Exiting a realm is managed separately. When a realm exit occurs, KVM synchronizes the GPRs, vGIC state and timer state in the *rec_exit* structure. It then processes any exit events that are specific to the Realm environment. For common exit scenarios, such as stage 2 aborts, KVM falls back to the standard KVM handling routines.

The RMM specification provides a new mechanism for a guest to communicate with KVM called "Host Call". For now, this is hooked up to the existing support for HVC calls from a normal guest, and registers are managed during REC entry and exit.

5.3.2 Realm memory management

Arm CCA memory management in KVM is designed with Stage 2 control handled by the TF-RMM, while physical memory pages are managed by KVM. The system uses fixed 4K pages with IPA sizes, L2 block mappings with typically 2MB and LPA2. Instead of maintaining shadow page tables, KVM donates RTT pages directly to the TF-RMM.

KVM uses RMI calls to delegate pages to the Realm world. When delegated, the pages are inaccessible to the Normal World unless explicitly shared by the guest. The runtime Granule Protection Fault (GPF) faults can be handled in KVM caused by unauthorized access to protected memory. Host page walks trigger kernel *OOPs* and user-space faults generate *SIGBUS*.

KVM can destroy the Realm VM at any time to reclaim the pages. There is no support for paging, memory must be pinned by the VMM.

KVM manages RIPAS to permit VMM to control memory state with RAM and EMPTY for protected regions. Transitions require TF-RMM interaction and page un-delegation with TF-A.

5.3.3 RMI interface

KVM adds new SMC definitions for RMI calls to communicate with TF-RMM and creates wrapper functions to simplify error handling for RMI calls.

5.3.4 Device virtualization

For GIC virtualization, KVM saves and restores GIC state from REC structure. It handles RMM-provided GIC List Registers.

TF-RMM manages generic timer state during REC execution, and KVM handles host-side updates.

Physical device assignment is not yet supported by the RMM v1.0, so it is not a good idea to permit device mappings within the realm and softare needs prevent them when the guest is a realm.

5.4 kvmtool for KVM hosting

KVM is the backbone of virtualization in the Linux kernel, but it requires a VMM launcher to create and manage virtual machines. To facilitate this, the host kernel exposes device node interfaces that let user-space applications to interact with KVM. The host Linux system provides three-level ioctl interfaces between the VM launcher and KVM.

System ioctls are applied to the whole kvm subsystem. It is used to create virtual machines via *KVM_CREATE_VM*. VM type with *KVM_VM_TYPE_ARM_NORMAL* is for normal VM, *KVM_VM_TYPE_ARM_REALM* for Realm. KVM support for Realm is advertised via new CAP *KVM_CAP_ARM_RME* and Realm's life cycle is managed via *KVM_ENABLE_CAP* UAPI.

VM ioctls are applied to one entire virtual machine. They are used to create virtual CPUs. VM ioctls run only from the same process that is used to create the VM.

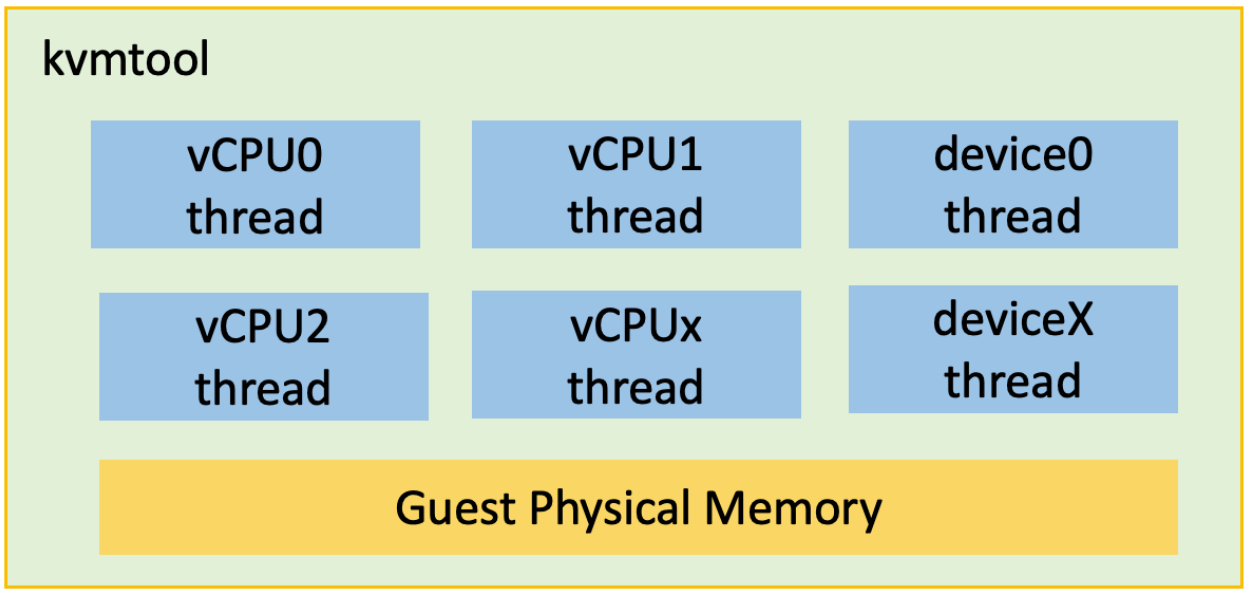
VCPU ioctls control the operation of a single virtual CPU and run only from the same thread that was used to create the vCPU.

