Cross-platform **O**pen **S**ecurity **S**tack for C**on**nected Device

D5.7 CROSSCON Stack Installation and Usage Guidelines

Document Identification			
Status	Final	**Due Date**	30/10/2025
Version	1.0	**Submission Date**	28/11/2025

Related WP	WP5	**Document Reference**	D5.7
Related Deliverable(s)	D1.5, D2.3, D3.3, D3.4, D4.3, D4.4	**Dissemination Level (*)**	PU
Lead Participant	CYSEC	**Lead Author**	Malvina Catalano Gonzaga
Contributors	BEYOND, ATOS, TUD, UMINHO, UNITN, UWU	**Reviewers**	3MDEB, SLAB

Keywords:
CROSSCON Stack, Deployment Architecture, Installation, Usage Guidelines.

Document Information

List of Contributors	
Name	Partner
Malvina Catalano Gonzaga	CYSEC
Yannick Roelvink	CYSEC
João Miguel Costa Sousa	UMINHO
Ziga Putrle	BEYOND
Hristo Koshutanski	ATOS
Alberto Tacchella	UNITN
Tymoteusz Burak	UWU
Fabian Schmitt	UWU
Nikhilesh Kumar Singh	TUD

Document History			
Version	Date	Change editors	Changes
0.1	7/08/2025	Yannick Roelvink	Initial version, document layout definition and allocation of sections to contributors
0.2	23/09/2025	João Miguel Costa Sousa	Integration of Section 3.1
0.3	25/09/2025	Malvina Catalano Gonzaga	Integration of Chapters 1 and 2
0.4	26/09/2025	Ziga Putrle	Integration of Section 3.5
0.5	02/10/2025	Nikhilesh Kumar Singh	Integration of Section 3.4.5
0.6	03/10/2025	Hristo Koshutanski	Integration of Section 3.4.6
0.7	05/10/2025	Alberto Tacchella	Integration of Sections 3.2, 3.3 & 3.4.4
0.8	06/10/2025	Tymoteusz Burak and Fabian Schmitt	Integration of Sections 3.4.1, 3.4.2 & 3.4.3
0.8a	9/09/2025	Malvina Catalano Gonzaga Yannick Roelvink	Add executive summary and conclusion. Perform document cleanup and submit for internal review
0.8b	21/10/2025	Malvina Catalano Gonzaga	Integrating SLAB and 3MBED review comments
0.9	27/11/2025	Juan Alonso (ATOS)	Quality Assessment.
1.0	28/11/2025	Hristo Koshutanski (ATOS)	Final version submitted.

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Malvina Catalano Gonzaga (CYSEC)	21/10/2025
Quality manager	Juan Alonso (ATOS)	27/11/2025
Project Coordinator	Hristo Koshutanski (ATOS)	28/11/2025

Table of Contents

List of Figures

List of Tables

List of Acronyms

Abbreviation / acronym	Description
API	Application Programming Interface
APU	Application Processing Unit
BCM	Behavioral Certification Manifest
CA	Client Application
CBA	Context-Based Authentication
CPU	Central Processing Unit
CSI	Channel State Information
DL	Deep Learning
FPGA	Field-Programmable Gate Array
FS	File System
GDB	GNU Debugger
GIC	Generic Interrupt Controller
GICC	GIC CPU Interface
GICD	GIC Distributor
GICR	GIC Redistributor
GPL	General Public license
GPOS	General Purpose Operating System
HV	Hyperviser
HW	Hardware
I/O	Input/Output
IOMMU	Input/Output Memory Management Unit
IoT	Internet of Things
IPC	Inter-Partition Communcation
JTAG	Join Test Action Group
MCU	Microcontroller Unit
MMIO	Memory-Mapped I/O
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSP	Microcontroller Subsystem Platform
mTLS	Mutual Transport Layer Security
OP-TEE	Open Portable Trusted Execution Environment
PA	Physical Address
PCB	Printed Circuit Board
PLIC	Platform-Level Interrupt Controller
PMP	Physical Memory Protection
PUF	Physical Unclonable Function
RPi	Raspberry Pi
RTU	Real Time Unit
SDEES	Software-Defined Execution Environments
SGX	Software Guard Extensions

SMMU	System Memory Management Unit
SW	Software
TA	Trusted Application
TCM	Trusted Computing Module
TEE	Trusted Execution Environment
UART	Universal Asynchronous Receiver-Transmitter
VA	Virtual Address
VM	Virtual Machine
TRNG	True Random Number Generators

Executive Summary

The CROSSCON Stack Installation and Usage Guidelines provide a comprehensive technical overview and practical instructions for deploying and utilizing the CROSSCON security stack, a modular, cross-platform open security solution for connected devices. This deliverable supports stakeholders and developers in integrating CROSSCON technologies across a range of heterogeneous environments, from low-end microcontrollers to high-end system-on-chip platforms.

The CROSSCON stack unifies key security components, including:

- The **CROSSCON Hypervisor**, which ensures secure isolation and partitioning in mixed-criticality systems.
- The **Bare-Metal Trusted Execution Environment (TEE)** for resource-constrained devices.
- The **TEE Toolchain** for secure updates, verification, and attestation processes.
- The **Trusted Applications** that provide features such as PUF-based authentication, context-aware security, control-flow integrity, and secure FPGA provisioning.
- The **CROSSCON SoC** for implementing hardware-assisted security primitives.

The document serves as a complete guide for installation, configuration, and operation of these elements. It offers reference configurations and practical examples to support system integrators and developers in replicating or extending CROSSCON deployments.

This deliverable also bridges work across the project's technical work packages, consolidating architecture, implementation, and validation activities, to ensure consistent interoperability and robust deployment of the CROSSCON ecosystem. The unified view of the stack showcases how the project meets its assumptions and requirements: enabling secure, interoperable, and trustworthy operation of IoT systems across diverse hardware platforms.

Deliverable D5.7 contributes to the accomplishment of milestone MS10 "Final CROSSCON stack validation in use case scenarios, final exploitation plan and community building results".

1 Introduction

1.1 Purpose of the document

The purpose of this report is to provide comprehensive guidelines for the installation, integration, and usage of the CROSSCON security stack. The CROSSCON stack has been developed to address the heterogeneity of Internet of Things (IoT) devices, ranging from low-end microcontroller units to high-end application processors, and to ensure secure, interoperable, and reliable deployment across diverse environments.

This report builds upon the collective work performed throughout the project, including requirements elicitation, architectural design, implementation, and validation activities. It consolidates these outcomes into practical guidance, ensuring that the CROSSCON stack can be effectively deployed and operated in real-world scenarios.

1.2 Relation to other project work

This deliverable is connected to and builds upon several other project outputs. It draws from the requirements and specifications defined in **D1.5 – Final Requirements and Technical Specifications [1]**, as well as the architectural design provided in **D2.3 – CROSSCON Open Specification (Final) [2]**.

It also complements the outcomes of:

- **WP3 (D3.3 [3], D3.4 [4])**, which focuses on the development of core components of the CROSSCON stack such as the hypervisor and trusted execution environments.
- **WP4 (D4.3 [5], D4.4 [6])**, which address the integration of trusted applications, secure storage, unique identifiers, and hardware features.
- It directly supports the validation and demonstration work reported in **WP5**, ensuring that the installation and usage guidelines are aligned with the testing and deployment of the CROSSCON stack.

By consolidating insights from these deliverables and outcomes, the present report provides a practical set of guidelines for deploying, configuring, and operating the CROSSCON security stack in heterogeneous environments.

1.3 Structure of the document

This report is organized into five chapters. Chapter 1 introduces the purpose, context, and structure. Chapter 2 presents the deployment architectures of the CROSSCON stack, covering device classification and the different instantiation models. Chapter 3 provides detailed installation and usage instructions for CROSSCON components including best practices for integration and operation. Chapter 4 concludes with recommendations and next steps.

This document has been designed to function as a standalone document, offering complete guidance without requiring direct consultation of other deliverables.

2 CROSSCON Stack Deployment Architectures

2.1 Device Class Definition

CROSSCON adopts a **security-centric** taxonomy to cope with heterogeneous IoT hardware. While the project recognises both *security capabilities* (what the device can enforce) and *security guarantees* (what the use case requires), the practical **class assignment is based exclusively on built-in hardware/firmware security capabilities**. These classes indicate how effectively a device can support secure services; actual guarantees still depend on correct policies and software. The device classes, defined in D1.5 are as follow:

- **Class 0 - NO SECURITY**
  Devices that have **no built-in security capabilities.** These are normally devices that respect ultra-low power and low costs constrains and are therefore not adequate to perform critical functions. These devices need to rely entirely on software-based security, which makes them the most vulnerable of all classes.
- **Class 1 - BASIC SECURITY**
  Resource constrained devices that include **basic hardware security capabilities** such as memory protection via MPU and basic privilege system. While these devices may have a better secure stack than Class 0 devices, they are still vulnerable to specific attacks.
- **Class 2 - STRONG SECURITY**
  Devices that already include **integrated or discrete hardware functions with security capabilities** (e.g., secure processing, crypto acceleration, data/peripheral protection). These are suited for applications that need robust embedded protections without the overheads of high-end platforms.
- **Class 3 - EXTENDED SECURITY**
  Devices for **high-security environments** (e.g., critical infrastructure, military applications, secure communications) with the most advanced security capabilities such as subsystem isolation, True Random Number Generators (TRNG), Physically Unclonable Functions (PUFs), or hardware-based intrusion detection.

While Class 0 and Class 1 devices could most likely be represented by devices that do not mount many security capabilities, Class 2 and Class 3 are expected to be rich with those, boosting the CPU baseline security capabilities.

It's worth mentioning that these classes are not necessarily absolute and there may be some overlap between them. Additionally, security is a complex and evolving field, so the classification may change over time as new threats emerge and new security capabilities are developed, as the developed device classification system is based on current technology and security standards.

2.2 Instantiation Options

Based on the software components running on CROSSCON's selected architectures and the hardware security primitives they utilize, various system stack configurations are possible. Various configurations are categorized into a set of CROSSCON instantiation options, ranging from environments without hardware security primitives to those featuring virtualization, TEE enforcement, and FPGA-assisted reconfigurable hardware.

A total of six instantiation options are defined. Notably, all environments, even those lacking hardware security primitives or dedicated TEE technologies, support the execution of multiple trusted applications.

2.2.1 Software-Only Isolation Environment

This environment targets resource-constrained, low-end devices lacking hardware security primitives.

According to the WP1 device classification, these are identified as **Class 0 devices**. Platforms in this

classification operate with a single privileged level. Therefore, CROSSCON employs a software-based approach to ensure isolation between standard and trusted applications. Figure 1 illustrates the system stack diagram of the MSP architecture, highlighting the software components responsible for implementing software-based isolation. The CROSSCON BareMetal TEE is one of the system components being developed in the context of WP3; it is described in more detail in Section 3.2.

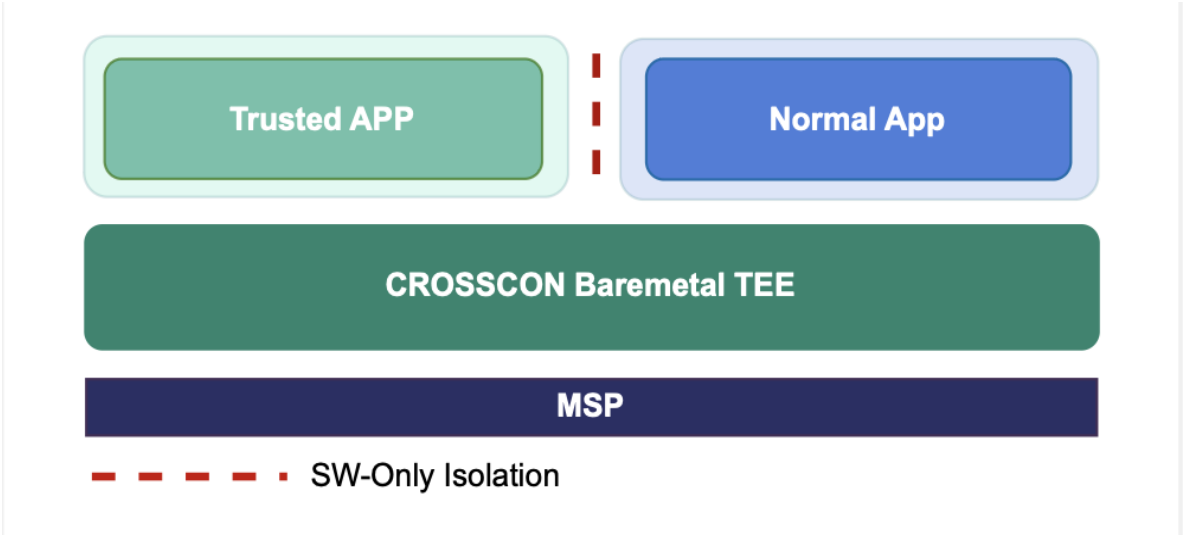


Figure 1: Multiple Isolated Environments on MSP Architecture Using SW-Only Isolation

2.2.2 Basic Memory Isolation Environment

This environment targets resource-constrained devices equipped with basic hardware security primitives, i.e., MMU and PMP. According to the WP1 device classification, these are identified as **Class 1 devices**. Platforms in this classification operate with two privilege levels: privileged and non-privileged. Software components running at the privileged level are responsible for configuring the basic hardware security primitives, such as the MPU in ARM devices and the PMP unit in RISC-V devices. As depicted in Figure 2, trusted and normal applications are isolated using these hardware security primitives. For such platforms, the CROSSCON BareMetal TEE is the software component responsible for maintaining the isolation between these applications.

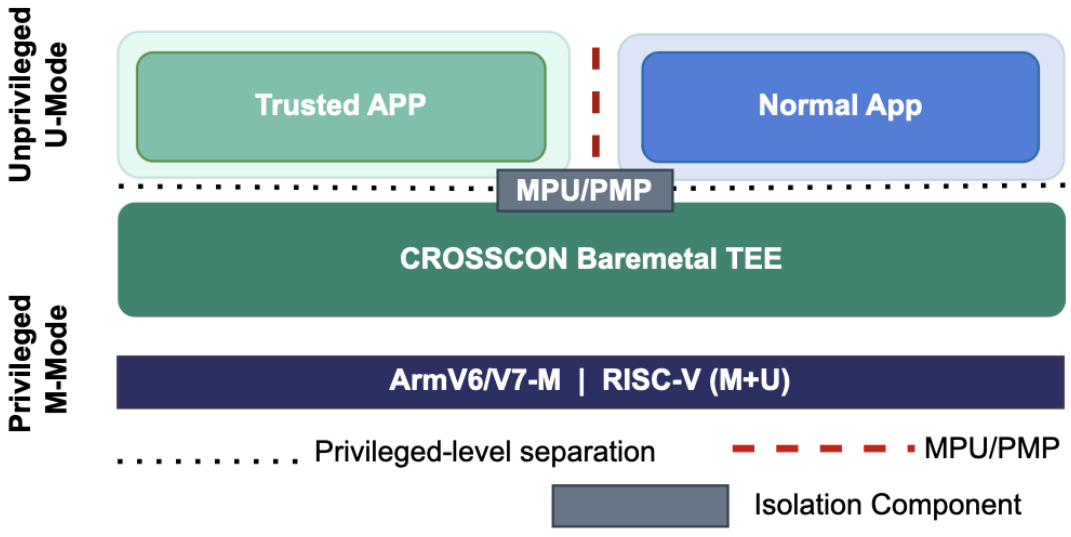


Figure 2: Multiple isolated environments on Armv6-M/v7-M architecture using basic memory isolation primitives

2.2.3 Virtualization-less Environment with TEE

This environment is tailored for devices equipped with hardware security primitives that can implement multiple isolated environments. In certain MCUs CROSSCON leverages TEE technologies to facilitate the multi-isolated and secure execution of trusted environments. According to the WP1 device classification, **Class 2 devices** can provide these environments, specifically the MCU devices with TrustZone-M support. Figure 3 shows an example of a system stack featuring multiple isolated environments on an Armv8-M architecture using TrustZone-M.

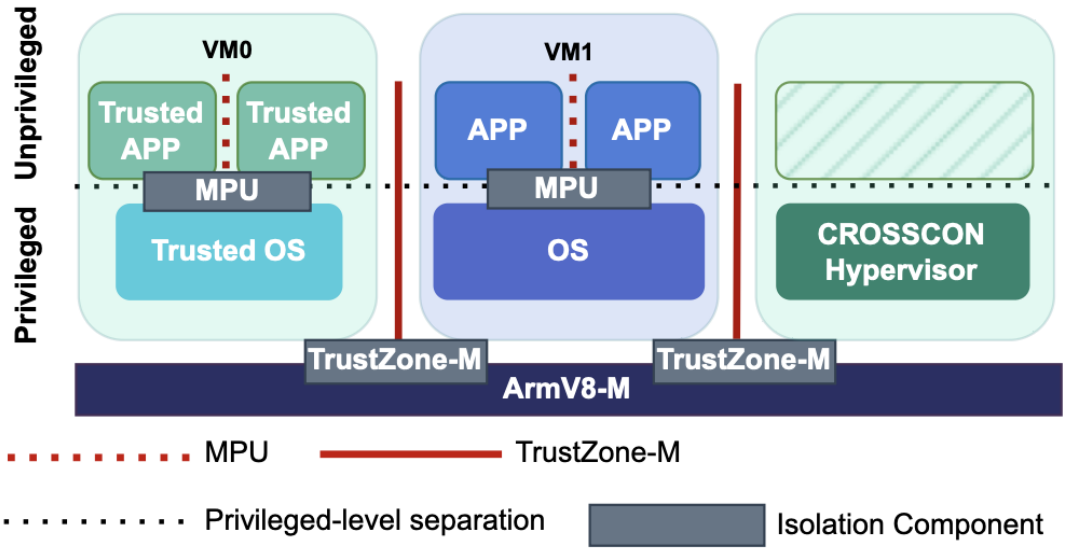


Figure 3: Multiple isolated environments in Armv8-M architecture using TrustZone-M assisted security primitive

2.2.4 TEE-less environment with Virtualization

This environment is designed for devices equipped with security hardware primitives that support hypervisor implementation but lack dedicated TEE hardware. To enable the execution of multiple trusted applications in such environments, CROSSCON utilizes VMs as independent trusted environments. Based on the WP1 device classification, both **Class 2 and Class 3 devices** can support these environments. Depending on the architecture, these environments can feature either three or four privileged levels, with a dedicated level for the hypervisor to manage VMs.

- In Armv7-A/V8-A and RISC-V (CVA6), the hypervisor operates above the firmware privileged level and leverages MMU second-stage virtual memory to enforce isolation between VMs.
- In RISC-V (BA51-H), the hypervisor does not rely on virtual memory for guest isolation. Instead, it uses PG to control physical memory access between VMs and external modules within the SoC.
- In Armv8-R, the hypervisor operates at the highest privileged level (see Figure 4) and enforces physical memory isolation using hardware security primitives, such as second-stage MPUs. Application-level isolation is achieved through basic security primitives, including the MMU, and MPU

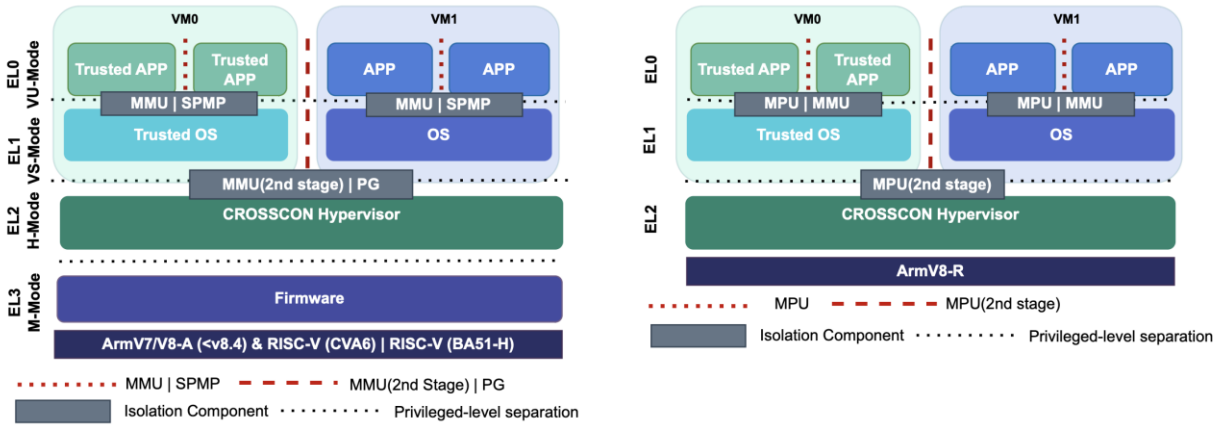


Figure 4: Multiple isolated environments on both APU (Armv7-A/V8-A and CVA6) and RTU (Armv8-R and BA51-H) devices of ARM and RISC-V architectures

2.2.5 Environment with TEE and Virtualization

This environment, like previous instantiation options, is designed for devices with hardware security primitives that support multiple isolated execution contexts. These devices leverage TEE technologies and virtualization mechanisms to enhance security. According to the WP1 device classification, **Class 2 and 3 APU devices** can implement these environments. APU architectures, such as Armv7-A/V8-A, utilize virtualization primitives (e.g., MMU) or TEE-assisted technologies (e.g., Arm TrustZone).

Figure 5 illustrates the hypervisor's role in managing multiple trusted services within isolated VMs.

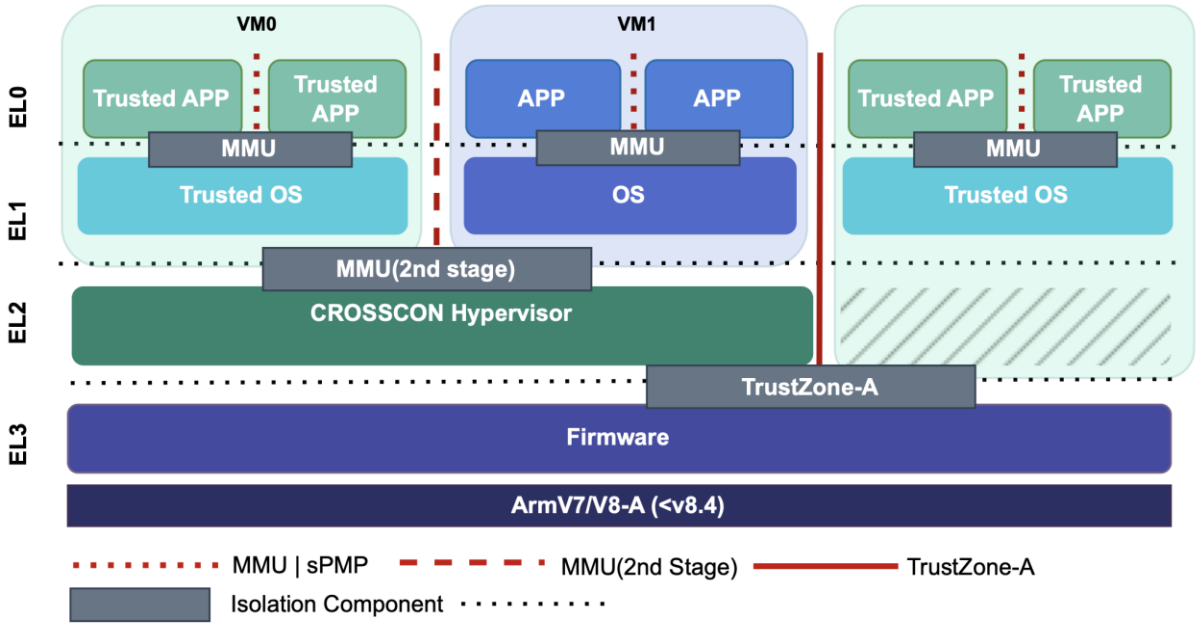


Figure 5: Multiple environments supporting multiple isolated execution contexts using hardware security primitives, including virtualization and TEE technologies on APU (Armv7-A/V8-A) devices

2.2.6 Environment with Virtualization and FPGA

This environment is designed for APU devices that integrate virtualization security primitives and FPGA capabilities. Virtualization ensures the secure execution of multiple isolated trusted VMs, while FPGA capabilities enable multi-tenancy services for trusted applications within these VMs. As part of CROSSCON, FPGA services include resource redistribution between tenants and the concept of virtual FPGA (vFPGA) (developed in UC5), as shown in Figure 6. To support multiple isolated trusted VMs, the Hypervisor operates at a dedicated privileged level on ArmV8-A/V7-A architectures, leveraging second-

stage MMU for isolation. Under these conditions, the Hypervisor manages which trusted VM can access FPGA-trusted services. Devices that support this environment are classified as **Class 3 devices** in the WP1 classification.

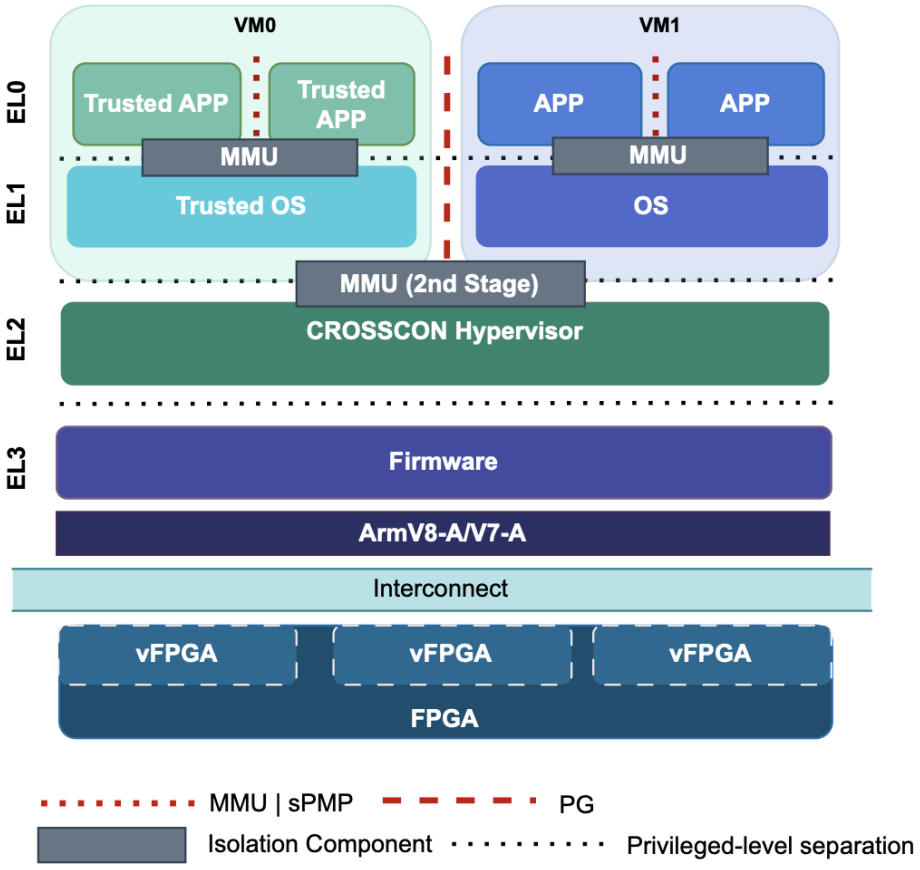


Figure 6: System stack representation of Armv6-M/v7-M architecture together with basic memory isolation primitives.

3 CROSSCON Stack Installation & Utilization

3.1 CROSSCON Hypervisor

CROSSCON Hypervisor, built upon Bao [15], is grounded in a lightweight, open-source and static partitioning hypervisor that aims to provide strong isolation and real-time guarantees. Its static partitioning designs targets mixed-criticality systems focusing on isolation for fault-containment and real-time behaviour. To implement the static partitioning hypervisor architecture:

1. resources are statically partitioned and assigned at VM instantiation time.
2. memory is statically assigned using 2-stage translation.
3. IO is pass-through only.
4. virtual interrupts are directly mapped to physical ones.
5. it follows a 1-1 mapping of virtual to physical CPUs (with no need for a scheduler).

In addition to the inherited architectural isolation features, CROSSCON Hypervisor also offers microarchitectural-level isolation, implementing mechanisms, such as cache coloring, to maintain the isolation of shared resources such as last-level caches. However, the static partitioning hypervisor design come with its own set of challenges that limit their applicability in IoT systems. Such limitations encompass (i) the absence of dynamic VM creation and management, and (ii) the incapacity to deliver per-VM TEE services. These challenges will be addressed by enhancing the static partitioning design of the Bao.

Dynamic-VM feature: The CROSSCON Hypervisor supports dynamic virtual machines, allowing VMs to be created, modified, or removed at runtime. In this mode, multiple VMs can still execute concurrently on a single CPU, with the additional ability to change the VM set without rebooting or reconfiguring the hypervisor. For this, CROSSCON Hypervisor leverage VM-stack mechanism detailed in D3.3. Additionally, to enable the creation of new VMs during run-time VM execution, VMs need to access to the hypervisor interface that enables them to send a config file to the hypervisor, i.e., through the VM create hypercall. The CROSSCON Hypervisor then parses this file to instantiate the new child VM. After this, the hypervisor proceeds to instantiate the child VM, while removing all resources, except for the physical CPUs, from the parent VM.

Dynamic VMs require that a CROSSCON Hypervisor driver on the host OS interacts with the CROSSCON Hypervisor through the hypervisor call interface. This interface serves three main objectives: VM Creation, VM Destruction, and VM Invocation. For more detailed explanation refer to D3.3 [3].

Per-VM TEE feature: The CROSSCON Hypervisor is designed to host trusted OSes, which traditionally run using TEE technologies, inside individual VMs. Instead of a single trusted OS executing in the secure world, multiple trusted-OS VMs can run in the normal world. This decomposes the trusted environment into smaller, mutually isolated units, which reduces the overall Trusted Computing Base (TCB), shrinks the attack surface, and ensures fault containment (i.e., a compromise of one trusted OS does not affect the others). As with the Dynamic-VM capability, per-VM TEE support relies on the VM-stack mechanism to bind each Trusted OS to a corresponding GPOS VM. At boot time, the hypervisor parses a configuration file to instantiate multiple GPOS / Trusted-OS pairs. As part of the CROSSCON project, OP-TEE has been integrated with the CROSSCON Hypervisor to run inside VMs on both Arm and RISC-V platforms. This enables per-VM TEE services by splitting a single monolithic TEE into multiple isolated per-VM TEEs. A detailed description of this mechanism is provided in Deliverable D3.3 [3].

Table 1: List of platforms supported by the CROSSCON Hypervisor

Architecture	Platform	Name used for compilation process
Arm	Qemu virt	qemu-aarch64-virt
	Raspberry Pi 4B	rpi4
	Xilinx Zynq UltraScale+ MPSoC ZCU102/104	zcu104
	LPCXpresso55S69 Development Board	lpc55s69
RISC-V	QEMU virt (RV64)	qemu-riscv64-virt

3.1.1 CROSSCON Hypervisor and VM Configuration

This section demonstrates how to configure the different VMs of the CROSSCON Hypervisor. The same content is also included in deliverable D2.4. It is presented here to serve as part of the user manual, ensuring that end-users can set up and use the CROSSCON Hypervisor without needing to consult other deliverables.

How to configure VMs on top of CROSSCON Hypervisor?

The CROSSCON Hypervisor's configuration is managed through a dedicated configuration in the form of a C source file. The configuration files for the CROSSCON Hypervisor are stored in a designated directory known as the configuration repository, identified by the make variable CONFIG_REPO. By default, the CONFIG_REPO is set to the configs directory located in the top-level directory of the CROSSCON Hypervisor. However, users have the flexibility to specify a different folder by setting the CONFIG_REPO option in the make command during the Hypervisor build process. For instance, a typical build command for CROSSCON Hypervisor would be:

```
make PLATFORM=target-platform CONFIG_REPO=/path/to/config CONFIG=config-nam
```

Taking a configuration named *config-name* as an example, the configuration source file can be located in the *CONFIG_REPO* directory in two formats:

1. **Single C Source File:** A C source file with the name *config-name.c*.
2. **Directory Format:** A directory named *config-name* with a single *config.c* file within it.

Next, the description of the various configuration options available to modify this C source file are presented. See the C source file example in following snap-code.

```
#include <config.h>

// Load guests' image
VM_IMAGE(img1_name, "/path/to/vm1/binary.bin");
VM_IMAGE(img2_name, "/path/to/vm2/binary.bin");

struct vm_config vm1 = { /* VM 1 Config*/ }
struct vm_config vm2 = { /* VM 2 Config*/ }

struct config config = {
// Shared memory region configuration
.shmemlist_size = N,
.shmemlist = (struct shmem[])
{
      [0] = {/*shared memory config*/,},
      [1] = {/*shared memory config*/,}, ...
      [N]           =           {/*shared           memory           config*/,}
},                    //                  Guests                  Configuration
.vmlist_size  =  NUM_VMs,  .vmlist  =  {  {  &vm1  },  {  &vm2  },  ...  }
};
```

The configuration file requires a global variable named *config* of the type *struct config*, which contains two distinct lists: (i) a list of shared memory regions (*shmemlist*) and (ii) a list of VMs (*vmlist*). The list of shared memory regions is optional and may be omitted from the configuration. The list of VMs is mandatory and must include at least one instance. Additionally, for each list, it is necessary to specify the list size using the parameters *shmemlist_size* and *vmlist_size.* Inconsistencies between the list size and the actual size may result in unpredictable behavior.

VM Configuration

After configuring the overall setup, each defined VM must also be configured. The first step is to ensure that the VM is loaded onto the platform. This can be achieved either by compiling and loading the VM separately, or by embedding it within the hypervisor binary. When embedding VM images into the hypervisor binary it is necessary to declare the VM images using the *VM_IMAGE* macro in configuration file. Here's an example usage of the *VM_IMAGE*:

```
VM_IMAGE(img_name, "/path/to/VM/binary.bin");
```

The VM_IMAGE macro has two parameters:

1. The *img_name,* a unique identifier associated with the image that will later be used to describe the image running on the VM.
2. A C string with the guest image's binary file path. It can be either an absolute path or a path relative to the config source file.

Next, to configure VM properties one should fill a *vm_config* struct with required parameters. For each VM defined in *vmlist* a *vm_config* struct is needed.

The CROSSCON Hypervisor configuration file enables precise partitioning of hardware resources, including CPU cores, memory, and I/O devices, by assigning them to specific VMs. All resources are exclusively allocated, ensuring strict isolation between VMs. To configure the partitioning of resources, configuration file provides several parameters to be configured inside *vm_config* struct. See the following snippet of code for get familiar with example parameters.

```
struct vm_config {
  struct {
      vaddr_t base_addr;
      paddr_t load_addr;
      size_t size;
      bool separately_loaded;
      bool inplace;
  }image;
  vaddr_t entry;
  cpumap_t cpu_affinity;
  colormap_t colors;
  size_t type;
  size_t children_num;
  struct vm_config **children;
  struct vm_platform platform;
};
```

In summary, each *vm_config* entry defines its VM image, memory address, CPUaffinity, color mapping, and platform details. For each VM, the following parameters must be specified:

- **image [mandatory]** - a structure containing details about guest image loading (*base_addr*, *load_adddr*, *size*, etc.)
- **entry [mandatory]** - defines the entry point address in the guest's address space.
- **platform [mandatory]** - a description of VM platform: resource assignments and requirements.
- **cpu_affinity [optional]** - represents a bitmap that signals the preferred physical CPUs assigned to the VM. If this value is mutually exclusive across all VMs, the physical CPUs assigned to each VM will follow the bitmap. Otherwise (in case of bit overlap or lack of affinity definition), the hypervisor will use a scheduler to allow multiple vCPUs to share a physical CPU core. See following Figure 7 as a representation.

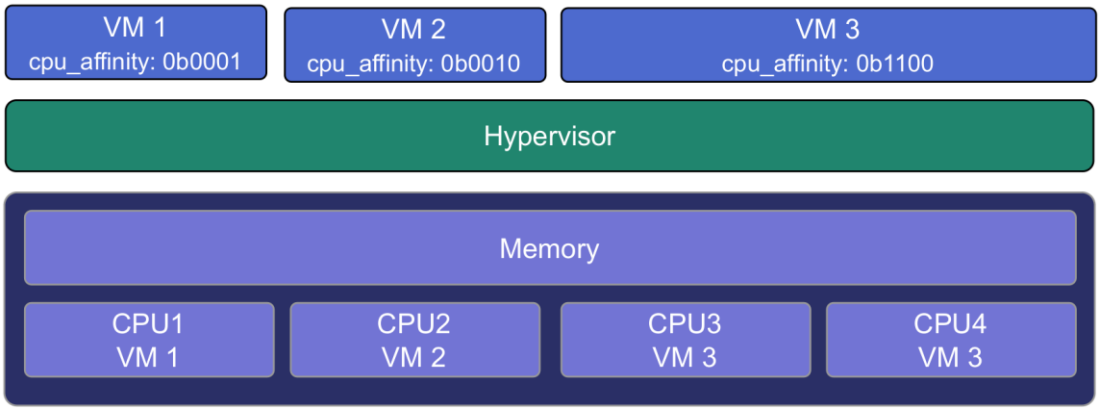


Figure 7: Configuration example involving 3 VM with different cpu_affinity configurations

- **colors [optional]** - Implements cache coloring technique and partitions LLC sets among different VMs with goal of minimizing cache confits and enhance overall system performance. This parameter represents a bitmap signaling cache colors of the VM. This value is truncated depending on the number of available colors calculated at run-time, i.e., it is platform dependent. The coloring mechanism is disabled by default. Use the following Figure 8 as an example.

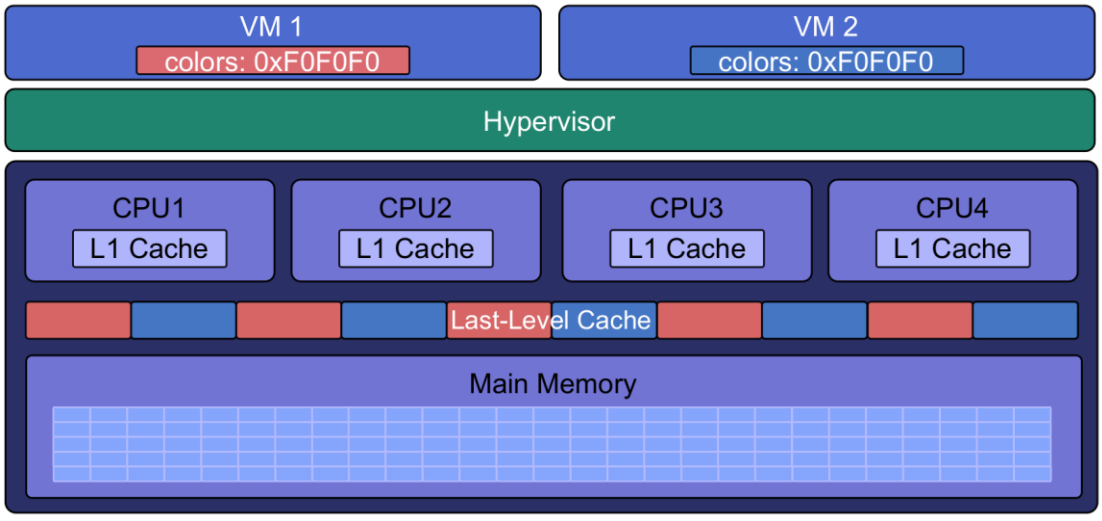


Figure 8: Configuration example involving 2 VM with different cache coloring configurations

Note: Cache coloring relies on careful assignment of colors to each VM. However, this mechanism may not work if the physical mapping feature is enabled for a specific memory region. Cache coloring exclusively operates in virtual memory systems, i.e., systems featuring MMUs for address translation.

- **type** - Describes the type of VM. Types include `CROSSCON_VM_ORDINARY` (regular VMs), `CROSSCON_VM_SDTZ` (TEE VMs following the TrustZone TEE model), `CROSSCON_VM_SDSGX`

(TEE VMs following the SGX TEE model), and `CROSSCON_VM_DYNAMIC` (VMs that will be managed dynamically). If no value is defined, type is `CROSSCON_VM_ORDINARY`.

- **children_num -** Number of child VMs (this property will be reffered later when using Dynamic VM and Per-VM TEE features).
- **children -** VMs can be configured in an execution control hierarchy, establishing parent-child relationship. This is used for sdTZ, sdSGX and dynamic VM management. This field enables VMs to declare multiple children in an array.

Moving directly to the configuration of the ***image* parameter** within each *`vm_image`*, this step involves defining several sub-parameters. These include:

- **base_addr [mandatory] -** corresponds to the image load guest address.
- **load_addr [mandatory]** - corresponds to the image load physical address. This value can be defined using the macro *`VM_IMAGE_OFFSET (img_name)`*.
- **size [mandatory] -** corresponds to the image size. For builtin images declared using *VM_IMAGE*, this value can be defined using the macro *`VM_IMAGE_SIZE (img_name)`*.
- **separately_loaded [optional] -** informs the hypervisor if the VM image is to be loaded separately by a bootloader. By default, *separately_loaded* is set as false.
- **inplace [optional] -** use the image inplace and don't copy the image. By default, inplace is set as false.

Attempting to manually configure *image* details may result in errors or undefined behavior. To ensure accurate and efficient configuration it is strongly recommended to leverage the designated macros provided by CROSSCON Hypervisor. These macros, namely *`VM_IMAGE_BUILTIN`* and *`VM_IMAGE_LOADED`*, are specifically designed to simplify the image configuration process, and enhance compatibility with the hypervisor.

1. ***VM_IMAGE_BUILTIN*** - This macro simplifies image configuration by requiring only the *`img_name`* and the image *`base_addr`*. This macro handles both the base address and image size by itself.
2. ***VM_IMAGE_LOADED* -** This macro requires additional configurations. It requires the definition of image *`base_addr`*, the image *`load_addr`*, and the image size. Using described macros not only streamlines the configuration steps, but also ensures adherence to the correct syntax and parameters.

See the next Figure 9 as a reference for their use. Each guest (or VM) is configured with *`separately_loaded`* set as false (on the right) and true (on the left).

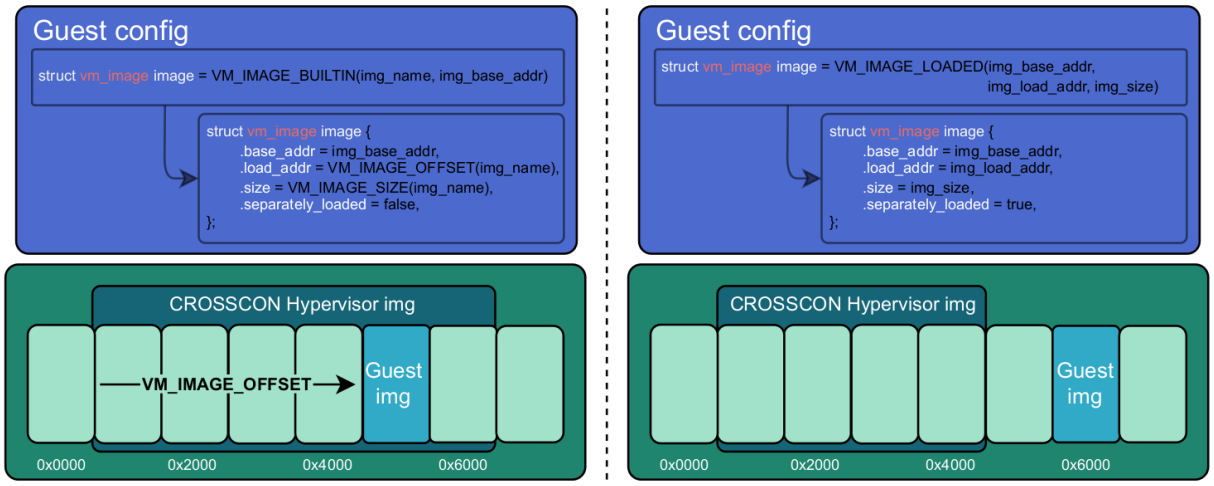


Figure 9: Configuration example involving CROSSCON Hypervisor configuration macros

The integration of the appropriate macro, tailored to one's specific use case, is crucial for ensuring consistency and reliability in one's VM setup. For instance:

- **IMAGE_BUILTIN**: Simplifies system configuration by letting CROSSCON Hypervisor determine image location automatically. No separate configuration or loading of guest images through a bootloader is required, and adjustments to the size of guest images are unnecessary.
- **IMAGE_LOADED**: Highly recommended, especially for MPU systems, where manual allocation of space for the guest image can be challenging if image is embedded into CROSSCON Hypervisor's binary. Without utilizing LOADED, CROSSCON Hypervisor will copy the image, potentially resulting in wasted space.

Moreover, if the *separately_loaded* parameter is configured as false, the hypervisor interprets this setting as an offset of the built-in guest image within its own image, denoted as *VM_IMAGE_OFFSET*. During run-time, the hypervisor uses this value to calculate the physical address. This calculation involves adding the address at which the hypervisor itself was loaded. However, if the *separately_loaded* parameter is configured as true, the guest image is not embedded in the hypervisor image; instead, it is loaded independently.

Just as image parameter ***platform*** parameter also contains several sub-parameters. By customizing this configuration, users can set up virtual platform to suit specific workload requirements and application needs for their VMs. The configuration includes the definition of:

1. Number of CPUs
2. Memory regions
3. Inter-Partition Communication (IPC)
4. Devices
5. Architectural-Specific Configurations

Number of vCPUs:

- **cpu_num [mandatory] -** defines the number of CPUs assigned to the VM.

Warning: Ensure that the cumulative count of CPUs allocated across all VMs listed in the *vmlist* does not exceed the total number of available CPUs on the platform. Failing to obey this requirement might result in the guest failing to boot without any warning.

Memory Regions:

For each VM, users can define multiple memory regions. To facilitate this, users first define the total number of memory regions via the *region_num* parameter:

- **region_num [mandatory]** - defines the number of memory regions in the VM, specifically, the number of *vm_mem_region* entries in the *vm_platform*'s regions list.

Then, each memory region is described by populating the struct *vm_mem_region*:

```
struct vm_mem_region {
      paddr_t base;
      size_t size;
      bool place_phys;
      paddr_t phys;
};
```

where:

- **base [mandatory]** - corresponds to the base guest address of the memory region.
- **size [mandatory] -** corresponds to the size of the memory region.

Note: It is mandatory for *base* and *size* to align with the smallest page size of the architecture. For MMU systems, this typically aligns to 4K, while for MPU systems, it aligns to 64 bytes.

- **place_phys [optional] -** the memory region is mapped on the virtual memory. It is important to note that the Virtual Address (VA) might not necessarily be the same as the Physical Address (PA). When

place_phys is set to true, the guest address corresponds to the physical address. This allows to specify the physical address of the memory region. By default, *place_phys* is set to false.

- **phys [mandatory if place_phys is true] -** it corresponds to the physical address where the memory region should be mapped.

Note: For enhanced performance, especially in MMU-based targets, it is recommended to align *base* and *size* to the architecture specific huge pages (e.g., 2MiB for Arm and RISC-V). Similarly, if *place_phys* is enabled, aligning *phys* to the architecture specific huge pages can also improve performance. In MPU systems, *place_phys* and *phys* are ignored. The usage of *place_phys* and *phys* allows users to manually allocate memory and obtain physical mappings. This feature provides a way to explicitly define the physical memory region.

Inter-Partition Communication (IPC):

Inter-Partition Communication (IPC) enables communication between distinct partitions in a computing system, facilitating data exchange, synchronization, and coordination between partitions. CROSSCON Hypervisor provides support for IPC, allowing VMs to establish communication channels between themselves. The IPC configuration involves defining the number of IPCs using the *ipc_num* field within the *vm_platform* struct. The specifics of each IPC are then outlined through the ipcs structure, including *base*, *size*, *shmem_id*, *interrupt_num*, and *interrupts*.

- **ipc_num [optional] -** defines the number of IPCs assigned to the VM. By default, *ipc_num* equals zero.
- **ipcs [mandatory if ipc_num > 0]** - vector of *ipc structs* that corresponds to the IPC specification and is configured through the following structure:

```
struct ipc {
	paddr_t base;
	size_t size;
	size_t shmem_id;
	size_t interrupt_num;
	irqid_t *interrupts;
};
```

where:

- **base [mandatory]** - corresponds to the base guest address of the IPC memory region.
- **size [mandatory]** - corresponds to the size of the IPC memory region.

Note: The size field must be less than or equal to the size of the shared memory. Additionally, for MPU systems, the *base* field is ignored, as the region address is the same as the shared memory object address. Also, it is mandatory for both base and size to be aligned with architecture specific smallest page size. For MMU systems, this corresponds to 4K for all architectures, while for MPU systems, the alignment corresponds to 64 bytes.

- **shmem_id [mandatory]** - corresponds to the ID of shared memory associated with the IPC.
- **interrupt_num [mandatory] -** defines the number of interrupts assigned to the IPC.
- **interrupts [mandatory if interrupt_num > 0] -** defines a list of interrupt IDs assigned to the IPC - (irqid_t[]) {irq_1, ..., irq_n}.

Note: Specifying a number of interrupts in the *interrupts* buffer that differs from the *interrupt_num* may result in undefined behaviour.

Devices:

Devices are also resources configured as part of *vm_platform.* Globally, they include:

- **dev_num [mandatory] -** corresponds to the number of device MMIO regions assigned to the VM.
- **devs [mandatory if dev_num > 0] -** vector of devices that corresponds to the specification of the VM's devices and is configured through the following *vm_dev_region* structure:

```
struct vm_dev_region {
	paddr_t pa;
```

```
        vaddr_t va;
        size_t size;
        size_t interrupt_num;
        irqid_t *interrupts;
        streamid_t id; /* bus master id for iommu effects */
};
```

where:

- **pa [mandatory] -** corresponds to the base physical address of a device.
- **va [mandatory]** - corresponds to the base guest virtual address of a device.
- **size [mandatory]** - corresponds to the size of a device memory region.

Note: It is mandatory for *base* and *size* to align with the smallest page size of the architecture. For MMU systems, this typically aligns to 4K, while for MPU systems, it aligns to 64 bytes. Please note that for MPU systems, the VA must match the PA.

- **interrupt_num [optional] -** corresponds to the number of assigned interrupts. *interrupt_num* equals 0. By default, *interrupt_num* equals 0.
- **interrupts [mandatory if interrupt_num > 0] -** defines a list of interrupt IDs generated by the device - `(irqid_t[]) {irq_1, ..., irq_n}`.
- **id [optional]** - corresponds to the bus master ID for IOMMU effects.

Warning: Specifying a number of interrupts in the *interrupts* buffer that differs from the *interrupt_num* may result in undefined behavior.

Architectural-Specific Configurations:

CROSSCON Hypervisor can be configured for different architectures. Among them are the ARM and RISC-V architectures. For different architectures there are different configurations. Both are configured within *arch* parameter in *vm_platform*. For configuring *arch*, *arch_vm_platform* struct is used.

- **arch -** pointer to *arch_vm_platform* structure that allows defining architecture-dependent configurations.

For Arm architecture, the arch_platform structure is defined as follows:

```
struct arch_platform {
        struct gic_dscrp {
            paddr_t gicc_addr;
            paddr_t gich_addr;
            paddr_t gicv_addr;
            paddr_t gicd_addr;
            paddr_t gicr_addr;
            irqid_t maintenance_id;
        } gic;
        struct smmu_dscrp {
            paddr_t base;
            streamid_t global_mask;
        } smmu;
        struct clusters {
            size_t num;
            size_t* core_num;
        } clusters;
};
```

The GIC interrupt controller struct *gic_dscrp* description is as follows:

- **gic.gicc_addr [mandatory for GICv2 platforms] -** base address for the GIC's CPU Interface.
- **gic.gich_addr [mandatory for GICv2 platforms] -** base address for the GIC's Virtual Interface Control Registers.
- **gic.gicv_addr [mandatory for GICv2 platforms]** - base address for the GIC's Virtual CPU Interface

- **gic.gicd_addr [mandatory] -** base address for the GIC's Distributor.
- **gic.gicr_addr [mandatory for GICv3/4 platforms]** - base address for the GIC's Redistributor.
- **gic.maintenance_id [mandatory]** - The interrupt ID for the GIC's maintenance interrupt.