Kvmtool is a lightweight tool for hosting KVM guests to run Linux guest VMs. As under KVM, every VM is implemented as a regular Linux process, scheduled by the linux kernel scheduler. Kvmtool also provides network and block virtual devices via virtio to VM.

Kvmtool creates one software thread per vCPU. Device models can run concurrently in vCPU thread while long running operations can run in additional device threads. There are standard KVM UABIs between the kernel and VMM for Realm creation, memory management, running and servicing.

The following figure shows one run-time kvmtool process for KVM guest hosting:

Figure 5-4: kvmtool hosting a Realm VM



Use the following command to launch a realm VM:

```
$ lkvm run
    --realm
    --restricted_mem
    [ --measurement-algo="sha256","sha512" ]
    [ --realm-pv="<realm-pv>" ]
    <normal-VM options>
```

Notes:

- `realm` is added to the `run` command to mark the VM as a confidential compute VM.
- `restricted_mem` is to use restricted memory for Realm guests that VMM can't access, one must option or fails to launch one realm VM.
- `measurement-algo` (Optional) specifies the algorithm selected to create the initial measurements by the TF-RMM for this Realm (defaults to sha256).
- `realm-pv` (Optional) Realm personalization value

Additional enhancements on kvmtool to enable Arm CCA:

- Doubles the IPA Size for a Realm VM since the IPA space of a Realm is split into Protected and Unprotected, with one alias of the other.
- Enforce *VIRTIO_F_ACCESS_PLATFORM* flag is for virtio on the Realms, to block device access.
- Disable *KVM_CAP_READONLY_MEM* for realms prevents any possibility of the host mapping guest memory into its own address space for security rationale.

5.5 Guest Linux in Realm

Guest Linux support in the Realm world is critical to expanding the Arm CCA ecosystem. As most existing confidential computing solutions rely on proprietary hypervisors, enabling Linux as a guest OS unlock accessibility for a broader range of users and use cases. The supporting software has been up-streamed into mainline kernel.

Guest Linux can communicate with the TF-RMM via RSI interface. These include:

- Realm Guest Configuration
- Attestation & Measurement services
- Managing the state of IPA pages
- Host Call service, to communicate with KVM

Wrappers for the full set of RSI commands are added, which can manage the RIPAS and discover the configuration of the realm. The VM running within the Realm needs to ensure that memory which is going to be used is marked as *RIPAS_RAM*, which means protected memory only accessible to the guest. This VMM can provide this , and it is subject to measurement to ensure the memory is setup correctly or the VM can set it itself. In the latter case, all described RAM is iterated over and set with RIPAS during primary vCPU boot.

Within the Realm, the most-significant bit of the IPA is used to select whether the access is to protected memory or to memory shared with the host. The guest kernel treats this bit as if it is attribute bit (named *PROT_NS_SHARED*) in the page tables and modifies it when sharing or unsharing memory with the host. This top bit usage also necessitates that the IPA width is made more dynamic in the guest. The VMM chooses a width and therefore which bit that controls the shared flag. The guest identifies this bit to mask it out when necessary.

In order to communicate with the host via virtio, the shared buffers must be placed in memory which has this top IPA bit set. This is achieved by implementing the *set_memory*{encrypted,decrypted}_ APIs for arm64 and forcing the use of bounce buffers. For now, all IO are treated as Non-secure/shared on Arm CCA v1.0.

To generate Arm CCA attestation token, one guest Trusted Security Module (TSM) *arm-cca-guest* driver is created, which registers with the configfs-tsm module to provide user interface to retrieve an attestation token from TF-RMM. When a new report is requested, this driver invokes the appropriate RSI interfaces to query an attestation token. The token retrieval operation must be requested on the same CPU on which the attestation token generation was initialized. Attestation token generation is a long-running operation and therefore, the token data might not be retrieved in a single call, RSI call is put in one loop until the token is ready. The driver's code path is under *drivers/virt/coco/arm-cca-guest/*.

Finally, the GIC is largely emulated by the untrusted host. The TF-RMM provides some management including register save/restore, but the ITS buffers must be placed into shared memory for the host to emulate.