For the SMMU struct *smmu_dscrp*:

- **smmu.base [mandatory]** - is the base address for the SMMU.
- **smmu.global_mask [optional, valid only for SMMUv2**] - a mask to be applied to all SMMUv2's Stream Match Registers.

Finally, when CPUs are organized in clusters, in the ARM architecture their IDs are assigned using a hierarchical scheme.

For RISC-V architecture, the *arch_platform* structure is defined as follows:

```
struct arch_platform {
      union irqc_dscrp {
            struct {
                  paddr_t base;
            } plic;

            struct {
                  struct {
                  paddr_t base;
                  } aplic;
            } aia;
      } irqc;
      struct {
              paddr_t base;
              irqid_t fq_irq_id;
      } iommu;
      struct {
            paddr_t base;
      } aclint_sswi;
};
```

In case the available interrupt controller is the legacy PLIC:

- **irqc.plic.base [mandatory if PLIC is available]** - is the base address for the PLIC.

In case the available interrupt controller is an AIA containing an APLIC:

- **irqc.aia.aplic.base [mandatory if APLIC is available]** - is the base address for the APLIC.

When an IOMMU is available:

- **iommu.base [mandatory if IOMMU is available] -** is the base address for the IOMMU.
- **iommu.fq_irq_id [mandatory if IOMMU is available]** - the Fault Queue interrupt ID (the current implementation assumes this is a wired interrupt).

Note: When mapping MMIO regions for guests, the memory regions associated with the interrupt controller must be excluded. In CROSSCON Hypervisor, interrupt controllers are target of MMIO trap and emulate. Mapping these regions can lead to conflicts or undefined behaviour, as they are typically managed by the CROSSCON Hypervisor through trap-and-emulate mechanisms. For instance, if a large MMIO range includes the GIC, the range should be split to create a "hole" for the GIC. This ensures that GIC memory regions (or their equivalents in other architectures, such as RISC-V) are not directly mapped into the guest's virtual address space.

Configuration Example:

The following example delineates an example configuration (referred to as "*config*") source file that is used to statically partition two VMs on an Armv8-A platform. This example corresponds to the instantiation option for APU devices, and its structure is illustrated in Figure 10.

```
#include <config.h>
/** * Declare VM images using the VM_IMAGE macro, passing an identifier and
the * path for the image. */
VM_IMAGE(vm1_image, "/path/to/vm1/binary.bin");
VM_IMAGE(vm2_image, "/path/to/vm2/binary.bin");

struct vm_config vm1 {
      .image = VM_IMAGE_BUILTIN(vm1, 0x80000000),
      .entry = 0x80000000,
      .cpu_affinity = 0x3,
      .colors = 0x55555555,
      .platform = {
            .cpu_num = 2,
            .region_num = 1,
            .regions = (struct vm_mem_region[]) {
            { .base = 0x80000000, .size = 0x100000 } },
            .dev_num = 1,
            .devs = (struct vm_dev_region[]) { {
            /* UART0 */
                  .pa = 0x1c090000,
                  .va = 0x1c090000,
                  .size = 0x10000,
                  .interrupt_num = 1,
                  .interrupts = (irqid_t[]) {38} } },
           .ipc_num = 1,
           .ipcs = (struct ipc[]) { {
                  .base = 0x80100000,
                  .size = 0x1000,
                  .shmem_id = 0,
                  .interrupt_num = 1,
                  .interrupts = (irqid_t[]) {42} } },
          .arch = {
                 .gic = {
                      .gicc_addr = 0x2C000000,
                      .gicd_addr = 0x2F000000 }
          }
      },
};

struct vm_config vm2 {
      .image = VM_IMAGE_BUILTIN(vm1, 0x00000000),
      .entry = 0x0000000,
      .cpu_affinity = 0xC,
      .colors = 0xAAAAAAAA,
      .platform = {
            .cpu_num = 2,
            .region_num = 2,
            .regions = (struct vm_mem_region[]) {
            {
                .base = 0x00000000,
                .size = 0x80000000 },
            {
                .base = 0x100000000,
                .size = 0x40000000 }
            },
            .dev_num = 2,
            .devs = (struct vm_dev_region[]) {
            { /* UART1 */
                .pa = 0x1C0B0000,
                .va = 0x1c090000,
                .size = 0x10000,
                .interrupt_num = 1,
```

```
                .interrupts = (irqid_t[]) {39} },
            { /* Timer interrupt */
            .interrupt_num = 1,
            .interrupts = (irqid_t[]) {27} } },
            .ipc_num = 1,
            .ipcs = (struct ipc[]) { {
            .base = 0x90000000,
            .size = 0x1000,
            .shmem_id = 0,
            .interrupt_num = 1,
            .interrupts = (irqid_t[]) {112} } },
            .arch = {
                .gic = {
                        .gicc_addr = 0x2C000000,
                        .gicd_addr = 0x2F000000
                }
        }
        },
};

/** * The configuration itself is a struct config that MUST be named config.
*/
struct config config = {
/** * This defines an array of shared memory objects that may be associated
* with inter-partition communication objects in the VM platform definition *
below using the shared memory object ID, i.e., its index in the list. */
.shmemlist_size = 1,
.shmemlist = (struct shmem[]) { [0] = {.size = 0x1000,} }, /** * This
configuration has 2 VMs. */
.vmlist_size = 2, .vmlist = { &vm1, &vm2 },
};
```

VM1 image is a file found in the directory */path/to/vm1/binary.bin* and the VM2 image file is */path/to/vm2/binary.bin*. The images are embedded in the binary as a result of using the *VM_IMAGE* macro. Next, the struct *config* is defined. One of the shared memory regions is configured for the system, which will be used as a communication channel between the two VMs. The shared memory region size is *0x1000 (4096)* bytes. Following macro invocation, the next section defines the configurations for the two VMs.

VM1 image base address in VM1's virtual address space is *0x80000000*. The macro *VM_IMAGE_BUILTIN* is used to easily configure VM1's image attributes. VM1 will begin execution at address *0x80000000*. Physical CPUs 0 and 1 are assigned to VM1. VM1 will use half of the available cache colors after the VM1 platform is defined. The platform will feature two vCPUS. There will be a single memory region starting from guest address *0x80000000* with size *0x100000 (1MB*). There will be a single device region. The device at address *0x1C090000* will be presented to VM1 at address *0x1C090000* (the same address as the physical device), and this device region is *0x10000*. The device features a single interrupt, *interrupt 38*.

A single IPC channel is configured. The shared memory will be presented to VM1 at address *0x80100000*. The shared memory region is *0x1000* bytes in size. The shared memory ID is *0* (the index of the shared memory in the array). The IPC channel features a single interrupt, *interrupt 42*. For architecture-specific features, the addresses of the VM will be able to access the GIC controller and GIC distributor (GICv2 interrupt controller) at *0x2C000000* and *0x2F000000*, respectively.

VM2 image base address in VM2's virtual address space is *0x00000000*. The macro *VM_IMAGE_BUILTIN* is used to specify VM2's image information. VM2 will begin execution at address *0x00000000*. Physical CPUs 2 and 3 are assigned to VM2. Because the preferred physical CPUs for each VM are mutually exclusive, the physical CPUs will be assigned to each VM according to their preferences. VM2 will use the other half of the available cache colors after the VM2 platform is defined. The platform will feature *two vCPUS*. There will be *two memory regions*, one starting from

address *0x00000000* with size *0x80000000 (2GB)* and the second starting at address *0x100000000* with size *0x40000000 (1GB).* There will be a single device region. The device at address *0x1C0B0000* will be presented to VM1 at address *0x1C090000* (NOT the same address as the physical device), and this device region is *0x10000*. The device features a single interrupt, *interrupt 39*. A second device is configured but without a memory region to allow VM2 to access the architectural timer *interrupt 27*.

A single IPC channel is configured. The shared memory will be presented to VM2 at address *0x90000000*, with size *0x1000* bytes. The shared memory ID is 0 (the index of the shared memory in the array). The IPC channel features a single interrupt, *interrupt 112*. For architecture-specific features, the addresses of the VM will be able to accessthe GIC controller and GIC distributor (GICv2 interrupt controller) *at 0x2C000000 and 0x2F000000*, respectively.

Note: When configuring the CROSSCON Hypervisor for a different architecture, including low-level ones such as Armv8-M, the configuration file remains unchanged. To work with Armv8-M, please use the *"next"* branch of the CROSSCON-Hypervisor repository. This branch is intended to become the main integration point for all architectures, platforms, and CROSSCON Hyp. features within a single repository. Currently, LPCXpresso55S69 Development Board is the only MCU platform supported by CROSSCON Hypervisor.

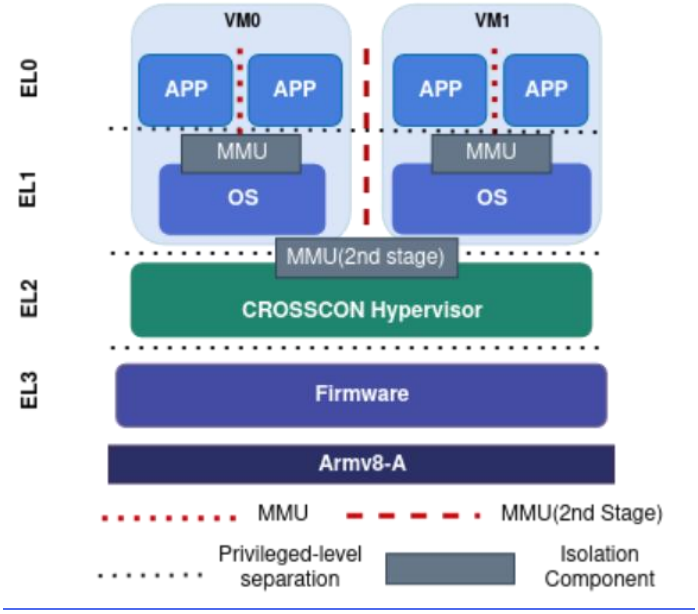


Figure 10: Configuration example of 2 VMs running on top of CROSSCON Hypervisor

3.1.2 Per-VM TEE Configuration Example

This configuration file defines a secure virtualization environment that creates two VMs: one of them is running Linux and the other is running OP-TEE OS, a TEE. The configuration follows TrustZone execution model with majority of system resources assigned to Linux, while OP-TEE OS runs in the environment isolated from Linux, enforcing strong isolation between them. The configuration establishes memory regions, device mappings, and shared resources to provide controlled communication between the two VMs.

```
#include <config.h>
/** * Declare VM images using the VM_IMAGE macro, passing an identifier and
the * path for the image. */
VM_IMAGE(linux_image, "/path/to/vm1/linux.bin");

struct vm_config linux {
	.image = VM_IMAGE_BUILTIN(vm1, 0x40200000),
	.entry = 0x40200000,
	.platform = {
		.cpu_num = 1,
```

```
            .region_num = 1,
            .regions = (struct vm_mem_region[]) {
            { .base = 0x40000000, .size = 0x30000000 } },
            .dev_num = 2,
            .devs = (struct vm_dev_region[]) { {
                  .pa = 0x9000000,
                  .va = 0x9000000,
                  .size = 0x10000,
                  .interrupt_num = 1,
                  .interrupts = (irqid_t[]) {33} },
                  {
                  .interrupt_num = 1,
                  .interrupts = (irqid_t[]) {27} },
                  },
           .ipc_num = 1,
           .ipcs = (struct ipc[]) { {
                  .base = 0x70000000,
                  .size = 0x200000,
                  .shmem_id = 0},
          .arch = {
                 .gic = {
                      .gicc_addr = 0x8010000,
                      .gicd_addr = 0x8000000,
                      .gicd_addr = 0x80A0000 }
          }
      },
};

VM_IMAGE(optee_os_image, "/path/to/tee/tee.bin");

struct vm_config optee_os {
      .image = VM_IMAGE_BUILTIN(optee_os_image, 0x10100000),
      .entry = 0x10100000,
      .cpu_affinity = 0xf,
      .type = CROSSCON_VM_SDTZ,
      .children_num = 1,
      .children = (struct vm_config*[]){ &linux },
      .platform = {
            .cpu_num = 1,
            .region_num = 1,
            .regions = (struct mem_region[]) {
            {
                .base = 0x10100000,
                .size = 0x00f00000 },
            .dev_num = 2,
            .devs = (struct vm_dev_region[]) {
            { /* PL011 */
                .pa = 0x9040000,
                .va = 0x9040000,
                .size = 0x10000,
            { /* Timer interrupt */
            .interrupt_num = 1,
            .interrupts = (irqid_t[]) {27,40} } },
            .ipc_num = 1,
            .ipcs = (struct ipc[]) { {
            .base = 0x70000000,
            .size = 0x20000,
            .shmem_id = 0,
            .arch = {
                .gic = {
                      .gicc_addr = 0x8010000,
                      .gicd_addr = 0x8000000,
                      .gicr_addr = 0x80A0000
```

```
                }
        }
        },
};

/** * The configuration itself is a struct config that MUST be named config.
*/
struct config config = {
/** * This defines an array of shared memory objects that may be associated
* with inter-partition communication objects in the VM platform definition *
below using the shared memory object ID, i.e., its index in the list. */
.shmemlist_size = 1,
.shmemlist = (struct shmem[]) { [0] = {.size = 0x20000,} }, /** * This
configuration has 2 VMs. */
.vmlist_size = 1, .vmlist = { &optee_os },
};
```

The system is designed to include two VM images: one for Linux, which is loaded from `<path-to-linux>/linux.bin`, and another for OP-TEE OS, sourced from `<path-to-tee>/tee.bin`. Both images are embedded in the hypervisor using the *VM_IMAGE* macro. The configuration also specifies a single shared memory region of *2MB (0x00200000),* for inter-VM communication. This shared memory, indexed as shmem_id = 0, is mapped to both VMs at *0x70000000*, allowing them to exchange data securely.

The Linux VM is assigned a single virtual CPU and a memory region of *768MB (0x30000000),* starting from address *0x40000000*. The Linux image loads and begins execution at *0x40200000*. In addition to memory, Linux has access to two devices:

1. a PL011 UART device mapped at *0x9000000* (with the same virtual address)
2. an interrupt-based device (the architectural timer) which requires guest access to the timer interrupt (ID 27).

The Linux VM is also assigned interrupt ID 33, likely associated with UART communication. Furthermore, Linux shares the GIC (Generic Interrupt Controller) configuration with OP-TEE OS, granting it access to manage system-wide interrupts.

The OP-TEE OS VM is also allocated one vCPU but is assigned a smaller *15MB (0x00F00000)* memory region starting from *0x10100000*. OP-TEE OS loads and starts execution at the same address. Unlike Linux, which operates as an independent VM, OP-TEE OS is configured as the primary (secure) VM, and Linux runs as its child VM. This hierarchy enforces a TrustZone-like security model where OP-TEE manages secure operations, while Linux operates in the non-secure domain. OP-TEE is granted access to the same PL011 UART device, but at a different virtual address (*0x9040000*). Additionally, OP-TEE OS is responsible for handling two interrupt sources: interrupt ID 27 (shared with Linux) and interrupt ID 40.

A key feature of this configuration is inter-VM communication via shared memory. The 2MB memory region *at 0x70000000* is mapped into both VMs' address spaces. This setup allows OP-TEE OS to act as a secure service provider, enabling trusted applications to interact with Linux while enforcing strict isolation. OP-TEE implements sensitive operations and controls access to security-critical resources.

In terms of interrupt handling, both VMs share the same GIC configuration, ensuring that interrupts are appropriately routed between the secure and non-secure execution environments. The GICC is located at *0x8010000*, the GICD at *0x8000000*, and the GICR at *0x80A0000*. This shared configuration allows efficient Linux operation while OP-TEE remains in control of critical security-sensitive operations.

In this configuration, the key distinction between the Linux VM and the OP-TEE VM lies in the `.type` parameter, which is set to *CROSSCON_VM_SDTZ* for the OP-TEE VM. Additionally, the Linux VM operates as a child of the OP-TEE VM through the VM stacking mechanism. This hierarchy is defined

using the `.children_num` and `.children` parameters. It is important to note that the configuration structure includes only one VM in the *vmlist, i.e.,* the OP-TEE VM. This guarantees that OP-TEE is loaded first and that the scheduler transitions to the Linux VM once the OP-TEE VM has booted. This setup effectively replicates the behavior of a real TrustZone environment.

3.1.3 Dynamic VM Creation

For configuring dynamic VMs, there are no feature-specific changes to the config.c file. In config.c file only the configuration of the parent VM is needed, i.e., the configuration of the VM that will invoke the dynamic VM during runtime. To demonstrate such capability, an example that combines the previously introduced per-VM TEE feature with the dynamic VM capability is provided [39]. In this setup, a Linux VM can launch an SGX TEE VM at runtime. This not only demonstrates how to configure dynamic VM features but also shows how to enable a TEE originally not conceived for such a platform and architecture to run on top of it, thereby highlighting TEE heterogeneity and interoperability.

For preparing such scenario, the configuration file will need at least one VM to operate as a parent and to invoke other VMs. This VM should be configured following the standard configuration in config.c file.

3.1.4 How to compile CROSSCON Hypervisor

This section explains how to compile the CROSSCON Hypervisor. However, before starting to explain the the compilation process, make sure that machine you are using for compiling has the following prerequisites listed in Table 2.

Table 2: List of CROSSCON Hypervisor compilation prerequisites and components

Tool	Version
arm-none-eabi-gcc	11.3.1
aarch64-none-elf-gcc	11.1.1
riscv64-unknown-elf-gcc	10.2.0
arm-none-eabi-gcc	13.3.1
make	4.2.1
dtc	1.5.0
gcc	9.3.0
mkimage	20.10
cmake	3.20.0
ninja	1.10.1

After meeting the prerequisites and starting the CROSSCON Hypervisor compilation, ensure that the following environment variables are defined:

- **CONFIG_REPO**: Specifies the directory containing the file.
- **PLATFORM**: Defines the target platform. The available platforms are listed in Table 2.
- **CONFIG**: Indicates the name of the configuration file.
- **OPTIMIZATIONS (optional)**: Sets the optimization level to be applied during hypervisor compilation.
- **SDEES (optional)**: Specifies the name of the Software-Defined Environment to be added. This usually refers to the TEE model that should be included to handle specific calls within the environment.
- **CROSS_COMPILE:** Specified the compiler to be used.

Example of CROSSCON Hypervisor compilation for ARM on QEMU:

According to Table 1, compiling the CROSSCON Hypervisor on QEMU for the ARM architecture uses the `qemu-aarch64-virt` platform. Consider the case of building a simple demo consisting of a Linux VM linked to an OP-TEE VM. Assume that all VM-side components (OP-TEE, TAs, CAs, Linux filesystem,

Linux kernel, device trees, etc.) have already been compiled. For further details, refer to the CROSSCON-Hypervisor-and-Isolation-Demos repository on the CROSSCON GitHub page.

In such a demo scenario, begin by cleaning the tree before compiling the hypervisor. This is done by invoking the clean command from within the CROSSCON-Hypervisor directory as follows:

```
make -C CROSSCON-Hypervisor/ \
PLATFORM=qemu-aarch64-virt \
CONFIG_BUILTIN=y  \
CONFIG_REPO=$CONFIG_REPO \
CONFIG=qemu-virt-aarch64-single-vTEE \
OPTIMIZATIONS=0 \
SDEES="sdSGX sdTZ" \
CROSS_COMPILE=aarch64-none-elf- \
Clean
```

Next, for compiling it, invoke the make command:

```
make -C CROSSCON-Hypervisor/ \
PLATFORM=qemu-aarch64-virt \
CONFIG_BUILTIN=y \
CONFIG_REPO=$CONFIG_REPO \
CONFIG=qemu-virt-aarch64-single-vTEE \
OPTIMIZATIONS=0 \
SDEES="sdTZ" \
CROSS_COMPILE=aarch64-none-elf- \
-jnproc
```

Note that the TrustZone model is selected by setting the 'SEES' environment variable. In the end, expect to obtain a "crossonhyp.bin" file. And that's it, the hypervisor is compiled and ready for being attached to QEMU as bl33 file.

Example of CROSSCON Hypervisor compilation for RISC-V on QEMU:

Just as in ARM, for setting up such simple demo scenario, start by ensuring everything is clean before compiling Hypervisor, so invoke the command clean into the CROSSCON-Hypervisor folder using the following format:

```
make -C CROSSCON-Hypervisor/ \
PLATFORM=qemu-riscv64-virt \
CONFIG_BUILTIN=y \
CONFIG_REPO=$CONFIG_REPO \
CONFIG=qemu-virt-riscv64-single-vTEE \
OPTIMIZATIONS=0 \
SDEES="sdSGX sdTZ" \
CROSS_COMPILE=riscv64-unknown-elf- \
clean
```

Next, for compiling it, invoke the make command:

```
make -C CROSSCON-Hypervisor/
PLATFORM=qemu-riscv64-virt
CONFIG_BUILTIN=y
CONFIG_REPO=$CONFIG_REPO
CONFIG=qemu-virt-riscv64-single-vTEE
OPTIMIZATIONS=0
```

```
SDEES="sdTZ"
CROSS_COMPILE=riscv64-unknown-elf-
-jnproc
```

Similar to ARM, where pre-compiled bootloader stages are needed, in RISC-V there is the OpenSBI. In this case is advised to compile it by adding as part of firmware payload. As such, to complete this process compiling the OpenSBI is recommend as follows:

```
rm -rf opensbi/build/ make -C opensbi
PLATFORM=generic
FW_PAYLOAD=y
FW_PAYLOAD_FDT_ADDR=0x80100000
FW_PAYLOAD_PATH=<path to CROSSCON Hyp>/bin/qemu-riscv64-virt/builtin-configs/qemu-virt-riscv64-single-vTEE/crossconhyp.bin
CROSS_COMPILE=riscv64-unknown-elf-
-j7
```

After this step CROSSCON Hypervisor is ready to run on QEMU.

3.1.5 How to test CROSSCON Hypervisor with different setups

To optimize and accelerate the process of compilation for different setups (or *Demos*) a script on a README file on CROSSCON GitHub page [40] is provided to guide users through the required steps for different architectures and demos. Table 3 is the list of demos which are available in the repository.

Table 3: List of demos available for testing

Demo Name	Description	Associated Feature	Supported Arch
Simple Demo	This demo instantiates a Linux VM and an OP-TEE VM. Use "*run-demo-vtee.sh*" script to compile.	Per-VM TEE	RISC-V, Arm
Demo 1	This demo instantiates two Linux VMs each with an OP-TEE VM	Per-VM TEE	RISC-V, Arm
Demo 2	This demo instantiates a Linux VM and two OP-TEE VMs.	Per-VM TEE	RISC-V, Arm
Demo 3	This demo showcases a Linux VM and two OP-TEE VMs, where the OP-TEEs are vulnerable, and might try to compromise the Linux VM or the other OP-TEE instance.	Per-VM TEE	RISC-V, Arm
Demo 4	This demo showcases a Linux VM instantiating an SGX-like enclave on aarch64.	Dynamic VM	RISC-V, Arm
Demo 5	This demo showcases support for multiple programming models simultaneously (Enclave SGX + TZ model), with each TEE being configured with specific access to memory and IO.	Per-VM TEE	RISC-V, Arm
Demo 6	This demo instantiates a Linux VM a FreeRTOS VM, and Baremetal environment VM, and establishes IPC communication between the Linux and FreeRTOS VMs.	Multiple VMs	Arm

These demos can be replicated by running scripts stated into GitHub repository.

3.2 CROSSCON Bare-Metal TEE

On devices with severely constrained resources, running the CROSSCON Hypervisor may not be feasible. For this reason, two different bare-metal TEEs were developed: the BareTEE-noMPU and the BareTEE-MPU. The former can be used on devices lacking an MPU, whereas the latter can leverage the

presence of this component. Both TEEs strive to supply a reduced yet comprehensive set of security features to satisfy the three basic security requirements considered in the CROSSCON project: memory isolation, privilege separation and cross-domain intercommunication.

3.2.1 BareTEE-nonMPU

The nonMPU-BareMetal TEE is a bare-metal Trusted Execution Environment (TEE) for low-end embedded systems with no support for MMU/MPU. As a TEE, nonMPU-BareMetal-TEE creates a new execution environment where trusted applications can run without interference from other (untrusted) applications running on the device. We refer to these two execution environments as the TEE and the Untrusted Environment (UE). These two environments coexist on the device and are separated by the nonMPU-BareMetal-TEE's software-based primitives, which ensure that the TEE is completely isolated from the UE. Being a software-based TEE, nonMPU-BareMetal-TEE relies on a Trusted Computing Module (TCM) to enforce the isolation between the two environments. The TCM is a piece of software acting as Root of Trust (RoT) that is run in the TEE, alongside other Trusted Applications (TAs). These TAs can be deployed by security developers to implement the security services that are needed for the correct functioning of the device. These services can then be leveraged by any software running on the UE using a set of registered APIs. For more information, please refer to the GitHub repository [16].

3.2.1.1 Running nonMPU-BareMetal-TEE

Prerequisites

- **linux**: build-essential python3 pip3
- **pip3**: pyserial [matplotlib] [pandas]
- MSP430F5529 board with USB cable (currently using MSP-EXP430F5529LP)
- Code Composer Studio with MSP430 libraries official link [17]
- (optional) MSP430 debugger (e.g. MSP430F5529Launchpad version) board link [18]
- (optional) IAR workbench msp430 Kickstart edition official link [19]
- (optional) Java for the execution of powerConsumption measurement scripts.

Configuration

Before deploying the baremetal-TEE, its TCM must be configured. This can be done through the `TCM/core/src/core.h` header file which contains several macros that define the behaviour of the TEE. Notably, the FLASHADOW_ENABLE macro allows the TEE to be shipped with the control flow integrity module for the protection of the backward edges, as described in Section 3.4.4

Loading nonMPU-BareMetal-TEE TCM (with the default untrusted application)

To secure the microcontroller, the TCM (i.e. the root of trust for our architecture) needs to be loaded before any other program. The folder TCM/ contains all the required files for the compilation of nonMPU-BareMetal-TEE TCM. Although it is possible to also load an untrusted application right away, to be loaded alongside the TCM, currently the toolchain does not fully support it. The first initialisation should be performed with the default application. The following steps must be performed:

- (optional) Copy all the required untrusted application's source files inside the `TCM/app/src/` folder. The default application only calls the `receiveUpdate()` function from the TCMHooks to initiate a secure update. If such function is not required, or the `secureupdate` module is not loaded, the default application should be changed, and the appropriate logic should be added. If no application is loadedspecified, the default one will be loaded. When no application is loaded, a default one is started, which triggers a secure-update session and causes the system to wait for a deployable image via UART.
- (optional) Modify the `TCM/Makefile` file to update the binaries paths. By default, the toolchain will use the self-contained compiler in this repository.

- Generate the secure deployable image of the TCM using the make command from inside the TCM/ folder. This will generate a `deployable.out` binary (if curious, you can check it with readelf to see how it is structured).
- Load the `TCM/deployable.out` image on the device, e.g. using Code Composer Studio 'load' function. If everything was loaded correctly, you should see the following:
  - The red LED blinks 10 times
  - The red LED turns on for verification (a few seconds)
  - The green LED turns on →ready for incoming updates
- You can also reset the board to start the verification from the beginning. If you see any other combination of LED check section "LED Signals" for debugging.

3.2.1.2 Applications updates / Remote deployments

Compiling the update/application

To securely deploy an application on the nonMPU-BareMetal-TEE, a properly crafted update must be compiled. To automate this process, nonMPU-BareMetal-TEE offers the folder `UpdateApplication/` which contains the `Makefile` that must be used. Precisely, the following steps are required to compile the untrusted update:

- Copy all of the source files of the application ('.c' and '.s') to the `UpdateApplication/src/` folder. If the folder does not exist it should be created. These are the files that will be compiled.
- Execute the `make` command. This will generate the first executable deployable.out.
- Execute the `make libraries` command to generate some helper files for the instrumentation of the standard library code (check section "Library instrumentation" for more details).
- Execute the `make clean && make` command to generate the final executable `deployable.out` containing the instrumented code for both the application and the standard libraries used by it. NB: this executable is not an ELF file because it contains extra metadata added for nonMPU-BareMetal-TEE. To check the original ELF file you can inspect `appWithoutMetadata.out`.
- Proceed with the deployment

Deploying untrusted update

An application update can only be performed if the MCU is in the receiving update state (see Section LED signals). In order to enter this state, the running program must call the `callReceiveUpdate()` function from the `TCMhook.h` header file (as is the case when loading the TCM with the default application). When the MCU is ready to receive the update execute the `/toolchain/loadProgram.py` python script, passing as arguments the final application binary (e.g. `deployable.out`) and the serial port name (e.g. `/dev/ttsyAC2` on Linux or `COM4` on Windows).
NB: The serial port might change across executions or systems, please check the serial ports available on your system (especially those that become available as soon as the MCU board is plugged through the USB port).

If the serial port is correct, the script will uplaod applications in chunks. Each succesfull chunk sent will be followed by acknowledgement. As soon as the upload is finished, the script will output a “File sent” to the terminal and the TCM will begin its deployment operations. As soon as the TCM receives the image it will verify it. If the image is verified with success and the binary does not contain unsafe code (that is, code that lacks the instrumentation inserted by the nonMPU-BareMetal-TEE compilation toolchain), then it should be launched by the TCM. Otherwise, the TCM will restart the update process and wait for another valid image.

Be aware that the code in this repository is compatible with the MSP430 family of microcontrollers. However, the code, along with some of the toolchain components, needs to be updated to work with anything different than an MSP430f5529 MCU.

LED signals

Table 4 shows the current nonMPU-BareMetal-TEE' usage of the various LEDs.

Table 4: LED meaning

Green LED	Red LED	Description
blink*10 → ON	OFF	The TCM is waiting for an income update
blink*5 → ON	blink*5 → ON	Alternated blinking signals a successful verification of the client application
ON	ON	Reserved
OFF	OFF	Application started
OFF	ON	Verification of code in progress
OFF	blink*10 → ON	MCU has been reset, starting secure boot

(optional) Loading the custom BSL

Some functionalities of nonMPU-BareMetal-TEE are only available if the BSL section of the FLASH is modified. Specifically, the aim is to reduce some of its codebase in order to make room for the sensitive data. Moreover, its interface (z-area) needs to be modified to allow the TCM to interact with it without restrictions. To modify the bootloader, IAR workbench msp430 needs to be used, which is a closed-source IDE. Fortunately, its free kickstart edition can be used to modify part of the BSL to support our TCM safe Storage segment. The following steps must be be permed:

- Make a copy of the project `BSL/IAR_BSL_Source/IAR_v5_MSP430F552x_USB/` folder.
- Open the local copy with IAR workbench.
- Copy the files in the `BSL/modifiedFiles/` folder to the local copy of the project (these should be placed in the various subdirectories).
- Compile the project and upload it to the MCU. After compiling and uploading the BSL, the MSP430 board will be equipped with the nonMPU-BareMetal-TEE Safe Storage segment.

(optional) Loading the Key

Before loading the TCM, a cryptographic key needs to be loaded on the device so that it can be used by the various Trusted Applications (TAs). Note that this step is optional: a cryptographic key is only needed by some specific TAs, e.g. Remote Attestation. Still, the TCM offers some security guarantees which are independent of the nonMPU-BareMetal-TEE Safe Storage. To load the key, use the BSL/loadKey.c application by flashing it on the device, e.g. via CCS Studio. This application will take care of unlocking the BSL and storing a pre-defined key (which can be modified in the source file) on it, locking the BSL afterwards.

Library Instrumentation

Applications using common libc functions, such as `memset`, `memcpy` and similar, need a few extra steps to work correctly. Since our untrusted toolchain (i.e. the preprocessor and modifier scripts) only works with source files, it cannot be used to instrument the statically linked binaries. Such binary code derives from statically linked libraries, e.g. the libc library, for instance whenever functions such as malloc are required. The binary code is therefore appended to the rest of the instrumented application binary whenever compiling with the GCC linker. The instrumentation of the binary code is automatically performed by our toolchain (although it might not be optimal in some cases).

To fully optimise the library code used by an application, the following steps should be performed, some of which are manual. First, our modified linker script assigns the explicit application binary code (i.e. only the binary code derived from the files in `UpdateApplication/src/`, without the libc libraries) to the `.appText` code section. Code sections allow the code to be organised in the binary, allowing the linker to place them in the different parts of the memory. Although our custom

`.appText` section is placed alongside the rest of the application code, which by default would be assigned to the `.text` section, this separation allows us to easily spot the statically linked code. More precisely, while `.appText` will contain our source code, `.text` and `.lower.text` will only contain the code of the libraries. The library code can therefore be retrieved by extracting only the `.text` and the `.lower.text` sections from the assembly dump of the binary file. These sections will indeed only contain the code not derived from our source files, i.e. the statically linked binary code.

To facilitate this process, an auxiliary script auxGccParser that extracts the library code from the deployable and performs some optimisations is proposed. This will create the new file `UpdateApplication/src/helper.s` containing the uninstrumented code.

Optimisations

On top of creating a new assembly file, the auxGccParser script will output to console some possible optimisations, e.g. "Possible optimisation with `calla r7` found". These are dynamic calls found in the binary code that could be replaced with some static calls, thus improving the performance of the code. With the current toolchain, such optimisations are mere suggestions that require manual verification and inspection. Beware that some of them could not be valid! To figure out which are indeed valid and apply them, a manual inspection is required. Specifically, the following steps should/could be performed. Examining the newly created file `UpdateApplication/src/helper.s`, for each optimisation mentioned by the output of the above script, do the following:

- Find the dynamic call in question, e.g. `calla r7`, by searching the file.
- Look for the instruction that assign a value to the concerning register (e.g. `mova #_sbrk_r, r7`, which loads the address of `_sbrk_r` into `r7`). NB: this instruction must be looked for among the assembly instructions preceding the dynamic call.
- Replace the dynamic call (e.g. `calla r7`) with a call to the label used in the definition (e.g. `calla #_sbrk_r`).
- Remove the assignation (e.g. `mova #_sbrk_r, r7`) since no longer needed.

Note that it might not be trivial deciding whether the optimisation is valid or not: some might seem valid but might instead break the application. As a rule of thumb, the optimisation is valid if there is no other register assignment (e.g. for `r7`) between the dynamic call and the spotted assignment. This means that the call will indeed be only to that static address (e.g. `_sbrk_r`). NB: be aware of the jumps that might make the analysis of the code non-linear.

Debug and measurements

To debug nonMPU-BareMetal-TEE or to run measurements, it might be useful to run nonMPU-BareMetal-TEE in Code Composer Studio (CCS). To facilitate this, the archive `debugCCSProject.zip` contains an exported CCS project. This can be imported in CCS and used to debug the various components of nonMPU-BareMetal-TEE. Measurements can be computed using the internal clocks as well in debug mode, or with a logical analyser for a more accurate evaluation.

Debugging the project

The repository comes with two Code Composer Studio projects, zipped in two separate files: `defaultCCSProject.zip` and `flashadowCCSProject.zip`. Both can be unzipped and imported within CCS to allow the debugging. The two versions are used, respectively, to debug the TEE without CFI and the TEE with CFI.

Interacting with the TEE

NonMPU-BareMetal-TEE comes with two Trusted Applications (TAs): Secure Remote Update and Remote Attestation. However, nonMPU-BareMetal-TEE supports the creation of custom TAs to boost the security guarantee of the system. Each TA can implement one or more security services, which are exposed to the Untrusted Application via a set of APIs (e.g. `void callReceiveUpdate()`). Furthermore, a TA can also call another TA service by using the same endpoints.

A full list of the exposed APIs can be found in the GitHub repository [20].

3.2.2 BareTEE-MPU

The BareMetal-TEE with MPU is a solution for memory isolation and supervised execution aiming at increasing the security of embedded devices in a way that is mostly transparent to application developers. The architecture is fully compliant with the Global Platform TEE Client API specification and with a subset of the TEE Core API. The BareMetal TEE with MPU supports the deployment of up to two custom Trusted Applications (TAs) compliant with the Global Platform TEE Core API.

3.2.2.1 Configuration

The source files `microvisor_config.h` and `microvisor_config.c`, contained respectively in the folders `OS/Microvisor/Inc` and `OS/Microvisor/Src`, are designed to be modified to customize the microvisor configuration. These files contain the following:

- Network parameters, such as Wi-Fi SSID and password necessary for connecting the device to the Internet.
- Connection parameters, such as the IP and port of the remote Update Server queried to check and download updates during boot.
- Symmetric encryption key, used to encrypt and authenticate the communication with the Update Server (the same value must be set for both the Microvisor and the Update Server).

3.2.2.2 Compilation

Prerequisites

In order to compile both mbedtls library and the Baremetal TEE, the GNU ARM Embedded Toolchain should be installed on the system. It is normally included in the STM32CubeIDE (which will be used later on) with a path similar to this one (depending on the installation folder):

```
/home/user/st/stm32cubeide_1.15.0/plugins/com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.12.3.rel1.linux64_1.0.100.202403111256/tools/bin
```

Otherwise, the toolchain can be installed on the system as a standalone package.

Compilation of MBEDTLS

After cloning the repository, and before compiling the Baremetal TEE for the first time, it is required to follow this additional step due to the usage of parts of the `mbedtls` cryptography library inside the TEE codebase.

The required code of the library is stored in the `OS/Middlewares/mbedtls` folder, alongside a custom `Makefile` to obtain the library files (`libmbedcrypto.a`, `libmbedtls.a` and `libmbedtls.a`) needed. Only some parts of the library are used (see the `mbedtls/include/mbedtls/mbedtls_config.h` configuration file) and specific parameters are needed for the compilation to fit on and work with the target board.

To start the compilation process, use the following command while located in the library folder of mbedTLS (e.g. `user@pc:~/Code/MCU-fortifier/OS/Middlewares/mbedtls/library$`):

```
make CC=arm-none-eabi-gcc AR=arm-none-eabi-ar LD=arm-none-eabi-ld CFLAGS="-mcpu=cortex-m3 -mthumb -DUSE_HAL_DRIVER -DSTM32L475xx -Og -Wall -fdata-sections -ffunction-sections -g -gdwarf-2" LDFLAGS="--specs=nosys.specs --specs=nano.spec"
```

Note: it might require the arm-none-eabi-newlib package

Note: make sure to use the path of GNU Arm Embedded Toolchain installed on your system to point to the CC and AR executables, as explained above.

Compilation of Baremetal TEE

The following steps are needed to compile the Baremetal TEE.

- Move into the OS folder of source code (e.g. `user@pc:~/Code/MCU-fortifier/OS`).
- Edit the GCC_PATH and PREFIX variables inside the Makefile in order to point to the installation folder of the GNU ARM Embedded Toolchain on your system. A possible example is the following:
  `GCC_PATH=/home/user/st/stm32cubeide_1.15.0/plugins/com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.12.3.rel1.linux64_1.0.100.202403111256/tools/bin`
  `PREFIX=arm-none-eabi`
- Open a terminal and type: make

  Note: Access to peripherals is denied by default, you can enable it by uncommenting the corresponding line in OS/Makefile, for example:

```
# Enable peripherals. This list includes all the STM32L47x/L48x peripherals.
# Not every peripheral is supported by the specific platform.
#C_DEFS += -DACCESS_SPI
C_DEFS += -DACCESS_UART          # Also allows the access to USARTs/LPUARTs.
#C_DEFS += -DACCESS_I2C
#C_DEFS += -DACCESS_CAN
```

  If the compilation process was successful, the complete executable can be found in the Build folder under the name `MCU-Fortifier.elf`.

Note: The whole compilation process is intended for (and was tested on) Linux environments. In case any other OS is used (e.g. Windows) additional steps and small modifications are required (e.g. installation of MakeTools and Python, definition of environment variables, path changes, etc.).

3.2.2.3 Customization of included Trusted Applications (TAs)

Multiple Global Platform-compliant trusted applications (that offer different services) can be developed for the Baremetal TEE with MPU but only TWO of them can be compiled and run alongside the microvisor at the same time.

3.2.2.4 Global Platform Client API for Trusted Applications (TAs)

In order to be compliant with the GlobalPlatform standard, each TA deployed on the Baremetal TEE MUST implement this following Client API in order to allow the communication with the client (in a standardised way):

```
TEE_Result TA_CreateEntryPoint(void);
TEE_Result TA_OpenSessionEntryPoint(uint32_t param_types, TEE_Param params[4],
void **sessionContext);
TEE_Result    TA_InvokeCommandEntryPoint(void*    sessionContext,    uint32_t
commandID, uint32_t paramTypes, TEE_Param params[4]);
void TA_CloseSessionEntryPoint(void* sessionContext);
void TA_DestroyEntryPoint(void);
```

These functions should be implemented in a file that will act as the entry point to the TA.

Compilation and Deployment

Deploying a Trusted Applications requires different steps and modification to the TEE compilation process. A TA can be inserted in one of the two slots (TA1 and TA2) pre-defined inside the TEE. Depending on which TA slot you want to deploy our TA, it is important to use the variables defined in the `Makefile` and Linker Script (TA1 vs TA2, `TA1_INCLUDE vs TA2_INCLUDE`, etc.) accordingly.

In the following illustrative steps, the first TA slot (TA1) will be used to deploy the Bitcoin Wallet demo TA.

NOTE: in the case of a single TA deployed, it is necessary to fill the TA1 slot first. Also, for the sake of simplicity, it is possible to leave the empty TA2 included in the repository in the second slot if nothing else is used. This will require less modification to the TEE Makefile (otherwise everything related to TA2 should be commented out).

NOTE: in the client application, the first `TEEC_InitializeContext` call assigns the context to TA1. Therefore, all subsequent GP Client API calls with the same context/session will invoke TA1. To execute the functions of TA2, another call to `TEEC_InitializeContext` must be executed. Using this approach, it is not possible to interact with TA2 without having a context set for TA1 (no need to Open a Session), but a more advanced solution with UUID is under consideration.

The required modifications for deploying the TA are as follows:

- Create a subfolder with the name of the TA inside the "`OS/TAs`" folder.
- Modify the Makefile of the TEE (available in the path `OS/Makefile`) according to the following points.
  - Specify the entry point file of the TA (the one that is implementing the Client API functions, e.g. `TA_CreateEntryPoint, TA_OpenSessionEntryPoint)` in the TA1 variable:

```
TA1 = TAs/bitcoin-wallet-ta/bitcoin_wallet_ta.c
```

- Specify the paths of the .h files/folder used by the TA in the `TA1_INCLUDE` variable and prepend "-I" to each path:

```
TA1_INCLUDE = \
-ITAs/bitcoin-wallet-ta \
-ITAs/bitcoin-wallet-ta/include \
-ITAs/bitcoin-wallet-ta/crypto
```

- Specify the other source files used by the TA in the `TA1_C_SOURCES`:

```
TA1_C_SOURCES = \
TAs/bitcoin-wallet-ta/bip39.c \
TAs/bitcoin-wallet-ta/bip32.c \
TAs/bitcoin-wallet-ta/crypto/sha2.c \
TAs/bitcoin-wallet-ta/crypto/ecdsa.c \
TAs/bitcoin-wallet-ta/crypto/hmac.c \
TAs/bitcoin-wallet-ta/crypto/secp256k1.c \
TAs/bitcoin-wallet-ta/crypto/ta_ripemd160.c \
TAs/bitcoin-wallet-ta/crypto/memzero.c \
TAs/bitcoin-wallet-ta/crypto/bignum.c \
TAs/bitcoin-wallet-ta/crypto/pbkdf2.c
```

- Comment/uncomment the following lines of the Makefile based on the number of TAs used and that needs to be compiled. In this case, only the TA one is used, while the compilation for the TA two is not performed:

```
# list of TA1 C objects
MICROVISOR_OBJECTS += $(addprefix $(TA1_BUILD_DIR)/,$(notdir $(TA1_C_SOURCES:.c=.o)))
vpath %.c $(sort $(dir $(TA1_C_SOURCES)))
# list of TA2 C objects
#MICROVISOR_OBJECTS += $(addprefix $(TA2_BUILD_DIR)/,$(notdir $(TA2_C_SOURCES:.c=.o)))
#vpath %.c $(sort $(dir $(TA2_C_SOURCES)))
....
$(TA1_BUILD_DIR)/%.o: %.c Makefile | $(TA1_BUILD_DIR)
$(CC) -c $(CFLAGS) -Wa,-a,-ad,-alms=$(TA1_BUILD_DIR)/$(notdir $(<:.c=.lst)) $< -o $@
#$(TA2_BUILD_DIR)/%.o: %.c Makefile | $(TA2_BUILD_DIR)
#$(CC) -c $(CFLAGS) -Wa,-a,-ad,-alms=$(TA2_BUILD_DIR)/$(notdir $(<:.c=.lst)) $< -o $@
```

- Modify the linkerscript of the TEE (path OS/BOOTLOADER.ld) as following:
  - Specify the name of entry point for the TA followed by .o (in the example bitcoin_wallet_ta) in the definition of the ta1 segment of FLASH_TA1 area. The other TA files, if correctly specified in the TA1_C_SOURCE variable of the Makefile, should be automatically inserted and compiled in the Build/TA/TA1 folder.

```
.code_TA1 : {
  . = ALIGN(4);
   /* Entry point object for TA1 which include GP Client API*/
```

```
 Build/bitcoin_wallet_ta.o(.text  .text*  .rodata  .rodata*  .constdata
.constdata*)
    /* Other objects of TA1 */
  Build/TA/TA1/*.o(.text    .text*    .rodata    .rodata*    .constdata
.constdata*)
  . = ALIGN(4);
} >FLASH_TA1
```

- Specify in the same way the name of the entry point file of the TA in the BSS segment for TA1:

```
.bss_TA1 : {
  . = ALIGN(4);
  *(.bss_TA1)
  /* Entry point object for TA1 which include GP Client API*/
  Build/bitcoin_wallet_ta.o(.bss .bss*)
  /* Other objects of TA1 */
  Build/TA/TA1/*.o(.bss .bss*)
  . = ALIGN(4);
} > RAM_TA1
```

- Add the right include directives for the header files that define the entry points for the Global Platform API. This should be done both for Client API and Core API and in all the files that use them.

```
#include "tee_client_api.h"
#include "tee_core_api.h"
```

After completing these steps it should be possible to compile the Baremetal TEE with a custom TA integrated.

A full list of the available GP APIs can be found in the GitHub repository [21].

3.2.2.5 Fortifying an Application

In order to turn an embedded project into a fortified application that can be executed under the supervision of the Baremetal TEE and be a client of one or both trusted applications, one has to compile it for the Baremetal TEE and implement the logic interacting with the TA(s). For the detailed steps refer to the GitHub repository [22].

3.3 CROSSCON TEE Toolchain

3.3.1 Secure Update – Consumer module

This section describes the Firmware Consumer software module developed in the context of the CROSSCON Secure Update solution. It describles the software components that can be found in the Git repository [23]. These components are meant to be run on the IoT device that performs the update.

The repository contains two main components:

- The CROSSCON SUIT Manifest Parser, an extended version of the SUIT-Parser reference implementation developed by Arm [13] supporting the update of multiple components, a Software Bill Of Materials (SBOM) and a Behavioral Certification Manifest (BCM), alongside the CROSSCON API for the Secure Update.
- The Ethos proof checker, developed by the cvc5 team, which can be used to locally verify the validity of the formal proof certificates contained in a Behavioral Certification Manifest.

The next section focuses on how to use these components in turn.

3.3.1.1 CROSSCON SUIT Manifest Parser

The Manifest Parser is the component that reads an extended SUIT manifest and performs all the operations needed for a successful update (e.g. signature check, proof verification).

In order to build the Manifest Parser, the libcoap3 library is needed. Once the library is installed and available (via the pkg-config facility), the following build options are available:

- `make lib` will build the manifest parser as a library, available at the path `out/source/suit_parser.a`;
- `make` will build both the library (as in the preceding point) and a driver CLI program, available at the path `out/secure_update`.

The secure_update program accepts the following arguments:

```
secure_update <command> <manifest file> <options>
```

where <command> is one of the following strings:

- validate-manifest: this command is used to validate a received update manifest. This includes checking the signature, SBOM validation and verification of the proof certificates.
- extract-sbom: this command is used to extract a Software Bill of Materials from the update manifest.
- extract-properties: this command is used to extract the list of formal properties whose proof is contained in the Behavioral Certification Manifest.
- extract-image: this command is used to extract the update image from the update manifest, in case the image is bundled within the manifest.
- install-image: this is used to install a downloaded update image.
- update: this command is used to extract the binary image from the manifest and install it.
- help: this command shows a brief help message on the console.

The only option recognized is the –key option, which can be used to specify the public key needed to verify the manifest signature.

For example, to validate the manifest contained in the CBOR file `example1.suit` using the public key contained in the file `pubkey.pem`, the command to be issued is:

```
secure_update validate-manifest example1.suit --key pubkey.pem
```

In a production system, the update process will not be managed by the secure_update binary; rather, the API exposed by the CROSSCON Secure Update module will be used directly by the component which needs to perform the update.

3.3.1.2 Ethos Proof Checker

Ethos is a proof checker developed by the cvc5 team. It takes a formal proof certificate as input, specified in the CPC (Cooperating Proof Calculus) language, and checks if the certificate encodes a valid proof or not. This tool is invoked by the secure_update command when a proof certificate found in a Behavioral Certification Manifest needs to be checked locally on device.

For convenience, the repository includes two different executable versions of Ethos. The first one is compiled for the x86_64 architecture, whereas the second one is compiled for the ARM Aarch64 architecture that was used in the tests (performed on a Raspberry Pi 4B).

These two executables are automatically called by the secure_update command when necessary and need not be invoked explicitly by the user.

3.3.2 Secure Update – Infrastructure tools

This section describes all the infrastructural tools developed in the context of the CROSSCON Secure Update solution. It concerns the software components that can be found in the Git repository [24]. These components are not meant to be run on the upgrading device; rather, they concern the creation of the update packages and the various servers needed to the actual deployment of the update.

The repository contains five components:

- The CROSSCON SUIT Manifest Generator.
- The Python scripts for generating the proofs for some selected behavioral properties.
- A minimal implementation of a Firmware Server.
- An SBOM Verifier Server for on-demand vulnerability scanning.
- A Status Server to collect update results from devices.

The high-level workflow to generate an update package conforming to the CROSSCON secure update scheme is the following:

1. Produce the binary image by compiling and linking the firmware sources.
2. Prepare the JSON input file for the Manifest Generator, as in the usual SUIT workflow.
3. Run the appropriate proof-producing scripts on the firmware binary.
4. (optionally) Compress and encode in base64 the proof certificates.
5. Add the proof certificates obtained in point 3 to the JSON input to the Manifest Generator.
6. Run suit-tool to generate and sign the extended SUIT manifest.

The next section dives into more details on how to use each component included in the repository.

3.3.2.1 CROSSCON SUIT Manifest Generator

The Manifest Generator is a Python tool that can be used to generate an extended SUIT manifest. The Manifest Generator can be installed using the instructions in the README file and launched using the suit-tool command.

The suit-tool program accepts the following arguments:

```
suit-tool <command> <options>
```

where <command> is one of the following strings:

- create: this command is used to generate a new manifest starting from a properly formatted JSON file.
- sign: this command is used to cryptographically sign a previously generated manifest.
- verify: this command is used to verify the cryptographic signature of an existing manifest.
- parse: this command is used to parse an existing manifest into a cbor-debug or a json representation.
- keygen: this command is used to generate a signing key. It should not be used in production enviroments.
- pubkey: this command is used to get the public key for a supplied private key in uECC-compatible C definition.

The suit-tool has a built-in configurable logging facility. The logging level can be specified using the option `-l` followed by one of the following values: debug, info, warning, exception.

To **create a manifest**, invoke suit-tool as follows:

```
suit-tool create -i IFILE -o OFILE
```

The format of the JSON input file is thoroughly described in the README of the repository. The tool parses the given input file and produces an extended SUIT manifest. The option `-f` specifies the output format:

- `suit`: CBOR-encoded SUIT manifest (default)
- `suit-debug`: CBOR-debug SUIT manifest
- `json`: JSON-representation of a SUIT manifest

To generate a manifest with severable fields, add the `-s` flag.

To add a component to the manifest from the command-line, use the following syntax:

```
-c 'FIELD1=VALUE1,FIELD2=VALUE2,...'
```

The supported fields are:

- `file` the path fo a file to use as a payload file.
- `inst` the `install-id`.
- `uri` the URI where the file will be found.