5.6 EDKII guest firmware for Arm CCA

EDKII firmware for Arm CCA enables booting a Linux kernel in a Realm VM with features such as ACPI-based hardware description, Virtio v1.0 support, and booting from Virtio PCIe storage. All I/O is treated as non-secure/shared.

The firmware includes updates across the PEI and DXE phases, with key components:

- PEI Libraries
  - ArmCcaInitPeiLib: Configures Protected RAM and performs initial Realm setup
  - ArmCcaLib: Provides common CCA functions (e.g., RME detection, memory protection)
  - ArmCcaRsiLib: Implements Realm Service Interface (RSI) functions
- DXE Drivers
  - RealmApertureManagementProtocolDxe: Manages buffer sharing between Realm and Host
  - ArmCcaIoMmuDxe: Supports secure DMA using shared memory buffers

The boot process includes early hooks for configuring memory protection and reading Realm configuration, with support for platform-specific MMIO protection setup via ArmVirtMemInfoLib. Null libraries are provided for platforms which lack RME support.

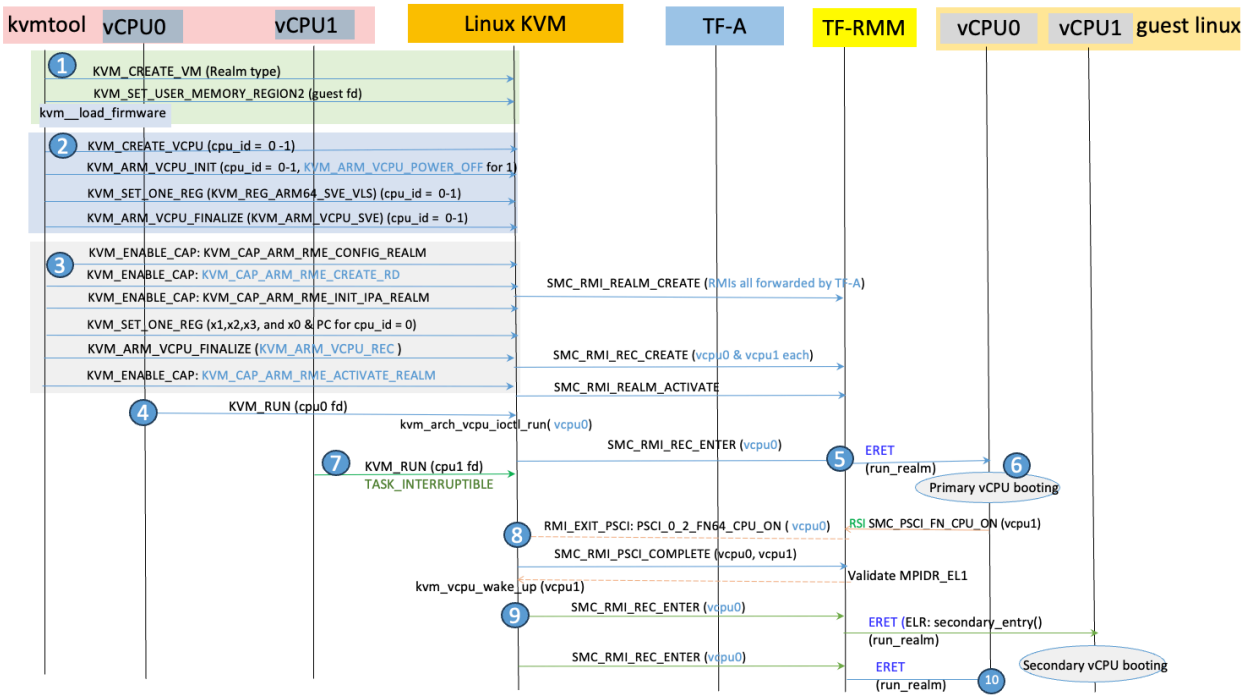
5.7 How to set up a Realm

This section describes one realm creation flow example, which explains the interaction between kvmtool, Linux KVM, TF-A, TF-RMM and guest linux in Realm. Also, the chapter Usage in the RMM describes flows between the Host, Monitor and RMM.

The following command launches a realm with two vCPUs:

```
./lkvm run -realm --restricted_mem -c 2 -m 512 --firmware KVMTOOL_EFI.fd  \
  --disk guest-disk.img --measurement-algo=sha256
```

Figure 5-5: Realm VM boot flow



Step 1: kvmtool does initialization, and creates one KVM Realm VM contex in host linux. It checks extensions like irqchip, PSCI support, and memory capability for restricted memory's case, then registers memory in host Linux and loads Realm firmware.

Step 2: kvmtool is ready to create vCPUs, and maps memory for *struct kvm_run* to facilitate the interaction between the user space and host linux. kvmtool configures vCPU features, *KVM_ARM_VCPU_REC* is set for realm VM.