To **sign an existing manifest**, invoke suit-tool as follows:

```
suit-tool sign -m MANIFEST -k PRIVKEY -o OFILE
```

where `PRIVKEY` is a secp256r1 ECC private key in PEM format.

If the COSE Signature needs to indicate the key ID, add a key id using the flag `-i`.

To **verify the signature of an existing manifest**, invoke suit-tool as follows:

```
suit-tool verify -k FILE -s SIGNED-MANIFEST
```

The algorithm used to create the signature of the manifest should be one of the following: secp256r1, secp384r1, secp521r1 or ed25519. The result of the verification is printed on the console. You can also specify an output file using the `-o` flag followed by the name of the file.

To **parse an existing manifest**, invoke suit-tool as follows:

```
suit-tool parse -m MANIFEST
```

If a json-representation is needed, add the `-j` flag.

To **generate an asymmetric keypair**, invoke suit-tool as follows:

```
suit-tool keygen -t TYPE -o KEYFILE
```

Valid values for the `-t` flag are the following: secp256r1, secp384r1, secp521r1 or ed25519. If the `-t` flag is omitted, the tool defaults to creating SECP256r1 keys. Note that cryptographic keys created in this way should be used only for testing purposes. Production systems should use closely guarded keys, such as keys stored in a Hardware Security Module (HSM).

To **retrieve a public key** for a supplied private key in the micro ECC format, invoke suit-tool as follows:

```
suit-tool pubkey -k FILE
```

where FILE contains a PEM private key. The tool will then print the public key in micro ECC format. To write the key into a file use the '-o' flag followed by the name of the file.

3.3.2.2 Proof Generating Programs

Two Python programs are provided to generate formal proof certificates for control flow-related properties (`check_cfp.py`) and binary instrumentation properties (`check_instr.py`). Both programs make use of the following dependencies:

- The angr binary analysis framework [14]; Python bindings for it must be available on the system (this may require setting up a virtual environment).
- The cvc5 SMT solver [15]; the corresponding binary must be available in the execution path.

The detailed usage descriptions for the two programs is detailed below:

The **check_cfp** program takes as input two binary files and looks for a proof of the specified control flow-related property. Currently it recognizes three properties of this kind:

1. *Control Flow Invariance*: The Control Flow Graphs of the two binaries are isomorphic. Informally, this means that the updated component has exactly the same possible executions paths of the replaced component.
2. *Control Flow Preservation*: The Control Flow Graph of the original binary embeds into the Control Flow Graph of the updated binary. Informally, this means that the updated component does not remove existing execution paths in the replaced component.
3. *Control Flow Reflection*: The Control Flow Graph of the updated binary embeds into the Control Flow Graph of the original binary. Informally, this means that the updated component does not add new execution paths to the replaced component.

The program can be invoked as follows:

```
python3 check_cfp.py <options> <original> <updated>
```

where <original> is the name of the binary file containing the old version of the binary, and <updated> is the name of the binary file containing the new version of the binary. The available options are:

- `-h`: displays a brief help message
- `-v`: displays the version number of the program
- `-o FILENAME`: specifies the name of the output file where the proof certificate will be written (default is `proof.cpc`)
- `-p PROPERTY`: selects the security property that should be proved. The supported properties are: `cfp` (default), for control flow preservation, `cfr` for control flow reflection, and `cfiso` for control flow invariance.
- `-s SYMBOL`: specifies the starting symbol for the Control Flow Graph reconstruction algorithm. The default is `main`.
- `-t SEC`: specifies the timeout (in seconds) for the invoked cvc5 subprocesses. The default is 3600 (i.e., 1 hour).
- `-m SYM1 SYM2`: this option can be used to add a matching between symbols SYM1 and SYM2 in the control flow graphs of the two programs.
- `-x`: this flag tells the program to only look for a proof of the property, without writing a proof certificate for it.
- `-k`: this flag can be used to keep the temporary SMT-LIB files generated by the tool.

The **check_instr** program takes as input a binary file and looks for a proof of the specified instrumentation-related property. Currently, the tool only supports checking for the absence of a user-defined sequence of instructions (using the `-x` flag); other, more complex kinds of instrumentation can be added in the future.

The program can be invoked as follows:

```
python3 check_instr.py <options> <binary>
```

where <binary> is the name of the binary file to be analyzed. The available options are:

- `-h`: displays a brief help message
- `-v`: displays the version number of the program
- `-o FILENAME`: specifies the name of the output file where the proof certificate will be written (default is `proof.cpc`)
- `-s SYMBOL`: specifies the name of the function to analyze. The default is `main`.
- `-t SEC`: specifies the timeout (in seconds) for the invoked cvc5 subprocesses. The default is 3600 (i.e., 1 hour).
- `-x REGEXP`: specifies the regular expression that identifies the particular instrumentation searched. This expression will be matched against the disassembled code of the specified binary file.
- `-k`: this flag can be used to keep the temporary SMT-LIB files generated by the tool.

Both programs will invoke the cvc5 SMT solver to look for a proof of the specified property. If the property does not hold, the program exits with the message “Property does not hold, aborting”. If the call to the cvc5 solver fails, the program exits with the message “No output from cvc5, aborting” and reporting the error message received from the cvc5 subprocess. If no problems are found, the program writes down the proof certificate and terminates successfully.

3.3.2.3 Firmware Server

The `firmware-server` folder contains a minimal firmware server which is suitable for testing purposes. Two implementations are provided, one over HTTP and another over CoAP.

To launch the HTTP server, run the following command:

```
python3 firmware_server_http.py
```

To launch the CoAP server, make sure that the libcoap3-dev package is installed in your system and install the aiocoap Python package with

```
pip3 install “aiocoap[all]”
```

Then run the CoAP server with

```
aiocoap-fileserver <firmware-image>
```

where <firmware-image> is the path to the firmware image file.

3.3.2.4 SBOM Verifier Server

The `sbom_server` folder contains a simple server for processing a Software Bill Of Materials (SBOM). To use it, first install the grype vulnerability scanner following the instructions on the repository [25]. Then install the required Python dependecies with

```
pip3 install fastapi uvicorn python-multipart
```

Finally start the server by issuing the command

```
uvicorn sbom_service:app --reload --host 10.200.10.10
```

3.3.2.5 Status Tracker Server and Client

The `status-server` folder contains a SUIT-compliant status tracker for the CROSSCON secure update system. It consists of a central server, a command-line management tool, and a client application designed to run on an IoT device.

The following components are provided:

- `suit_server.py`: this is the core application. It runs a Flask web server for API endpoints and a WebSocket server for real-time, bidirectional communication with devices. It manages device status and firmware information in an SQLite database.
- `updater_client.py`: this is the client application intended to run on an actual IoT device. It retrieves its unique MAC address, reports its status, and handles firmware updates by calling an external handler script.
- `update_handler.py`: this is a placeholder script that simulates a real-world firmware installation utility. It is called by the `updater_client` script to verify and apply a downloaded firmware file. It reports success or failure via its exit code.
- `server_cli.py`: this is a user-friendly command-line interface (CLI) for managing the server. It allows an administrator to view devices, list firmwares, and add or delete firmware files.
- `schema.sql`: the SQL schema used to initialize the SQLite database (suit_server.db) with the necessary tables for devices and firmwares.
- `device_simulator.py`: a basic device simulator that can be used for initial testing. It lacks filesystem interaction and external script calls.

To install the required Python dependecies, run the following command:

```
pip3 install flask websockets requests rich
```

Then follow these steps to set up the system:

Clone or download the project files into a single directory.

Launch the main server application. This will create the suit_server.db database file and the firmware_files/ directory if they don't exist.

```
python3 suit_server.py
```

The application output should indicate that both the Flask and WebSocket servers have started successfully.

Start the client application. It will automatically determine its MAC address, read its current version from version.info (or create the file), and connect to the server.

```
python3 updater_client.py
```

The server terminal will show a "Device connected" message.

Managing the Server. You can use the CLI tool `server_cli.py` to interact with the server.

In particular, you can:

- List connected devices, using the `devices` subcommand;
- List available firmware images, using the `firmwares` command;
- Add a new firmware image, using the `add` subcommand;
- Remove a firmware image, using the `delete` subcommand.

If an update fails, when you run `server_cli.py devices` the device's status will be shown in red.

3.4 CROSSCON Trusted Applications

3.4.1 PUF-based authentication

Within the CROSSCON stack three PUF authentication schemes have been proposed - ZK-PUF, PAWOS, PAVOC. All of them have been provided as baremetal applications targeting the LPC55S69 evaluation board and additional effort was into achieving a Trusted Application architecture for ZK-PUF following the GlobalPlatform API specification and security model.

3.4.1.1 ZK-PUF GlobalPlatform

UC1.1 Manifest

ZK-PUF implements a Trusted Application (TA) following the GlobalPlatform API specification and security model. It aims to demonstrate how PUF-based authentication can integrate with a TEE-like architecture even on constrained IoT hardware. This TA resides in the repository [26]. Its trust model relies on the CROSSCON Hypervisor for isolation and integrity. A subset of the GlobalPlatform Core and Client APIs has been implemented to structure communication and improve portability. Unlike full-fledged TEEs such as OP-TEE or MTower, this app uses the Zephyr RTOS because no TEE was available for the LPC55S69 platform when development began.

A practical reference demo is available [27], which includes a GUEST_VM showcasing example communication with the trusted app.

Prerequisites

After cloning the repository, initialize the virtual machines (VMs), the enrolment app, and the hypervisor submodules:

- python3-vev
- make
- LinkServer
- arm-none-eabi-objdump
- lua
- (optional) tio v3.8

After cloning the repository, initialize the virtual machines (VMs), the enrollment app, and the hypervisor submodules:

```
git clone https://github.com/crosscon/UC1.1-Manifest.git
cd UC1.1-Manifest
git submodule update --init --recursive
```

To not install west utility requirements globally using a python virtual environment is suggested. This can be overriden by passing USE_VENV=0 to make.

```
python3 -m venv .venv
source .venv/bin/activate
```

Installing Dependencies

To install python dependencies and Zephyr SDK run:

```
make install-deps
```

And to fetch required code for all zephyr targets run:

```
make update
```

Enrolling

The PUF implementation for the LPCxpresso55S69 requires two non-secret values to reconstruct the PUF response:

- Activation Code
- Key Code

For this reason, the Enrollment App [28] has been created. It extracts these values and prints them over a serial connection, after which they are saved and transformed into binaries for use in the final PUF_VM image.

Important: For this step connect the LPCxpresso55S69 board to your host machine using a USB cable through the P6 Debug Link port. Ensure that the J4 and J10 jumpers are unpopulated.

```
make enroll
```

Two scripts are provided within this step that are chosen based on users environment and input:

- capture_enroll.lua
  a) Automatically captures output and creates neccessary files.
- fallback_capture_enroll.lua
  b) Requires manual parsing of the serial output from the enrollment app.

Expected Output:

```
(...)
Activation code hex saved to /tmp/activation_code.hex
Activation code bin saved to /tmp/activation_code.bin
Intrinsic key hex saved to /tmp/intrinsic_key.hex
Intrinsic key bin saved to /tmp/intrinsic_key.bin
renamed '/tmp/activation_code.bin' -> `/home/user/UC1.1-
Manifest/build/enrollment_data/activation_code.bin
renamed '/tmp/activation_code.hex' -> `/home/user/UC1.1-
Manifest/build/enrollment_data/activation_code.hex
renamed '/tmp/intrinsic_key.bin' -> `/home/user/UC1.1-
Manifest/build/enrollment_data/intrinsic_key.bin
renamed '/tmp/intrinsic_key.hex' -> `/home/user/UC1.1-
```

```
Manifest/build/enrollment_data/intrinsic_key.hex
Output moved to build/enrollment_data/
```

Enrollment must be performed **once per device**. The resulting binaries will be reused in all future builds. Re-enrolling the device will make all previous COM values obsolete as PUF response will differ thus authentication won't be possible with previously used data.

Building

To build all components, apply the enrollment data and align VM start-points within hypervisor:

```
make build
```

Cleaning

To clean build output simply run:

```
make clean
```

Flashing

Connect the LPCxpresso55S69 board to your host machine using a USB cable through the P6 Debug Link port. Currently the script works only with the `LinkServer` utility. Then simply run:

```
make flash
```

All-in-one command

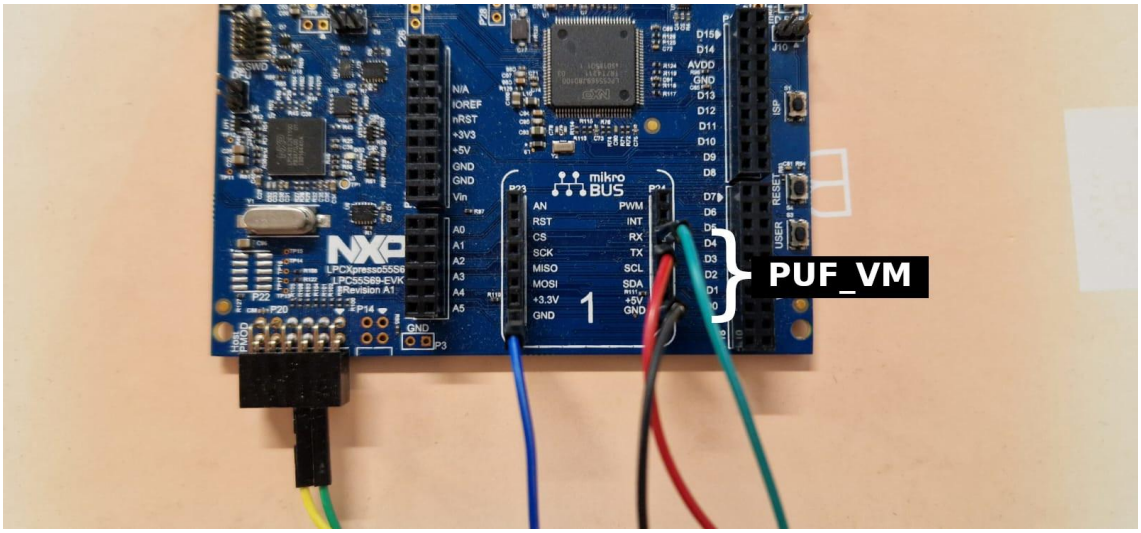
To install dependencies, build everything, and flash the image to the board in one command:

Important: Ensure the LPCxpresso55S69 board is connected via the P6 Debug Link port before running this command, as both enrollment and flashing steps require an active connection.

```
make all
```

Running

GUEST_VM occupies flexcomm 3/UART 3 while PUF_VM occupies flexcomm 2/UART 2.



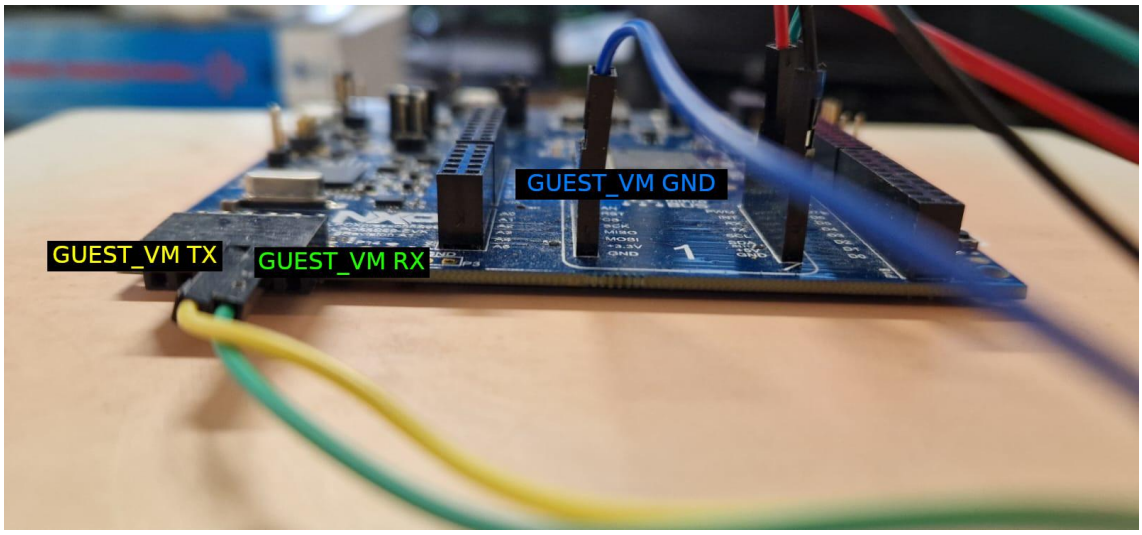


Figure 11: PUF_VM & GUEST_VM Overview

PUF_VM serial gives insight to debugging information while GUEST_VM serial provides an interactive shell from which invocations of Trusted Application functions can be made. Below is a list of all related commands available in the interactive shell.

```
$ help (...)

ta_commit_invoke : Invoke PUF_TA_get_commitment with arguments set by
ta_commit_set

ta_commit_set_challenge1 : Set challenge1 for ta_commit_invoke

ta_commit_set_challenge2 : Set challenge2 for ta_commit_invoke

ta_init : Call PUF_TA_init

ta_zk_get_proofs_invoke : Call PUF_TA_get_ZK_proofs

ta_zk_get_proofs_set_challenge1 : Set challenge1 for ta_zk_get_proofs_invoke

ta_zk_get_proofs_set_challenge2 : Set challenge2 for ta_zk_get_proofs_invoke

ta_zk_get_proofs_set_nonce : Set nonce for ta_zk_get_proofs_invoke

ta_zk_verify_proofs_invoke : Call PUF_TA_get_ZK_proofs
```

```
ta_zk_verify_proofs_set_COM_x : Set COM_x for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_COM_y : Set COM_y for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_P_x : Set P_x for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_P_y : Set P_y for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_g_x : Set g_x for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_g_y : Set g_y for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_h_x : Set h_x for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_h_y : Set h_y for ta_zk_verify_proofs_invoke
```

```
ta_zk_verify_proofs_set_nonce : Set nonce for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_v : Set v for ta_zk_verify_proofs_invoke

ta_zk_verify_proofs_set_w : Set w for ta_zk_verify_proofs_invoke
```

PUF_TA_Init

Must be called before other functions. Initializes PUF hardware and neccessary eliptic curve variables as well as returns *g* and *h*.

PUF_TA_get_commitment

Internally produces responses R_1 and R_2 in response to challenges C_1 and C_2 using the device commits R_1 and R_2 Pedersen commitment into *COM* and returns it.

$R_n = SHA256(SHA256(PUF\ Response, Trusted\ Application\ UUID), C_n)$

Additional hashing with TA UUID serves as a PoC of identity scoping that could happen in a true multi-tenant TEE. This mitigates the risk of some TA compromising platform via misuse or exfiltration of the physically unique secret.

$COM = g^{R1} \cdot h^{R2}$

COM can be stored in a public database as it doesn't disclose any information on R_1 / R_2 or device's PUF response.

Multiple *COM* values can be created using different pairs of C_1 / C_2.

PUF_TA_get_ZK_proofs

Once the device is enrolled, it can use this function to authenticate itself to other devices. The process is initiated by the verifier, which sends challenges C_1 and C_2 , along with a nonce *n*. The nonce ensures the freshness of the authentication process and prevents the replay of old or recorded protocol runs.

Two random values *r* and *u* are created which then formulate a commitment $P=g^r \cdot h^u$.

This is used to create a hash $\alpha = SHA256(P, n)$.

Two zero-knowledge proofs are calculated, denoted as *v* and *w*, where $v=r+\alpha R_1$ and $w=u+\alpha R_2$. These proofs enable to demonstrate knowledge of R_1 and R_2 to the verifier, without disclosing the actual values of R_1 and R_2.

P, *v* and *w* are returned by the function.

PUF_TA_verify_ZK_proofs

To verify device the following equation must hold true

$g^v \cdot h^w = g^{r+\alpha R1} \cdot h^{u+\alpha R2} = g^r \cdot g^{\alpha R1} \cdot h^u \cdot h^{\alpha R2} = P \cdot (g^{R1} \cdot h^{R2})^\alpha = P \cdot COM^\alpha$

The function takes values generated by · prover and returns *TEE_SUCCESS* if the equation holds.

For the security model this function doesn't have to run in the secure world but is provided for clarity.

The struct with all necessary values needs to be provided in a specific order as a memory reference due to nature of TEE communication.

```
typedef struct {
   uint8_t g_x[32];
   uint8_t g_y[32];
   uint8_t h_x[32];
   uint8_t h_y[32];
   uint8_t COM_x[32];
   uint8_t COM_y[32];
   uint8_t P_x[32];
   uint8_t P_y[32];
   uint8_t v[64];
```

```
  uint8_t w[64];
  uint8_t n[64];
} zk_proofs_mem_struct;
```

Handler	Function ID	Parameter 1 (*attr/b*)	Parameter 2 (*attr/b*)	Parameter 3 (*attr/b*)	Parameter 4 (*attr/b*)
PUF_TA_init	0x00112233	*g_x* (*TEE_PARAM_ATTR_TYPE_MEMREF_OUTPUT* / 32 bytes)	*g_y* (*TEE_PARAM_ATTR_TYPE_MEMREF_OUTPUT* / 32 bytes)	*h_x* (*TEE_PARAM_ATTR_TYPE_MEMREF_OUTPUT* / 32 bytes)	*h_y* (*TEE_PARAM_ATTR_TYPE_MEMREF_OUTPUT* / 32 bytes)
PUF_TA_get_commitment	0x11223344	*C_1* (*TEE_PARAM_ATTR_TYPE_MEMREF_INPUT* / 32 bytes)	*C_2* (*TEE_PARAM_ATTR_TYPE_MEMREF_INPUT* / 32 bytes	*COM_x* (*TEE_PARAM_ATTR_TYPE_MEMREF_OUTPUT* / 32 bytes	*COM_y* (*TEE_PARAM_ATTR_TYPE_MEMREF_OUTPUT* / 32 bytes
PUF_TA_get_ZK_proofs	0x22334455	*C_1* / P_x (*TEE_PARAM_ATTR_TYPE_MEMREF_INOUT* / 32 bytes)	*C_2* / P_y (*TEE_PARAM_ATTR_TYPE_MEMREF_INOUT* / 32 bytes)	*n* / *v* (*TEE_PARAM_ATTR_TYPE_MEMREF_INOUT* / 64 bytes)	*w* (*TEE_PARAM_ATTR_TYPE_MEMREF_OUTPUT* / 64 bytes)
PUF_TA_verify_ZK_proofs	0x33445566	*zk_proofs_mem_struct* (*TEE_PARAM_ATTR_TYPE_MEMREF_INPUT* / 448 bytes)	- (*TEE_PARAM_ATTR_TYPE_MEMREF_NONE* / 0 bytes)	- (*TEE_PARAM_ATTR_TYPE_MEMREF_NONE* / 0 bytes)	- (*TEE_PARAM_ATTR_TYPE_MEMREF_NONE* / 0 bytes)

3.4.1.2 Baremetal Implementation

All proposed PUF authentication schemes are also provided as Baremetal applications developed with the MCUXpresso IDE [29][29]. To run them one must clone the repository, build, and flash the corresponding project from within the MCUXpresso IDE.

3.4.2 Context-based authentication

3.4.2.1 Overview

Context-based Authentication (CBA) enables an IoT device in a secure location to authenticate itself to another party (e.g. a fellow IoT device) using Channel State Information (CSI) data collected from its Wi-Fi environment. The device first enrolls CSI data with a remote server, and, during authentication, it sends a fresh measurement of CSI data to the server alongside a nonce. Upon successful validation of the CSI data using a machine learning model, the server returns a signature of the nonce back to the device, which uses it for authentication by verifying the signature.

The implementation provided for CROSSCON runs on a Raspberry Pi 4. Reproducing the demo also requires a separate machine capable of running Docker containers, and both devices must be on the same network. The RPi must be connected to the network via Ethernet since the Wi-Fi chipset cannot be used to capture CSI data and be used for internet connection simultaneously.

The rest of this section outlines the configuration and integration of the CBA Trusted Service. It requires two components to be configured: the trusted service on the CROSSCON hypervisor, and the remote server. It is important that the following steps are followed in order as the configurations partially depend on each other. The final image has three VMs running on the Hyperviser (HV): Besides OP-TEE OS and a host Linux, a third VM is running on the HV to facilitate secure access to the Wi-Fi chipset. This VM is provided pre-built and doesn't require separate configuration or building.

3.4.2.2 Setting up the Remote Server

The TA is connected to the remote server that performs the actual authentication. The remote server should be configured first since the keys used for the communication channel need to be added statically to the TA. The remote server is provided as a Python script in a containerized docker environment. The configuration is done via environment variables, and the installation requires only building the Docker container.

First, clone or download the repository [30] (Release v0.3), and go to the repository's root directory. There, lays a *.env* file with the environment variables to configure. The following table lists the variable names and their purpose.

Table 5: List of environment variables for the remote server

Environment Variable	Description
CA_CERT_PATH	Path to the certificate used for signing the mTLS client certificates
CA_KEY_PATH	Path to the key used for signing the mTLS client certs
SSL_CERT_PATH	Path to the TLS certificate used for the server
SSL_KEY_PATH	Path to the key for the TLS server certificate
SIGN_KEY_PATH	Path to the key for signing nonces upon successful authentication
SIGN_CERT_PATH	Path to the certificate used to verify the signature of the nonces
CSI_DATABASE_PATH	Path to where the CSI enrollment data is stored (e.g. /db)
ML_MODEL_SAMPLES_PER_RECORDING	Number of samples the machine learning model uses for authentication (64)
ML_MODEL_CHECKPOINT_PATH	Path to the checkpoint data of the trained machine learning model (must be mounted as a Docker volume); e2e.pt is provided in this repo
ACCEPTANCE_THRESHOLD	Threshold for number of devices to match between current environment and enrollment (e.g. 5 devices in enrollment, 3 matches found in current environment --> 0.6); float between 0 and 1
PERMIT_RE_ENROLLMENT	Allow multiple enrollment requests with the same client ID
ALLOW_GUEST_NO_CERT	Allow enroll/prove requests with CSI data without a client certificate
DUMP_INCOMING	Write all incoming command data to /app/dump
CONN_ONLY_MODE	For testing the communication between the hypervisor and the remote server. Skips processing the CSI samples and CSI data and can be used to verify that the connection between the remote server and the Pi works.

As a next step, build the Docker container by running the following command from the root directory:

```
docker build -t crosscon-cba-remote:latest .
```

Secure communication between the remote server and the service on the RPi is facilitated through mTLS, and the challenge-response nature of this protocol works through digital signatures. As a result, the server must be equipped with various keys. These keys will also be required when configuring the TA, and they can be generated with the following command:

```
docker run --rm --env-file ./env --volume ./keys:/keys python create_keys.py
```

Furthermore, testing the TA on the RPi requires a digital signature signed with the key generated above. This signature can be created by running (it will be printed in the terminal):

```
docker run --rm --env-file ./env --volume ./keys:/keys python demo_signature.py
```

The volume paths can be arbitrary, as long as the environment variables are adjusted accordingly.

For running the server, using a docker-compose.yml is advised. An example is provided in the repository:

```
services:
  remote:
    image: crosscon-cba-remote:latest
    volumes:
      - "./keys:/keys:ro"
      - "./db:/db"
      - "./e2e.pt:/ml/mlmodel:ro"
    env_file: ".env"
    ports:
      - 5432:5432
    environment:
     - PYTHONUNBUFFERED=1
```

Starting the server is as simple as running the following command:

```
docker compose up -d
```

Setting up the TA on the HV

The demo repository is based on the Hypervisor and TEE Isolation Demos provided by UMINHO and adapted by 3MDEB. This modified OP-TEE OS enables Trusted Applications (TAs) to access Channel State Information (CSI) data from the Wi-Fi peripheral via a platform-independent, GlobalPlatform API-style way. On the RPi, this API is integrated in a second Linux VM, which facilitates communication between the TA running inside OP-TEE OS, and a modified WiFi driver, which handles hardware interactions. Please refer to the repository [31] for more details. This repository also contains the files for running a host Linux VM with a client application for the TA, OP-TEE OS with the TA, and the second Linux VM with the WiFi driver (release *Initial Release*). Clone it and switch into the repository's root directory for the next steps.

It is recommended to use a Docker environment to build and configure the CBA TA on the HV. The hypervisor configuration for running the three VMs (OP-TEE OS + Nexmon VM + Real-world Linux) can be found in `rpi4-ws/configs/rpi4-single-vTEE-dual-linux`.

Table 6 lists several variables which must be adjusted during configuration, and paths for the files that contain them. For all the files, the configuration options are located at the very top. Prior to building any demo, changes, according to the table below, must be made. For the certificates and the demo signature, please see the remote server section. Change the *CSI_PHYSICAL_ADDR_** variables if the configuration mentioned above was applied.

Table 6: List of important files and their corresponding environment variables for CBA

File	Environment Variable	Description
cba_ta/ta/network_handling.c	CONTEXT_BASED_AUTHENTICATION_SERVER_HOST	remote server host
	CONTEXT_BASED_AUTHENTICATION_SERVER_PORT	remote server port
	CONTEXT_BASED_AUTHENTICATION_SERVER_SSL_CERT	remote server TLS certificate

File	Environment Variable	Description
cba_ta/ta/signature_handling.c	CONTEXT_BASED_AUTHENTICATION_SERVER_SIGNATURE_CERT	remote server signature certificate
cba_ta/ta/cba.c	TA_CONTEXT_BASED_AUTHENTICATION_WIFI_CHANNEL	WiFi channel (depends on the access point)
	TA_CONTEXT_BASED_AUTHENTICATION_BANDWIDTH	WiFi channel bandwidth (leave at 20 MHz for the provided machine learning model)
	TA_CONTEXT_BASED_AUTHENTICATION_RECORDING_TIMEOUT	CSI recording timeout
	TA_CONTEXT_BASED_AUTHENTICATION_SAMPLES_PER_DEVICE	CSI samples per device (leave at 64 for the provided machine learning model)
cba_ta/host/main.c	SERVER_TEST_SIGNATURE	Sample signature created by the remote server to demonstrate verification
optee_os/core/pta/csi.c	CSI_PHYSICAL_ADDR_START	base address for shared memory between OP-TEE and Nexmon VM (only change if required)
	CSI_PHYSICAL_ADDR_SIZE	size of shared memory between OP-TEE and Nexmon VM (only change if required)

To build the RPi image, build the docker container first using the following command:

```
docker build -t crosscon_hv:latest -f env/Dockerfile .
```

After the image has been built, run the script `env/run.sh` to start the container and open a shell inside it.

To build the RPi image, run the following command inside the container:

```
env/build_rpi4.sh --all
```

To create and flash the image to SD card the following commands should be executed:

```
sudo env/create_hyp_img.sh
sudo dd if=./crosscon-demo-img.img of=<drive> bs=4M conv=fsync
```

After that, you can run the image by first connecting RPi to your machine using UART to USB adapter and starting up minicom. Supply power to RPI and hit any key when asked to stop u-boot from attempting auto-boot:

```
minicom -D /dev/ttyUSB0 -b 115200
[...]
scanning bus xhci_pci for devices... 2 USB Device(s) found
     scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot:  0
U-Boot>
```

Then you can boot the image by running the following command to load the image into memory and jump to it:

```
fatload mmc 0 0x200000 crossconhyp.bin; go 0x200000
```

During the boot process, the Pi's action LED will blink twice in short succession with a longer break in-between before repeating (like __-_-___-_-_...). Before continuing, please wait for three longer blinks, followed by the LED turning off (_—_—_—_____). This indicates that the second VM is booted successfully.

When asked for a username and password, use the username *root* with an empty password to log in. To receive an IPv4 address via DHCP, use the following command:

```
udhcpc –i eth0
```

Testing the Trusted Service

Once the RPi is running the CROSSCON Hypervisor with a shell connection available via Serial, and the remote server running, the trusted service can be tested with the following procedure.

The first step is enrollement. Invoke the TA's enrollment function with the following command:

```
context_based_authentication_demo enroll &
```

After enrolling the certificate, the TA attempts to collect a current set of CSI measurements, which are forwarded to the server to establish a baseline fingerprint. Verify with the server console logs that a certificate was requested and that CSI samples were received in the CSI enrollment phase.

```
Client connected
ID: <client id>
Client <client id> wants to enroll CSI data with <number of> samples
CSI data enrolled successfully
Client disconnected
```

With the enrollment phase complete, the device can request authentication with the following command:

```
context_based_authentication_demo prove &
```

If the command returns without any errors, the authentication attempt was successful.

Finally, verifying a signature is straight forward. The signature to be verified is statically added to the application at compilation. Simply run:

```
context_based_authentication_demo verify
```

Use the Trusted Service within a custom application

The provided demo host application is designed to showcase the capabilities of the trusted service. The TS can also be used in custom applications by invoking the TA in OP-TEE OS via the GlobalPlatform API. The list below shows the available TA commands, a short description, their parameters, and their return values. When building a custom application, the provided host application in `cba/host` can be used for reference.

- GET_NONCE: can be used to generate a random nonce for the service with the size required for sending along the authentication request.
  - Parameter: MEMREF_OUTPUT (exactly 16 bytes length)
  - Returns: `TEE_SUCCESS`

- ENROLL: enrolls a client certificate for secure communication with the remote server, then enrolls the baseline CSI data to the server; must be run before running the prove command, and must be run exactly once
  - Parameters: none
  - Returns: `TEE_SUCCESS` on success, `TEE_ERROR_EXTERNAL_CANCEL` if the server cancels the request or rejects the CSI data
- PROVE: provided with a nonce, this command collects CSI data from the Pi's environment, sends it to the server with the nonce, and returns the nonce signed by the server if the server deems the CSI data to be authentic
  - Parameters: `MEMREF_INPUT` (exactly 16 bytes, nonce), `MEMREF_OUTPUT` (nonce signed by the server, exactly 512 bytes), `VALUE_OUTPUT` (length of the returned signature, differs from the size of the array)
  - Returns: `TEE_SUCCESS` on success, `TEE_ERROR_EXTERNAL_CANCEL` if the server rejects the authentication attempt
- VERIFY: verifies if a given signature corresponds to a given nonce signed by the server's private key
  - Parameters: `MEMREF_INPUT` (exactly 16 bytes, nonce used to prove creation), `MEMREF_INPUT` (server signature, length exactly as provided as response by the prove command)
  - Returns: `TEE_SUCCESS` on successful verification, `TEE_ERROR_EXTERNAL_CACEL` on failed verification

To include a custom application in the image built by the environment above, add a folder with the file to the configuration option *`BR2_ROOTFS_OVERLAY`* in the file *`support/br-aarch64.config`*. Again, see the folder layout for building the provided demo application for further guidance.

3.4.3 Remote Attestation

3.4.3.1 Overview

Remote Attestation provides a way for an application in one VM (hereafter called Host VM) to attest another VM (hereafter VM-under-test). This service requires limited knowledge about the memory content of the VM-under-test's memory. In case the attestation fails, use-case-specific fallback mechanisms can be triggered.

The attestation service searches for a given pattern in the VM-under-test's memory. Once it finds this pattern, the content of a certain memory segment is hashed and sent to a remote server. This server is equipped with the target memory content and compares the received hash to the hash of the data it is provided. If the hashes match, attestation succeeds, else it fails. The verdict is then sent back to the device, where it can be used by the application which requested the attestation.

Communication between the remote server and the IoT device is secured using mTLS, which requires both a server certificate and a client certificate. The client certificate is enrolled in the IoT device during a separate enrollment phase.

For the CROSSCON project, this service is provided for the Raspberry Pi 4, which is required to run the demo. In addition, a separate machine capable of running Docker containers is required, and the RPi and the machine must be able to communicate via a common network. The RPi must be connected to the network via Ethernet.

The remainder of this section describes how to build the service for a RPi using the CROSSCON Hypervisor. It uses only one Linux VM as both the VM hosting the client application as well as the VM-under-test. A small segment of memory is attested by the service.

3.4.3.2 Configuring and Building the Remote Server

First, the remote server must be built and configured, since its certificate is required by the other components during compilation. The server is provided in a contenirized docker environment, so

building it means building the Docker image. First, clone the Git repository [32] and go into its root directory.

To build the Docker image, run the following command:

```
docker build -t crosscon-ra-remote:latest .
```

With this image, the required certificates can be generated using the following command:

```
docker --rm -e CA_KEY_PATH='/keys/ca_key.pem' -e CA_CERT_PATH='/keys/ca_cert.pem' -e SSL_KEY_PATH='/keys/ssl_key.pem' -e SSL_CERT_PATH='/keys/ssl_cert.pem' --volume ./keys:/keys python create_keys.py
```

As the next step, provide the bytes as they should be located on the VM-under-test's memory to the verifier. For this, create a file called *verified* with the raw bytes. Writing this file can be done e.g. with a short Python script. For this demo, use the following command:

```
python -c "f = open('verified', 'wb'); f.write(bytes([1, 2, 3, 0, 0, 0, 0, 0])); f.close()"
```

Last, run the following command to create an empty file called *ndb*. This is required to ensure every attestation request is used only once:

```
touch ndb
```

Docker is recommended for running the server. For this, the following *docker-compose.yml* file is provided in the repository:

```
service:
  remote:
    image: crosscon-ra-remote:latest
    volumes:
      - "./keys:/keys:ro"
      - "./verified:/verified:ro"
      - "./ndb:/ndb:rw"
    environment:
      - CA_KEY_PATH=/keys/ca_key.pem
      - CA_CERT_PATH=/keys/ca_cert.pem
      - SSL_KEY_PATH=/keys/ssl_key.pem
      - SSL_CERT_PATH=/keys/ssl_cert.pem
      - TARGET_VALUE_PATH=/verified
      - NONCE_DB_PATH=/ndb
    ports:
      - 5432:5432
```

To start the server, run the following command:

```
docker compose up -d
```

It is advised to leave the environment variables in the provided *docker-compose.yml* file as they are. To change the location of the files on the host system, change the paths to the volumes instead of changing the environment variables.

3.4.3.3 Configuring the TA and other components in the CROSSCON Hypervisor

The Trusted Service on the HV consists of a TA running on top of OP-TEE OS, which is extended with additional functionality. As a build system, the CROSSCON Hypervisor demo repository is used.

First, clone the demo repository [33] and go to its root directory.

Then, run the following commands to initialize the submodules:

```
git pull
git submodule init
git submodule update
```

Since Remote Attestation requires a slightly modified version of OP-TEE OS, delete the current OP-TEE OS folder and clone the separate repository. In the same step, also clone the TA.

```
rm -r optee_os
git clone https://github.com/crosscon/remote-attestation-optee-os.git optee_os
cd optee_os
git checkout separate-config-split-work
cd ..

git clone https://github.com/crosscon/remote-attestation-trusted-application.git ra_ta
cd ra_ta
git checkout per-block
cd ..
```

When it comes to configuration, first the hypervisor must be configured. For the demo, the following hypervisor configuration is used.

```
#include <config.h>

VM_IMAGE(linux_image, "../lloader/linux-rpi4.bin");
VM_IMAGE(optee_os_image, "../optee_os/optee-rpi4/core/tee-pager_v2.bin");
struct vm_config linux = {
        .image = {
                .base_addr = 0x20200000,
                .load_addr = VM_IMAGE_OFFSET(linux_image),
                .size = VM_IMAGE_SIZE(linux_image),
},
.entry = 0x20200000,
.type = 0,
.platform = {
.cpu_num = 1,
.region_num = 2,
.regions =  (struct mem_region[]) {
.base = 0x20000000,
.size = 0x40000000,
.place_phys= true,
.phys = 0x20000000
},
.regions=  (struct mem_region[]) {
.base = 0x40000000,
.size= 0x1000,
.place_phys = true,
.phys = 0x40000000
                }
                .ipc_num = 1,
                .ipcs = (struct ipc[]) {
                .base = 0x08000000,
                .size= 0x00200000,
                .shmem_id = 0
                }
```

```
.dev_num = 5,
.devs = (struct dev_region[]) {
{
.pa= 0xfc000000,
.va= 0xfc000000,
.size = 0x03000000
},
{
.pa = 0x600000000,
.va = 0x600000000,
.size = 0x200000000
},
{
.interrupt_num = 182,
.interrupts = (irqid_t[]) {
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104,
105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140,
141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152,
153, 154, 155, 156, 159, 160, 161, 162, 163, 164,
165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176,
177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188,
189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200,
201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212,
213, 214, 215, 216
}
},
{
.pa = 0x7d580000,
.va= 0x7 d580000,
.size = 0x10000,
.interrupt_num = 4,
.interrupts = (irqid_t[]) { 0, 4, 157, 158 }
},
{
.interrupt_num = 1,
.interrupts = (irqid_t[]) { 27 }
}
}
.arch = {
.gic=  {
.gicd_addr = 0xff841000,
.gicc_addr = 0xff842000
}
}
}
```

```
};
struct vm_config optee_os = {
.image = {
.base_addr = 0x10100000,
.load_addr = VM_IMAGE_OFFSET(optee_os_image),
.size = VM_IMAGE_SIZE(optee_os_image)
        },
        .entry = 0x10100000,
        .cpu_affinity = 0xf,
        .type = 1,
        .children_num = 1,
        .children = (struct vm_config*[]) { &linux },
        .platform = {
        .cpu_num = 1,
        .region_num = 2,
        .regions = (struct mem_region[]) {
        {
        .base = 0x10100000,
        .size = 0x00f00000,
        .place_phys= true,
        .phys = 0x10100000
                },
                {
                .base = 0x40000000,
                .size = 0x1000,
                .place_phys = true,
                .phys=  0x40000000,
                }
        },
        .ipc_num = 1,
        .ipcs = (struct ipc[]) {
        {
        .base = 0x08000000,
        .size= 0x00200000,
        .shmem_id= 0
                }
        },
        .dev_num = 2,
        .devs = (struct dev_region[]) {
        {
        .pa = 0xfe215000,
        .va = 0xfe215000,
        .size = 0x1000
                },
                {
                .interrupt_num= 1,
                .interrupts=  (irqid_t[]) { 27 }
                }
        }
        .arch = {
        .gic=  {
```

```
            .gicd_addr = 0xff841000,
            .gicc_addr = 0xff842000,
                }
        }
    }
};
struct config config = {
CONFIG_HEADER
.shmemlist_size = 1,
.shmemlist = (struct shmem[]) {
[0] = { .size = 0x00200000 }
        },
        .vmlist_size = 1,
        .vmlist = { &optee_os }
};
```

This configuration attaches the same physical memory region to multiple VMs. By default, the hypervisor is designed to prevent this behavior. To allow for this, modify the return statement in the function *mem_reserve_ppool_ppages* located in file *CROSSCON-Hypervisor/src/core/mem.c* to always return *true*.

Second, configure the TA. Here, three values must be changed:

- In *ra_ta/ta/remote_attestation_config.h*, update the host name and port to reflect the server host (default port is *5432*).
- In *ra_ta/ta/network_handling.c*, update the variable *REMOTE_ATTESTATION_SERVER_SSL_CERT* with the remote server's TLS certificate as generated previously.

Third, configure the modified OP-TEE OS. OP-TEE was extended with a pseudo-TA (pTA). Unlike a regular TA, a pTA is compiled directly into OP-TEE OS and runs with OP-TEE OS kernel privileges. It can be invoked by other TAs via GlobalPlatform API-compatible mechanisms. Kernel privileges are mandatory for directly reading memory, which is required for taking the measurement. As such, it is called by the TA to perform the measurement on its behalf. For this, the pTA must know the memory address of the VM that is being validated. Multiple memory segments can be configured if multiple different VMs need to be verified.

Update *optee_os/core/pta/memread.c*, and set the variable *config* to the following value:

```
struct vm_mem_mapping config config = {
.mappings_num = 1,
.mappings = (struct vm_mem_mapping[]) {
[0] = {
.phy= 0x40000000,
.size = 0x1000
}
}
};
```

When using custom VMs with custom memory alignments, update the address and size to correspond to the memory address mapped to both OP-TEE OS and the VM-under-test in the HV config.

Lastly, the build system must be configured to add the TA and its host application to the Linux filesystem. For this, add the following entry to the *BR2_ROOTFS_OVERLAY* variable in the file *support/br-aarch64.config:*

```
../ra_ta/to_buildroot-aarch64
```

3.4.3.4 Building the HV image

Now that all elements are configured, the build process itself can begin. It follows largely the instructions in *rpi4-ws/README.md*, which provides further insight into the build process. This section will yield a finished SD card, from which the demo can be booted. For the sake of conciseness, most commands are referenced to the README.md file.

As a first step, the SD card must be cleaned and formatted with the required volumes. Then, the Raspberry Pi and ARM firmwares as well as the bootloader must be built and copied to the prepared SD card. For this, please refer to the sections at the top of the README.md file.

After that, the OP-TEE OS and Linux VMs as well as the hypervisor must be built. For building OP-TEE OS, parts of the Linux filesystem, and the OP-TEE client, again see steps 1, 2 and 3 for *Building components* in the README.md.

Then, build the Remote Attestation TA with the following command set (assuming the prior commands were executed as indicated by the README):

```
cd ra_ta

BUILDROOT=`pwd`/../buildroot/build-aarch64/
export CROSS_COMPILE=$BUILDROOT/host/bin/aarch64-linux-
export HOST_CROSS_COMPILE=$BUILDROOT/host/bin/aarch64-linux-
export TA_CROSS_COMPILE=$BUILDROOT/host/bin/aarch64-linux-
export ARCH=aarch64
export PLATFORM=plat-virt
export TA_DEV_KIT_DIR=`pwd`/../optee_client/optee-rpi4/export-ta_arm64
export TEEC_EXPORT=`pwd`/../optee_client/out-aarch64/export/usr/
export OPTEE_CLIENT_EXPORT=`pwd`/../optee_client/out-aarch64/export/usr/
export CFG_TA_OPTEE_CORE_API_COMPAT_1_1=n
export DESTDIR=./to_buildroot-aarch64
export DEBUG=0
export CFG_TEE_TA_LOG_LEVEL=0
export O=`pwd`/out-aarch64

rm -rf out-aarch64/
make clean
make -j`nproc`

mkdir -p to_buildroot-aarch64/lib/optee_armtz
mkdir -p to_buildroot-aarch64/bin

cp out-aarch64/*.ta to_buildroot-aarch64/lib/optee_armtz
cp host/remote_attestation_demo to_buildroot-aarch64/bin/remote_attestation_demo
chmod +x to_buildroot-aarch64/bin/remote_attestation_demo

cd ..
```

After building the TA, execute the steps 7, 8 and 9 from the README.md instructions.

The last step after building the components is to build the final HV image file. For this, the following command is used:

```
CONFIG_REPO=$(pwd)/CROSSCON-Hypervisor/configs

make -C CROSSCON-Hypervisor/ \
PLATFORM=rpi4 \
CONFIG_BUILTIN=y \
```

```
CONFIG_REPO=$CONFIG_REPO \
CONFIG=rpi4-single-vTEE \
OPTIMIZATIONS=0 \
SDEES="sdSGX sdTZ" \
CROSS_COMPILE=aarch64-none-elf- \
Clean

make -C CROSSCON-Hypervisor/ \
PLATFORM=rpi4 \
CONFIG_BUILTIN=y \
CONFIG_REPO=$CONFIG_REPO \
CONFIG=rpi4-single-vTEE \
OPTIMIZATIONS=0 \
SDEES="sdSGX sdTZ" \
CROSS_COMPILE=aarch64-none-elf- \
-j$(nproc)

cp CROSSCON-Hypervisor/bin/rpi4/builtin-configs/rpi4-single-vTEE/crossconhyp.bin .
```

As the final step, copy the file `crossconhyp.bin` from the repository root directory to the root directory of the SD Card prepared in earlier steps.

3.4.3.5 Testing the Trusted Service

After building the image, run it by first connecting RPI to the machine using UART to USB adapter and starting the minicom. Supply power to RPI and hit any key when asked to stop u-boot from attempting auto-boot:

```
minicom -D /dev/ttyUSB0 -b 115200
[...]
scanning bus xhci_pci for devices... 2 USB Device(s) found
      scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot:  0
U-Boot>
```

Then the image can be booted by running the following command to load the image into memory and jump to it:

```
fatload mmc 0 0x200000 crossconhyp.bin; go 0x200000
```

When asked for a username and password, login as *root* with an empty password. To get an IP address via DHCP, run the following command.

```
udhcpc -i eth0
```

First, a client certificate must be enrolled with the TA. For this, use the command below to run the enrollment procedure.

```
remote_attestation_demo enroll
```

The secure communication can then be tested with the following command:

```
remote_attestation_demo test
```

To emulate the behavior of a VM-under-test, use the *devmem* command to write specific values to the memory. The demo application is configured to search for the pattern [1, 2, 3], and then measure a

memory region of length 8 bytes (including the pattern). For a positive attestation, write the desired values to the memory region under test and request attestation.

```
devmem 0x40000000 64 0
devmem 0x40000000 8 1
devmem 0x40000001 8 2
devmem 0x40000002 8 3

remote_attestation_demo attest
```

To showcase an unsuccessful attestation attempt, modify at least one of the bytes within the region, but not part of the pattern:

```
devmem 0x40000005 8 5

remote_attestation_demo attest
```

To test the continuous attestation feature even without a connection to the remote server, unplug the network cable before running the attestation command. Then, run the command below and plug the cable back in. The attestation service will automatically connect to the server as soon as it is reachable again and complete the attestation request.

```
remote_attestation_demo queue &
```

Finally, for a "Not found" scenario, run the following command to distort the pattern and run the attestation.

```
devmem 0x40000000 8 0

remote_attestation_demo attest
```

3.4.3.6 Using the Remote Attestation Service within a Custom Application

The provided demo host application is designed to showcase the capabilities of the remote attestation trusted service. The service can also be used in custom applications by invoking the TA in OP-TEE OS via the GlobalPlatform API. The list below shows the available TA commands, a short description, their parameters, and their return values. When building a custom application, the provided host application in *ra_ta/host* can be used for guidance.

- **ENROLL_CERT**: instructs the TA to enroll a client certificate, which will be used for secure communication with the remote server; must be run exactly once before running the `request_attestation` command
  Parameters: none
  Returns: `TEE_SUCCESS` on success, `TEE_EXTERNAL_CANCEL` on server abort
- **REQUEST_ATTESTATION**: takes a measurement and tries to send it to the remote server for attestation if possible; if the server is unreachable, it is put into the queue
  Parameters: `VALUE_INPUT` (index of the VM to be attested as defined in the OP-TEE OS pseudo-TA), `MEMREF_INPUT` (memory pattern which is used as a starting point for the attestation), `VALUE_INPUT` (number of bytes to be included in the measurement, including the memory pattern)
  Returns: `TEE_SUCCESS` on successful attestation, `TEE_ERROR_EXTERNAL_CANCEL` on failed attestation, `TEE_ERROR_COMMUNICATION` on communication issue (the request will be put into the queue for later attestation)
- **REQUEST_ATTESTATION_FROM_QUEUE**: waits for the remote server to be reachable, then takes the top attestation request stored in the queue and forwards it to the server; must be invoked separately for each attestation request in the queue

Parameters: none
Returns: `TEE_SUCCESS` on successful attestation, `TEE_ERROR_EXTERNAL_CANCEL` on failed attestation

- **TEST_REMOTE**: used to test secure mTLS connection to the remote side
  Parameters: none
  Returns: `TEE_SUCCESS` on successful connection, `TEE_ERROR_COMMUNICATION` on communication errors

3.4.4 Control-Flow Integrity

The CROSSCON nonMPU-BareMetal TEE includes a Control-Flow Integrity (CFI) module that protects both forward- and backward-edges of the application control-flow. The backward edge protection is achieved with a shadow stack: each call and each return instruction is instrumented to populate and check a shadow stack, respectively. The shadow stack is protected through the secure storage of the underlying bare-metal TEE. As for the forward-edges protection, every time the application needs to jump to an unknown location in the code (dynamic jump/branch), the TEE is invoked. This will then check if the destination address is legitimate by comparing it with a set of allowed destinations. This set, which stems from a previously obtained control flow graph of the application, is stored in the secure storage as well.

The CFI module can be enabled by configuring the TCM. This can be done through the `TCM/core/src/core.h` header file which contains several macros that define the behaviour of the TEE. Notably, the `FLASHADOW_ENABLE` macro allows the TEE to be shipped with the control flow integrity module.