Two POSIX threads to schedule vCPUs are created. VCPU 0 is the boot CPU, the other vCPU starts in a power-off state, ready to be scheduled by PSCI CPU booting flow in Realm VM.

In this step, KVM allocates VMID value for the Realm VM.

Step 3: In the last initialization list, kvmtools launches ioctl *KVM_CAP_ARM_RME* in multiple times to configure Realm parameters, delegate memory pages for realm descriptor and configure Stage 2 translation table. KVM calls *SMC_RMI_REALM_CREATE* to TF-RMM to create realm VM context in TF-RMM. Then kvmtool initializes IPA range and populates Realm memory.

During this step, the ioctl *KVM_ARM_VCPU_FINALIZE* with feature *KVM_ARM_VCPU_REC* is served in kvm to create vCPU REC context, and to call *SMC_RMI_REC_CREATE* to TF-RMM. The last thing is to activate and seal the measurement for the realm during the initialization stage.

Step 4: The POSIX thread function is mainly one loop to call *ioctl (KVM_RUN)*. It checks exit reason from KVM for proper handling, like MMIO emulation, system shutdown. The thread for vCPU0 calls this ioctl to trap into the handler in Linux KVM kernel. The handler calls *SMC_RMI_REC_ENTER* in a loop until the time slice for the process is used or some emulation is needed from user space.

Step 5: Non-Secure SMC handler in TF-RMM serves the *SMC_RMI_REC_ENTER* request. It has a sanity check, and runs guest realm software via exception return.

Step 6: Realm EDKII loads linux kernel and file system. The primary vCPU booting flow in Realm Linux looks similar as in Non-secure world VM. The Realm kernel iterates over the available memory ranges and converts the state to protected memory.

Step 7: The thread for vCPU1 is suspended in KVM since it's set with power-off state in step 2. Note: The physical CPUs for vCPU0 and vCPU1 might be different.

Step 8: The *PSCI_CPU_ON* call is trapped into TF-RMM, then routed back to Linux KVM as the scheduling of vCPU or the dynamic binding of physical CPU and vCPU is controlled by KVM. KVM needs call *SMC_RMI_PSCI_COMPLETE* to provide the references of running and target RECs to the TF-RMM, so that the TF-RMM can complete the PSCI request. This is necessary because the TF-RMM does not maintain a mapping from MPIDR values to REC address.TF-RMM validates the mapping and sets up the target REC's context including PSTATE, PC and SCTLR_EL1. After that, KVM wakes up the targeted vCPU task.

Step 9: vCPU1 task is scheduled to run. Secondary CPU boot starts via EL3 monitor and Realm EL2 TF-RMM's collaboration.

Step 10: vCPU0 task is ready to run, and the Realm execution returns from *PSCI_CPU_ON* call on guest kernel.

5.8 Hands-on with Arm CCA example Stack

You can download the whole software stack for Arm's Base AEM FVP RME platform by referring to AEMFVP-A-RME stack. For more practical information, go to Arm Learning Path portal:

Build and run the Arm CCA stack on an Arm FVP

Run an application in a Realm using the Arm Confidential Compute Architecture (CCA)

For the latest Arm CCA software, shrinkwrap build is an alternative, refer to Overview for one whole understanding and cca-4world to build the whole Arm CCA software stack with 4 worlds.

For more information on an individual software stack like TF-RMM and TF-A build, go to the following links:

building-and-running-tf-a-with-rme

Also you can build TF-RMM firmware with fake host mode if you want to learn the RMI and RSI call flow and TF-RMM in general.

5.8.1 Debug Arm CCA software stack

The setup is as following on the shrinkwrap build:

- Build TF-RMM and TF-A with Debug version

```
shrinkwrap build cca-4world.yaml --overlay buildroot.yaml  --overlay debug/rmm.yaml
 \
--overlay debug/tfa.yaml  --btvar GUEST_ROOTFS='${artifact:BUILDROOT}'
```

- Create one yaml named *debug.yaml* under shrinkwrap config folder

```
run:
  params:
    '--iris-connect tcpserver,allowRemote,port=7100': null
    '--print-port-number': null
```

- Run command

```
shrinkwrap run cca-4world.yaml --rtvar ROOTFS=rootfs.ext2 \
   --rtvar CMDLINE="mem=1G earlycon nokaslr root=/dev/vda ip=dhcp acpi=force" \
   --overlay=debug.yaml
```

- The FVP pauses and we need record the IRIS debug port to be attached to Arm DS during running the model.
- On Arm DS debugger, create one Debug Connection to collect the port and with the IP address found from the console log.

6. 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
- edk2 Arm CCA guest firmware patches
- arm64: Support for running as a guest in Arm CCA
- arm64: Support for Arm CCA in KVM

7. 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