3.4.5 Secure FPGA Provisioning

This section explains how to deploy and use the Secure FPGA Provisioning stack on the AMD/Xilinx ZCU102 platform within the CROSSCON environment. It covers the bill of materials, build and installation steps for the Trusted and Regular VM, how to prepare an encrypted configuration package, and how client applications (CAs) request secure partial reconfiguration of virtual FPGAs (vFPGAs). The guidance aligns with the UC5 architecture and process flows validated in D5.6 [38].

3.4.5.1 Architecture Overview

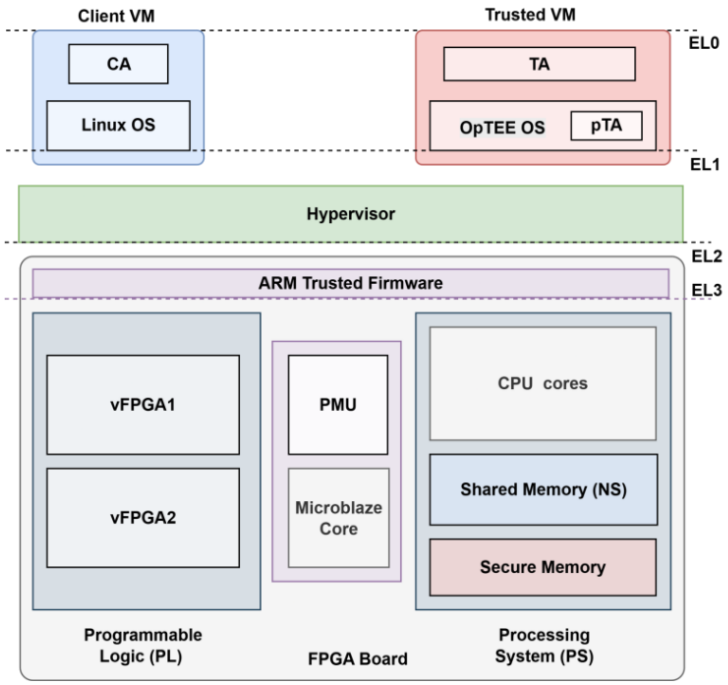


Figure 12: Stack Architecture for Secure FPGA Provisioning

- As described in Figure 12, two VMs run on top of the CROSSCON Hypervisor: A Trusted VM hosting OP-TEE (Trusted Application (TA) and psedo TA (pTA)) and a Linux (Regular) VM hosting the client application (CAs).
- Secure DRAM is mapped only in the Trusted VM and exception level 3 (EL3) firmware; the hypervisor forwards the pTA's reconfiguration SMC to ARM Trusted Firmware (TF-A). Only this privileged path can trigger the PMU to load a partial bitstream.
- The FPGA is floor-planned into a static part and two reconfigurable partitions (vFPGA1, vFPGA2). LEDs in each vFPGA are used for functional validation after configuration.

3.4.5.2 Configuration used

In this section, the different configurations used in building this stack are described. High-level users do not need modify these individually and can utilize the interactive script provided with the setup.

1. CROSSCON Hypervisor

The setup is built on top of the CROSSCON hypervisor. Following is the configuration used in this setup.

```
#include <config.h>

// Linux Image
VM_IMAGE(linux_image, "../lloader/linux-zcu.bin");

// Linux VM configuration
struct vm_config linux = {
    .image = {
        .base_addr = 0x00200000,
        .load_addr = VM_IMAGE_OFFSET(linux_image),
        .size = VM_IMAGE_SIZE(linux_image),
    },
    .entry = 0x00200000,

    .type = 0,

    .platform = {
        .cpu_num = 1,
        .region_num = 1,
        .regions =  (struct mem_region[]) {
            {
                .base = 0x00000000,
                .size = 0x20000000,
            }
        },
        .ipc_num = 1,
        .ipcs = (struct ipc[]) {
            {
                .base = 0x42000000,
                .size = 0x00800000,
                .shmem_id = 0,
            },
        },
        .dev_num = 4,
        ...
};

VM_IMAGE(optee_os_image, "../optee_os/optee-aarch64/core/tee-pager_v2.bin");
struct vm_config optee_os = {
    .image = {
        .base_addr = 0x60000000,
        .load_addr = VM_IMAGE_OFFSET(optee_os_image),
        .size = VM_IMAGE_SIZE(optee_os_image),
    },
```

```
    .entry = 0x60000000,
    .cpu_affinity = 0xf,

    .children_num = 1,
    .children = (struct vm_config*[]) { &linux },

    .type = 1,


    .platform = {
        .cpu_num = 1,
        .region_num = 1,
        .regions = (struct mem_region[]) {
            {
            // Region 1: 16MB of RAM before the shared memory
            .base = 0x60000000,
            .size = 0x01000000, // 16MB
            .phys = 0x60000000,
            .place_phys = true,
            },
        },
        .ipc_num = 1,
        .ipcs = (struct ipc[]) {
            {
                .base = 0x42000000,
                .size = 0x00800000,
                .shmem_id = 0,
            }
        },
        .dev_num = 1, // UC5
        .devs = (struct dev_region[]) {
            {
                .va = 0xFF000000,
                .pa = 0xFF000000,
                .size = 0x10000,
            },
        },
        .arch = {
            .gic = {
                .gicc_addr = 0xF9020000,
                .gicd_addr = 0xF9010000
            },
        }
    },
};


struct config config = {

    CONFIG_HEADER
    .shmemlist_size = 1,
    .shmemlist = (struct shmem[]) {
        [0] = { .size = 0x00800000, .phys = 0x42000000, .place_phys = true },
    },
    .vmlist_size = 1,
    .vmlist = {
        &optee_os
    }
};
```

2. OPTEE configuration

The OPTEE build is done with the following configuration:

```
make -C ./ \
  O="./optee-aarch64" \
  CROSS_COMPILE="aarch64-none-elf-" \
  PLATFORM="zynqmp" \
  PLATFORM_FLAVOR="zcu102" \
  ARCH="arm" \
  CFG_PKCS11_TA=n \
  CFG_SHMEM_START="0x42000000" \
  CFG_SHMEM_SIZE="0x00800000" \
  CFG_CORE_DYN_SHM=n \
  CFG_NUM_THREADS=1 \
  CFG_CORE_RESERVED_SHM=y \
  CFG_CORE_ASYNC_NOTIF=n \
  CFG_TZDRAM_SIZE="0x02000000" \
  CFG_TZDRAM_START="0x60000000" \
  CFG_TEE_RAM_VA_SIZE="0x800000" \
  CFG_GIC="n" \
  CFG_ARM_GICV2=y \
  CFG_CORE_IRQ_IS_NATIVE_INTR=n \
  CFG_ARM64_core=y \
  CFG_USER_TA_TARGETS=ta_arm64 \
  CFG_DT=n \
  CFG_CORE_ASLR=n \
  CFG_CORE_WORKAROUND_SPECTRE_BP=n \
  CFG_CORE_WORKAROUND_NSITR_CACHE_PRIME=n \
  CFG_TEE_CORE_LOG_LEVEL=4 \
  CFG_TA_LOG_LEVEL=4 \
  CFG_VULN_PTA=y \
  DEBUG=1 -j$(nproc)
```

Further, the TA build is configured with the following definitions.

```
#define TA_FLAGS (TA_FLAG_SINGLE_INSTANCE | TA_FLAG_MULTI_SESSION |
TA_FLAG_INSTANCE_KEEP_ALIVE)

#define TA_STACK_SIZE      (16 * 1024)
#define TA_DATA_SIZE       (6 * 1024 * 1024) // 6MB
#define PGT_CACHE_SIZE     (16) // Increased for more page table entries
```

3.4.5.3 Installation Guidelines

1. Get the resources

- The necessary code and details can be found in the CROSSCON repository [34]

```
git clone https://github.com/crosscon/UC5-IP_Protection_for_Secure_Multi-Tenancy_on_FPGA.git
cd UC5-IP_Protection_for_Secure_Multi-Tenancy_on_FPGA
export ROOT=$(pwd)
```

- The Linux VM is a Buildroot based custom Linux OS. The client applications can be downloaded from the cloud repository [9]
- Download and unzip in the Buildroot directory in the $ROOT folder.

2. Get the ZCU 102 firmware

- The necessary firmware is available [35]
- Unzip the contents and copy to the project location using the following commands:

```
cd $ROOT
cp -r uc5_zcu102_firmware zcu-ws/
cd $ROOT
```

- Now, generate BOOT.BIN with this firmware using the following commands:

```
cd $ROOT
cd zcu-ws/uc5_zcu102_firmware
```

```
bootgen -arch zynqmp -image bootgen.bif -w -o BOOT.BIN
cp BOOT.BIN ../bin
cd $ROOT
```

3. Get default bitstreams

- Default bitstreams are used to wipe the vFPGAs when a deallocation is requested by a client.

The default bitstreams can be found in reference [36]

- Unzip the contents and copy to the project location using the following command:

```
 cd $ROOT
cp default_bitstreams/* optee_os/ta/ta_com/
cd $ROOT
```

4. Prepare an SD-card to be used for booting the ZCU 102 board.

In this example, the SD card is `/dev/mmcblk0` and the partitions are `/dev/mmcblk0p1`, `/dev/mmcblk0p2`, etc. Modify it according to your system.

i) Make sure all partitions are unmounted

```
umount /dev/mmcblk0*
```

ii) Delete all partitions

```
sudo fdisk /dev/mmcblk0
```

Then run the commands:

```
* Press `d` until there are no more partitions (if it asks you for the partition, press `return` for the default)
* Press `w` write changes and exit
```

iii) Create partition

```
sudo fdisk /dev/mmcblk0
```

Then run the commands:

```
- `o` to create a new empty DOS partition table
- `n` to create a new partition. Select the following options:
   - `p` to make it a primary partition
   - the automatically assigned partition number by pressing `return`
   - `16384` (this gap is needed for some of the selected boards)
   - the max default size by pressing `return`
   - if it asks you to remove the file system signature press `y`
- `a` to make the partition bootable
- `t` to set the partition type:
   - type `c` for W95 FAT32 (LBA)
- `w` to write changes and exit
```

iv) Format partition

Format the created partition to a FAT filesystem:

```
sudo mkfs.fat /dev/mmcblk0p1 -n boot
```

Remove and insert the SD card to automatically mount it. The setup is now ready to be built.

5. Building the stack

- Building the stack can be done with a user-friendly script provided in the repository. The script can be run with the following command:

```
./rebuild_and_deploy.sh
```

Follow the on-screen steps. When prompted with the following:

```
CROSSCON Hypervisor shared memory support (CROSSCONHYP_SHMEM) [N/m/y/?] (NEW) y
```

Press y and [ENTER]. When finished, the followign message shall be printed:

```
================================================================================
=====
ALL STEPS COMPLETED SUCCESSFULLY!"

The build artifacts were copied and the SD card has been safely unmounted."
To boot the system, insert the card into the ZCU102 board and run the"
following command from the U-Boot serial console:"
fatload mmc 0 0x200000 crossconhyp.img; bootm start 0x200000; bootm loados; bootm go"
================================================================================
=====
```

- Internally, the script builds OPTEE VM with the TA and pTA, and two client applications on the Linux VM using Buildroot to generate the filesystem.

Usage

1. Ensure that the ZCU 102 board is configured to boot in SD Card mode. Information regarding AMD ZCU 102 can be found in this AMD User Guide [37].

2. On the ZCU board, the orientation of the SW6 switches is used to configure SD card mode is shown in Figure 13.

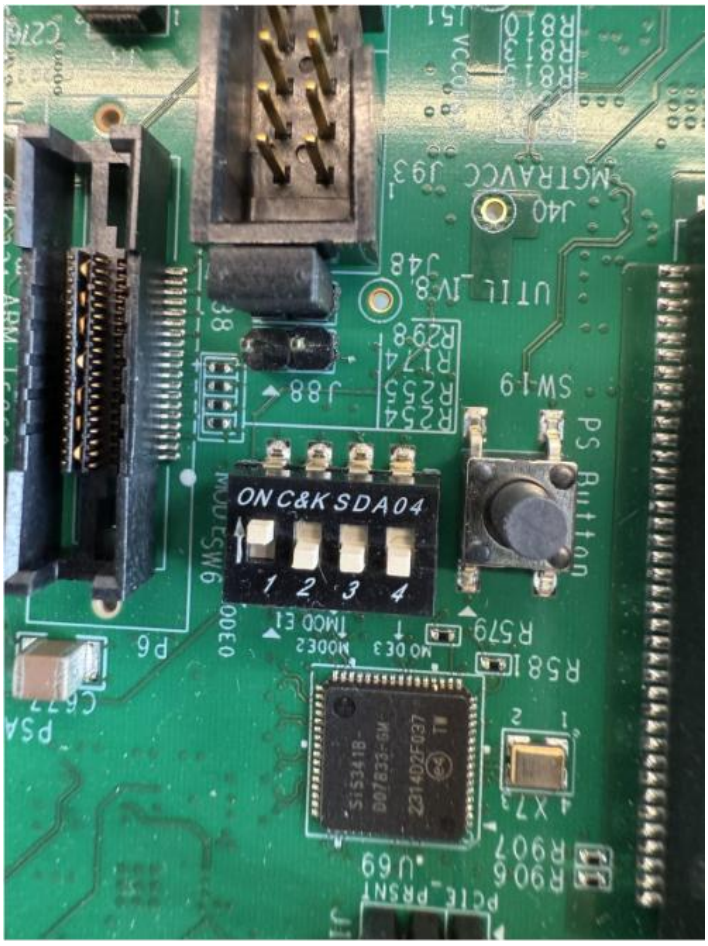


Figure 13: SW6 switch configuration to boot ZCU102 board from SD card

- Insert the SD card in the board's SD card slot.
- Connect the board to your host machine with the available micro-USB JTAG/UART port. In the example setup, the two ports were identified as/dev/ttyUSB0 and /dev/ttyUSB1. Open a new terminal and connect to it. For example:

```
sudo miniterm -D /dev/ttyUSB0 -b 115200
```

```
sudo miniterm -D /dev/ttyUSB1 -b 115200
```

- Turn on the board.
- Run u-boot commands. Quickly press any key to skip autoboot. If time-frame was missed, press `ctrl-c` until the U-Boot prompt is visible. Then load the crossconhyp image, and jump to it:

```
fatload mmc 0 0x200000 crossconhyp.img; bootm start 0x200000; bootm loados; bootm go
```

- The firmware, crossconhyp, and its guests should be printing messages on the UARTs as shown below:

```
ZynqMP>     fatload mmc 0 0x200000 crossconhyp.img; bootm start 0x200000; bootm loados; bootm go
58241304 bytes read in 3862 ms (14.4 MiB/s)
## Booting kernel from Legacy Image at 00200000 ...
   Image Name:   crossconhyp_uboot
   Created:      2025-07-28  19:27:14 UTC
   Image Type:   AArch64 Linux Kernel Image (uncompressed)
   Data Size:    58241240 Bytes = 55.5 MiB
   Load Address: 00200000
   Entry Point:  00200000
   Verifying Checksum ... OK
Working FDT set to 0
Host not halted after 16000 microseconds.
   Loading Kernel Image

Starting kernel ...



  _____ _____   ____   _____ _____  _____ ____  _   _
 / ____|  __ \ / __ \ / ____/ ____|/ ____/ __ \| \ | |
| |    | |__) | |  | | (___| (___ | |   | |  | |  \| |
| |    |  _  /| |  | |\___ \\___ \| |   | |  | | . ` |
| |____| | \ \| |__| |____) |___) | |___| |__| | |\  |
 \_____|_|  \_\\____/|_____/_____/ \_____\____/|_| \_|

  _    _                        _
 | |  | |                      (_)
 | |__| |_   _ _ __    ___ _ __ _____   ___ ___  ___  _ __
 |  __  | | | | '_ \ / _ \ '__\ \ / / / __|/ _ \| '__|
 | |  | | |_| | |_) |  __/ |   \ V /| \__ \ (_) | |
 |_|  |_|\__, | .__/ \____|_|    \_/ |_|___/\___/|_|
          __/ | |
         |___/|_|

CROSSCONHYP WARNING: smmuv2 does not support coherent page table walks
CROSSCONHYP INFO: Initializing VM 1
NNOOTTCROSSCONHYP INFO: VM 1 adding memory region, VA 0x60000000 size 0x1000000
CROSSCONHYP INFO: VM 1 adding MMIO region, VA: 0xff000000 size: 0xff000000 mapped at 0xff000000
CROSSCONHYP INFO: VM 1 adding IPC for shared memory 0 at VA: 0x42000000  size: 0x400000
CROSSCONHYP INFO: VM 1 adding memory region, VA 0x42000000 size 0x400000
CROSSCONHYP INFO: VM 1 is sdTZ (OP-TEE)
CROSSCONHYP INFO: Initializing VM 2
CROSSCONHYP INFO: VM 2 adding memory region, VA 0x0 size 0x20000000
```

CROSSCONHYP INFO: VM 2 adding MMIO region, VA: 0x80000000 size: 0x80000000 mapped at 0x80000000
CROSSCONHYP INFO: VM 2 adding MMIO region, VA: 0xf9030000 size: 0xf9030000 mapped at 0xf9030000
CROSSCONHYP INFO: VM 2 adding MMIO region, VA: 0x0 size: 0x0 mapped at 0x0
CROSSCONHYP INFO: VM 2 assigning interrupt 40
CROSSCONHYP INFO: VM 2 assigning interrupt 41
…
CROSSCONHYP INFO: VM 2 assigning interrupt 187
CROSSCONHYP INFO: VM 2 adding MMIO region, VA: 0x0 size: 0x0 mapped at 0x0
CROSSCONHYP INFO: VM 2 assigning interrupt 27
CROSSCONHYP INFO: VM 2 adding IPC for shared memory 0 at VA: 0x42000000 size: 0x400000
CROSSCONHYP INFO: VM 2 adding memory region, VA 0x42000000 size 0x400000
CROSSCONHYP INFO: VM 2 is sdGPOS (normal VM)
CROSSCONHYP INFO: VM 1 is parent of VM 2
D/TC:0 add_phys_mem:635 ROUNDDOWN((((0xFF000000)) & ~((__typeof__((0xFF000000)))((1U << (21U))) - 1)), CORE_MMU_PGDIR_SIZE) type IO_SEC 0xff000000 size 0x00200000
D/TC:0 add_phys_mem:635 ROUNDDOWN(0xFFC80000U, CORE_MMU_PGDIR_SIZE) type IO_NSEC 0xffc00000 size 0x00200000
D/TC:0 add_phys_mem:635 ROUNDDOWN(0xA0000000U, CORE_MMU_PGDIR_SIZE) type IO_NSEC 0xa0000000 size 0x00200000
D/TC:0 add_phys_mem:635 VCORE_UNPG_RX_PA type TEE_RAM_RX 0x60000000 size 0x00099000
D/TC:0 add_phys_mem:635 VCORE_UNPG_RW_PA type TEE_RAM_RW 0x60099000 size 0x00167000
D/TC:0 add_phys_mem:635 TA_RAM_START type TA_RAM 0x60200000 size 0x01e00000
D/TC:0 add_phys_mem:635 TEE_SHMEM_START type NSEC_SHM 0x42000000 size 0x00400000
D/TC:0 add_va_space:675 type RES_VASPACE size 0x00a00000
D/TC:0 add_va_space:675 type SHM_VASPACE size 0x02000000
D/TC:0 dump_mmap_table:800 type TEE_RAM_RX va 0x60000000..0x60098fff pa 0x60000000..0x60098fff size 0x00099000 (smallpg)
D/TC:0 dump_mmap_table:800 type TEE_RAM_RW va 0x60099000..0x601fffff pa 0x60099000..0x601fffff size 0x00167000 (smallpg)
D/TC:0 dump_mmap_table:800 type SHM_VASPACE va 0x60200000..0x621fffff pa 0x00000000..0x01ffffff size 0x02000000 (pgdir)
D/TC:0 dump_mmap_table:800 type RES_VASPACE va 0x62200000..0x62bfffff pa 0x00000000..0x009fffff size 0x00a00000 (pgdir)
D/TC:0 dump_mmap_table:800 type NSEC_SHM va 0x62c00000..0x62ffffff pa 0x42000000..0x423fffff size 0x00400000 (pgdir)
D/TC:0 dump_mmap_table:800 type TA_RAM va 0x63000000..0x64dfffff pa 0x60200000..0x61ffffff size 0x01e00000 (pgdir)
D/TC:0 dump_mmap_table:800 type IO_NSEC va 0x64e00000..0x64ffffff pa 0xa0000000..0xa01fffff size 0x00200000 (pgdir)
D/TC:0 dump_mmap_table:800 type IO_SEC va 0x65000000..0x651fffff pa 0xff000000..0xff1fffff size 0x00200000 (pgdir)
D/TC:0 dump_mmap_table:800 type IO_NSEC va 0x65200000..0x653fffff pa 0xffc00000..0xffdfffff size 0x00200000 (pgdir)
D/TC:0 core_mmu_xlat_table_alloc:526 xlat tables used 1 / 5
D/TC:0 core_mmu_xlat_table_alloc:526 xlat tables used 2 / 5
I/TC:
I/TC: OP-TEE version: d015c528e-dev (gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)) #1 Mon Jul 28 19:25:59 UTC 2025 aarch64
…
I/TC: Primary CPU initializing

```
D/TC:0 0 call_initcalls:40 level 1 register_time_source()
D/TC:0 0 call_initcalls:40 level 1 teecore_init_pub_ram()
D/TC:0 0 call_initcalls:40 level 3 check_ta_store()
D/TC:0 0 check_ta_store:417 TA store: "Secure Storage TA"
D/TC:0 0 check_ta_store:417 TA store: "REE"
D/TC:0 0 call_initcalls:40 level 3 verify_pseudo_tas_conformance()
D/TC:0 0 call_initcalls:40 level 3 tee_cryp_init()
D/TC:0 0 call_initcalls:40 level 4 tee_fs_init_key_manager()
D/TC:0 0 call_initcalls:40 level 6 mobj_init()
D/TC:0 0 call_initcalls:40 level 6 default_mobj_init()
I/TC: Primary CPU switching to normal world boot
I/TC: Reserved shared memory is enabled
I/TC: Dynamic shared memory is disabled
I/TC: Normal World virtualization support is disabled
I/TC: Asynchronous notifications are disabled
```

```
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[    0.000000] Linux version 5.11.0-g0eb9b5b5f26d-dirty
…
[    0.000000] printk: bootconsole [cdns0] enabled
[    0.000000] CPU features: detected: Kernel page table isolation (KPTI)
…
[    1.805577] Freeing unused kernel memory: 30336K
[    1.810314] Run /init as init process
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving random seed: [    1.889002] random: dd: uninitialized urandom read (512 bytes read)
OK
Starting tee-supplicant: Using device /dev/teepriv0.
OK
Starting network: OK
ERR [145] TEES:main:921: failed to find an OP-TEE supplicant device

Welcome to Buildroot
buildroot login:
```

Login using the credential ‘root’ (no password required). Following are the services available to clients.

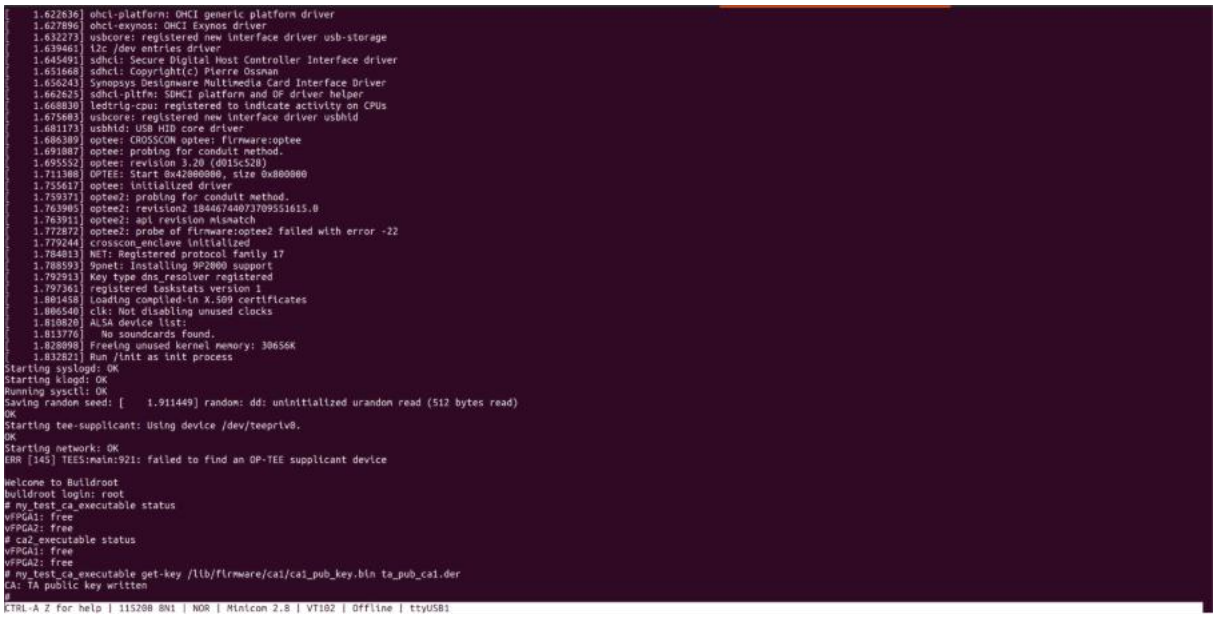


Figure 14: Usage example

Figure 14 shows that the setup is un and both the Cas can successfully check the status of the vFPGAs, while Figure 15 shows a successful client registration.

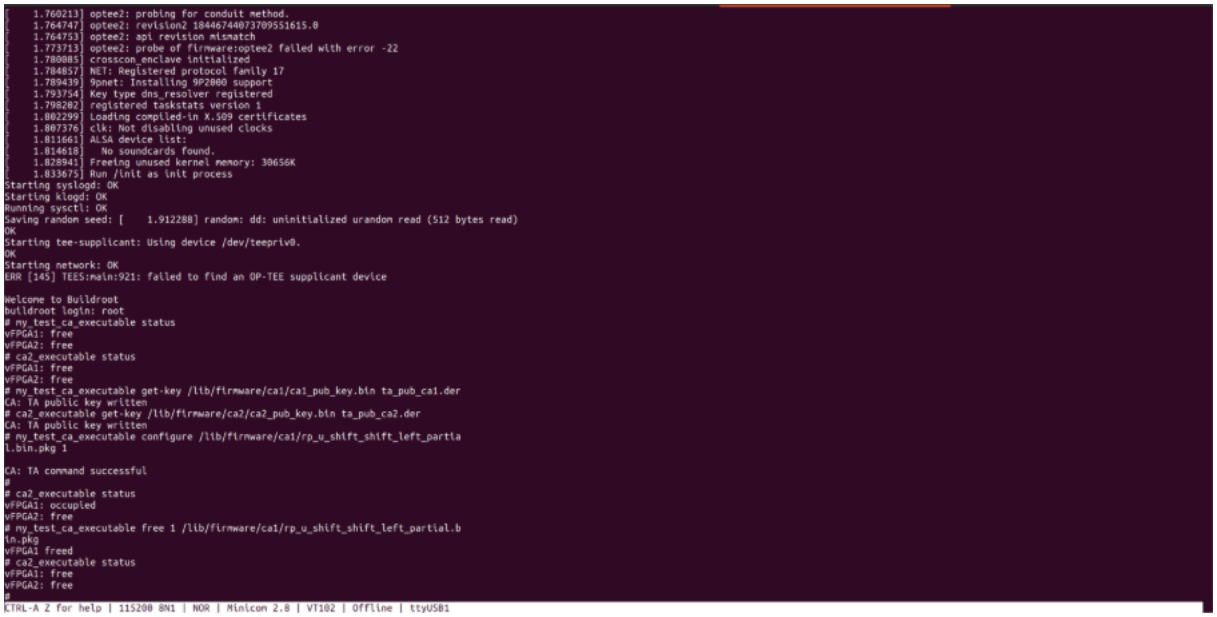


Figure 15: Usage example, successful configuration and deallocation

1. vFPGA Management

In the Linux terminal, check the vFPGA availability with the clients:

```
my_test_ca_executable status
```

```
ca2_executable status
```

2. vFPGA Client Registry

A client registry is necessary for key exchange and vFPGA configuration. The client can use the following command with with its RSA public key to do the registration and key-exchange with the TA.

```
my_test_ca_executable get-key /lib/firmware/ca1/ca1_pub_key.bin ta_pub_ca1.der
```

```
ca2_executable get-key /lib/firmware/ca2/ca2_pub_key.bin ta_pub_ca2.der
```

3. vFPGA Configuration

Configuration is done with the client package. The pacakge is created by the client and includes the encrypted bitstream along with metadata. The client also passes the vFPGA identifier for configuration.

```
my_test_ca_executable configure /lib/firmware/ca1/rp_u_shift_shift_left_partial.bin.pkg 1
```

The Linux terminal should return a successful operation, as well as more internal prints in the OpTEE terminal.

Along with the acknowledgement from the TA, successful bitstream configuration is observable through the LED interaction (patterns encoded in the bitstream). The bitstream configured in the previous step performs a Shift left pattern on the LEDS: [3,2,1,0].

At this stage, the vFPGA status can be checked again, as illustrated in the screenshots in Figure 14 and Figure 15.

4. vFPGA Deallocation

To free the CA runs the following command with the vFPGA identifier and a package for verification.

```
my_test_ca_executable free 1 /lib/firmware/ca1/rp_u_shift_shift_left_partial.bin.pkg
```

Once the vFPGA is freed, the status of the LEDs [3,2,1,0] also changes. The left shift in LEDs would change back to the default right shift in LEDs [3,2,1,0].

The other client has a partial bitstream that interacts with LEDs [7,6,5,4]

User modifications

In this section, customizations that can be done to setup based on user requirements are discussed.

1. Hypervisor modifications

- Changes to the Hypervisor configuration can be done by modifying the config.c file described in Section 3.1.1.

2. OPTEE Modifications

- The TA and pTA can be modified from the source code in optee-os folder.

3. Client Modifications

- Clients can be added/removed/modified in the host_apps folder.
- Clients are built as buildroot package, so there needs to be a rebuild done if more features are to be added to the CA.

4. Bitstream modifications.

- The secure FPGA provisioning implementation is agnostic to specific bitstream design.
- First, the top design file needs to be included in generating the BOOT.BIN (Step 2 in Installation).
- The partial bitstreams for the clients need to be packaged with builldroot by copying to the location:

```
$ROOT/buildroot/board/zcu102/overlay/
```

- The partial bitstreams need to be provided as an encrypted package. The repository has a folder client-side-helper-scirpts to help interested users with the encryption and header creation.
- To share the public key of the CA, it must first be converted to a raw blob using the following command:

```
python make_raw_blob.py in_key.[der|pem] out.bin
```

- To create the encrypted package for sharing, the following command needs to be executed:

```
pack.sh  <partial_bitstream.bit>  <output.pkg>  [uuid_hex]
```

Here, ca_sign_priv.pem is RSA-2048 private key of the CA and pta_pub.pem: RSA-2048 Public key of the TA.

3.4.6 Network Anomaly Detection

The CROSSCON network anomaly detection service is implemented based on the Atos Eviden's R&D asset, called LADS. Refer to D3.3 [3] for more details. The main challenge was to adapt its Deep Learning (DL) models and telemetry generation to the small form factor IoT platforms such as Raspberry Pi4B used in the project, as well as the L-ADS's software stack adapted to operate on ARM processors. As a results, two main components have been extended to implement the anomaly detection service – the *Telemetry* engine and the *Brain* module. The telemetry engine scoped to network protocols OSI layers 2, 3 and 4, while the Brain module to an optimised (quantized) DL model to fit small RAM and executions on less demanding CPU processors.

3.4.6.1 Deployment architecture

The reference deployment architecture is shown in Figure 16. It is based on the VirtIO-net standard which is supported by the CROSSCON hypervisor. The architecture enables all device's traffic to be visible to the anomaly detection service, whereas no traffic from the normal world applications on the device should bypass the detection service visibility. The anomaly detection service with its DL models shall be run in an isolated VM from the normal application world based on the CROSSCON hypervisor support for such architectures.

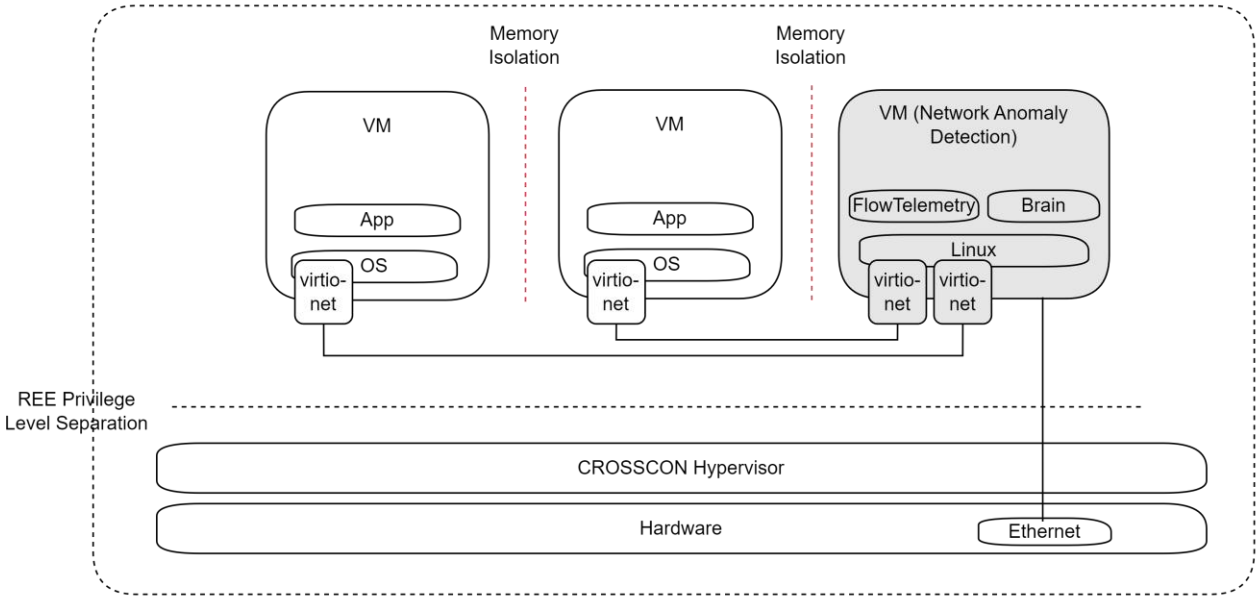


Figure 16: Network Anomaly Detection Reference Deployment Architecture

3.4.6.2 System requirements and operational capacity

The system requirements for the anomaly detection system are a Linux distro and a Docker installation. Given the deployment architecture shown in Figure 16, the minimum and recommended hardware requirements for the VM that hosts the service are defined. In addition, the operational capacity of the anomaly detection service are specified for the minimum and recommended resources.

Table 7: Network Anomaly Detection Requirements and Capacity of Operation

Requirements	Minimum	Recommended
RAM	1 GB	1.5 GB
Num. of CPUs	1 (1.8GHz, Cortex A72)	2 (2.4GHz, Cortex A76)
HDD	10 GB	20 GB
Packets per second telemetry	500 pps	1500 pps

Requirements	Minimum	Recommended
Flows/decisions per second (fps)	100 fps	200-300 fps
Number of DL models (Scope: OSI layers 2, 3, 4.)	1 model	2-3 models

3.4.6.3 Setup and Configuration

As part of the core setup of the anomaly detection, it is necessary to set the network interfaces. The name of the interfaces is added to the field `NetworkInterface` in the *config.properties* file of the Telemetry sensor.

```
OperationModality = online
NetworkInterface = enp0s3, enp0s8
FileName = live_traffic_telemetry.csv
OutPath = data/
```

Figure 17: Network Anomaly Detection Telemetry Basic Configuration

Figure 17 shows the basic configuration of the telemetry sensor.

- **OperationModality:** Either online or offline. Indicates if a live traffic capture is required for telemetry generation (online) or an offline traffic capture to be used (offline). If offline modality, a list of .pcap files are expected under the InputPath field.
- **NetworkInterface**: Indicates what network interfaces the system should sniff traffic in promiscuous mode. A comma separated list of network interfaces in a host environment. The field is used only in online modality, ignored otherwise.
- **FileName**: Indicates under what name the sensor should store the corresponding flow telemetry. The outome is a .csv file of the telemetry.
- **InputPath**: Indicates the folder where the pcap or pcapng files are available. The sensor currently supports PCAP and PCAPNG files. The field is ignored in online modality.
- **OutPath**: Indicates under what folder the telemetry results are stored, i.e. where to store the FileName.
- **ReadIPv4, ReadIPv6, ReadNonIPProtocols**: Each field is either true or false. Activates telemetry for OSI layers 2, 3, and 4. If ReadNonIPProtocols is set to false, the sensor will generate telemetry for IPv4 and/or IPv6 protocols of OSI laters 3 and 4. The three fields are set to true by default and do not need to be specified in the config file, unless a different setup is needed.

Figure 18 shows the Brain configuration for traffic prediction based on pre-trained models. This config file contains the following fields:

- `version_name`: Specifies the version name.
- `clean`: A Boolean field that cleans all the outputs if enabled.
- `report`: If True, the Brain will generate a report of the anomalies detected, such as important network features for anomaly, non-conformity of features, and value deviations from training.
- `min_listening`: A numerical field that specifies the number of minutes for capturing traffic during the TRAINING phase. In PREDICTION modality, this field is ignored.
- `ms_cooldown`: A numerical field that specifies the number of milliseconds for cooldown if there are no flows arrived.
- `chunksize`: A numerical field that specifies the chunk size in PREDICTION mode.
- `mode`: This field could be TRAINING or PREDICTION.
- `models`: Specifies which models will be loaded during the prediction phase.

```
version_name = crosscon
clean = false
report = true
```

```
min_listening = 600
ms_cooldown = 500
batch_size = 64
epochs = 200
hidden_layers= default
mode = TRAINING
logging_level = info
train_contexts = [
  {"context": "default"}
    ]
models = [
    {"model": "RPi4b", "parallelism": 1, "context": "default",
"threshold":10}
    ]
```

Figure 18: Network Anomaly Detection Brain Basic Configuration

Operation. The anomaly detection service offers two main operations:

- *Training*: By default, upon the first run, the service starts training a Deep Learning model on legitimate traffic as seen on the specified network interfaces. It is necessary to specify the amount of time needed to train, e.g. 600 min. Once trained, the service automatically switches to a prediction modality. The threshold is automatically set according to the results of training.
- *Prediction*: The service is in a monitoring mode on the same network interfaces used for training. All flows of incoming traffic are predicted if legitimate of anomalous. These predictions are performed real-time and stored in the log file of the Brain located under the folder `/outputs/`. The log file with the predictions can be communicated to a dedicated remote platform, say a SIEM entry point, using the RSYSLOG tool.

3.5 CROSSCON SoC

CROSSCON SoC is a SoC design developed as part of the CROSSCON project to show how hardware isolation mechanisms developed during the project can be leveraged, together with the CROSSCON Hypervisor, to obtain a secure RISC-V execution environment for mixed-criticality IoT devices that require strong software (SW) and hardware (HW) isolation.

The chapter startes with the instructions on how to set-up a CROSSCON SoC on Arty-A7 100T [10] board and run an example program. Following an overview of how the examples are structured and basic parts of the SoC that developers need to be aware of.

3.5.1 Setting up the CROSSCON SoC and running a basic example

The CROSSCON SoC can be tried out by running one of the examples located in `crosscon_soc/examples` directory of CROSSCON SoC GitHub repository [11]. To run an example one must:

1. upload the CROSSCON SoC's bitstream to the Arty-A7 100T board,
2. build the example for the BA51-H core, and
3. connect a debug key to the BA51-H, upload the binary of the example and run it.

All the necessary resources are available on the CROSSCON SoC's Github repository [11]. Please check out the repository before proceeding with the following instructions.

3.5.1.1 Uploading the CROSSCON SoC bitstream

The CROSSCON SoC's bitstream can be uploaded to the Arty-A7 100T board using the OpenOCD. One can install openocd by `sudo apt install openocd` on Debian based systems. Note that you need use openocd v0.12 or newer.

Connect the Arty-A7 board with your PC using a USB-to-micro-USB cable via the USB-JTAG port. See the Arty-A7 reference manual [12] for the overview of the Arty-A7 board.

Upload the bitstream by going to the `crosscon_soc/scripts/upload_bits` folder and running:

```
./upload.sh ../../bits/crosscon_soc_a7_v0.2.bit
```

If the bitstream was uploaded successfully, you should see an output similar to the following:

```
Open On-Chip Debugger 0.12.0
Licensed under GNU GPL v2
For bug reports, read http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "jtag". To override use 'transport select '.
jtagspi_program
Info : clock speed 5000 kHz
Info : JTAG tap: xc7.tap tap/device found: 0x13631093 (mfg: 0x049 (Xilinx), part: 0x3631, ver: 0x1)
```

and the "DONE" LED of the Arty-A7 board should be illuminated green.

3.5.1.2 Setup the RISC-V toolchain

Checkout the RISC-V toolcahin [13] repository and compile it:

```
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
mkdir install_dir
./configure --prefix=$(pwd)/install_dir --with-arch=rv32imafc_zicsr_zifencei_zca --with-abi=ilp32f --with-target-cflags=-g --with-target-cxxflags=-g '--with-multilib-generator=rv32imac-ilp32--; rv32imafc_zca-ilp32--; rv32imac_zicsr_zifencei_zca-ilp32--;'
make
```

Note that the toolchain will use the rv32imacf_zca_zcb_zcf ARCH and the ilp32f ABI by default. When compiled, the toolchain will be available in the `riscv-gnu-toolchain/install_dir` directory. Set the RISCV environment variable to bin directory of the compiled toolchain where the riscv32-unknown-elf-* commands are available:

```
export RISCV=/path/to/riscv-gnu-toolchain/install_dir/bin
```

3.5.1.3 Build the example program

Go to the example's directory `crosscon_soc/examples/` and run

```
./setup.sh
```

to setup the source files. You can compile the example by running

```
./build.sh
```

If the example was compiled successfully, the last line of the output should be `# Done`.

3.5.1.4 Upload and run the binary

In order to run the example, the binary needs to be upload to the CROSSCON SoC's memory using a JTAG compatible debug key.

First, connect the debug key to the Arity-A7's JD port, as shown by Figure 19, and to your PC via a USB-to-micro-USB cable.

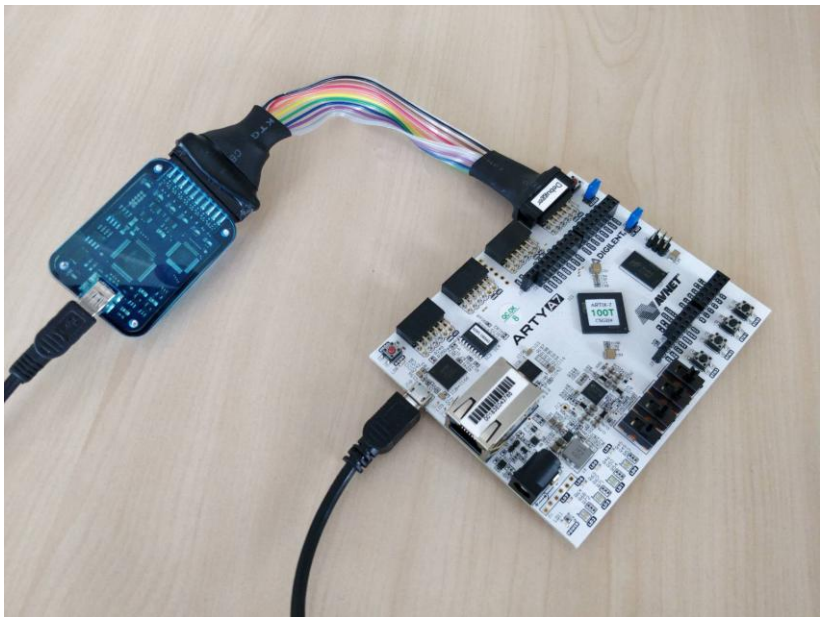


Figure 19: Test Setup

For the following instructions, assuming that Beyond's debug key [14] is being used with the appropriate cable that has the following pinout on the side that connects to the Arty-A7's JD Pmod port:

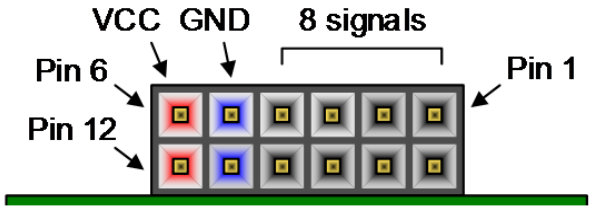


Figure 20: Arty-A7's Pmod pin layout (looking from outside of PCB edge)

Table 8: Required JTAG pin input for the Arty-A7's JD Pmod port where Input and Output types denote input/output to/from the targeted JTAG port

Pin	Signal	Type
1	TDO	Output
2		
3	TCK/SWCLK	Input
4	TX	Output
7	TDI	Input
8	TMS/SWDIO	Input
9	nTRST	Input

If you are using some other debug key, the following instructions might not work out of the box, so you'll need to do a bit of research, but the main idea is to use the OpenOCD to connect to the BA51-H using GDB over the debug key.

3.5.1.5 Run the OpenOCD

Run the OpenOCD server so that we'll be able to talk to the debug key using GDB:

```
openocd -f ../../scripts/openocd/crosscon_soc.openocd.cfg
```

The following should appear:

```
Open On-Chip Debugger 0.12.0-snapshot (2024-02-16-12:42)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : clock speed 3000 kHz
Info : TAP riscv.cpu does not have valid IDCODE (idcode=0xfffffffe)
Info : datacount=2 progbufsize=0
Warn : We won't be able to execute fence instructions on this target. Memory may not always
appear consistent. (progbufsize=0, impebreak=0)
Info : Examined RISC-V core; found 1 harts
Info :  hart 0: XLEN=32, misa=0x401411a5
Info : starting gdb server for riscv.cpu on 3333
Info : Listening on port 3333 for gdb connections
Ready for Remote Connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

3.5.1.6 Listening to UART

In the case of Beyond's debug key, the UART port is exposed as a device file `/dev/ttyUSBx`.

We'll use tio command line tool to interact with it. You can install the tio by running

```
sudo apt install tio
```

You can list the available /dev/ttyUSBx devices by running:

```
tio --list
```

You should see something similar to

```
Device              TID      Uptime [s] Driver       Description
------------------- ---- ------------- ------------- -------------------------
/dev/ttyUSB1        aOSQ     9969.776 ftdi_sio      Digilent USB Device
/dev/ttyUSB3        CJFY     9965.260 ftdi_sio      Debug Key
```

In the case of Beyond's debug key, the relevant device is the one with the `Debug Key` description.

Once you know which device file is the right one, run:

```
./listen_on_uart.sh /dev/ttyUSBx
```

where `/dev/ttyUSBx` is replaced with the actual device file path.

Something like this should be visible:

```
[09:41:50.690] tio v3.3
[09:41:50.690] Press ctrl-t q to quit
[09:41:50.699] Connected to /dev/ttyUSB1
```

3.5.1.7 Upload and run the program using GDB

Run

```
./run_gdb.sh
```

Now the program was executed. If you setup everything correctly, you should something similar to the following output on UART. Note that, because the use of UART is not synchronized, guest VM outputs might overlap.

...

```
  _____ _____   ____   _____ _____  _____ ____  _  _
 / ____|  __ \ / __ \ / ____/ ____|/ ____/ __ \| \ | |
| |    | |__) | |  | | (___| (___ | |   | |  | |  \| |
| |    |  _  /| |  | |\___ \\___ \| |   | |  | | . ` |
| |____| | \ \| |__| |____) |___) | |___| |__| | |\  |
 \_____|_|  \_\\____/|_____/_____/ \_____\____/|_| \_|

 _   _                    _
| | | |                  (_)
| |__| |_  _ _ __   ___ _ ____   ___ ___  ___ _ __
|  __  | | | | '_ \ / _ \ '__\ \ / / / __|/ _ \| '__|
| |  | | |_| | |_) |  __/ |   \ V /| \__ \ (_) | |
|_|  |_|\__, | .__/ \___|_|    \_/ |_|___/\___/|_|
         __/ | |
        |___/|_| 17:59:30
```

CROSSCONHYP INFO: Initializing VM 0
CROSSCONHYP WARNING: trying to flush caches but the operation is not defined for this platform
CROSSCONHYP INFO: VM 0 adding MMIO region, VA: 0x10700000 size: 0xfffff mapped at 0x10700000
CROSSCONHYP INFO: VM 0 adding MMIO region, VA: 0x24000000 size: 0xfffff mapped at 0x24000000
CROSSCONHYP INFO: VM 0 is sdGPOS (normal VM)
CROSSCONHYP INFO: Initializing VM 1
CROSSCONHYP WARNING: trying to flush caches but the operation is not defined for this platform
CROSSCONHYP INFO: VM 1 adding MMIO region, VA: 0x10700000 size: 0xfffff mapped at 0x10700000
CROSSCONHYP INFO: VM 1 adding MMIO region, VA: 0x24000000 size: 0xfffff mapped at 0x24000000
CROSSCONHYP INFO: VM 1 is sdGPOS (normal VM)
[guest 0] Bare-metal test guest
additional data = 'Additional data of the guest vm 0'
msg = 'Test message of the guest vm 0. Test message of the guest vm 0. Test message of the guest vm 0'
[guest 1] Bare-metal test guest
additional data = 'Additional data of the guest vm 1'
msg = 'Test message of the guest vm 1. Test message of the guest vm 1. Test message of the guest vm 1'
[guest 0] Encrypting ...
[guest 1] Encrypting ...
[guest 0] Done passing initial parameters.
[guest 1] Done passing initial parameters.
[guest 0] Waiting for output data ...
[guest 0] data_out[0] (hex) = 0x0b8b5e7d9b287a29a845d5cd0c30c283
[guest 1] Waiting for output data ...
[guest 1] data_out[0] (hex) = 0x67a7fc2a295ad1606ba14a0abafd7bcc
[guest 0] Waiting for output data ...
[guest 0] data_out[1] (hex) = 0xadf362d09c80e22e7726834c0b256822
[guest 1] Waiting for output data ...
[guest 1] data_out[1] (hex) = 0x05d237e2ec08d8ea5fbdedb80d2983e2
[guest 0] Waiting for output data ...
[guest 0] data_out[2] (hex) = 0x4300f37ba965fbd3ed154aec5f2e2845
[guest 1] Waiting for output data ...

```
[guest 1] data_out[2] (hex) = 0x7dd2309a742e5e45fc4deeb84c2f8268
[guest 0] Waiting for output data ...
[guest 0] data_out[3] (hex) = 0x144ec7e22963a7946c9f95f1480722e4
[guest 1] Waiting for output data ...
[guest 1] data_out[3] (hex) = 0xec7ba1315eaf7ed602979c27255747c2
[guest 0] Waiting for output data ...
[guest 0] data_out[4] (hex) = 0x7e5b397a394b79151d0ee3fd762e1cf2
[guest 1] Waiting for output data ...
[guest 1] data_out[4] (hex) = 0xf534537bbdc07027b692e0107f8c04fd
[guest 0] Waiting for output data ...
[guest 0] data_out[5] (hex) = 0x1a947dd9231d6af00fa5279aa9967b7c
[guest 1] Waiting for output data ...
[guest 1] data_out[5] (hex) = 0xe5455efee00662ae29097499c58e1c9d
[guest 0] tag (hex) = 0x3dda769317e930acb2151044f9ca26c8
[guest 1] tag (hex) = 0xa84a704cdf8b104bef0431b975faf694

[guest 0] Decrypting ...
[guest 1] Decrypting ...
[guest 0] Done passing initial parameters.
[guest 1] Done passing initial parameters.
[guest 0] Waiting for output data ...
[guest 0] data_out[0] (hex) = 0x54657374206d657373616765206f6620
[guest 1] Waiting for output data ...
[guest 1] data_out[0] (hex) = 0x54657374206d657373616765206f6620
[guest 0] Waiting for output data ...
[guest 0] data_out[1] (hex) = 0x746865206775657374207766d20302e20
[guest 1] Waiting for output data ...
[guest 1] data_out[1] (hex) = 0x746865206775657374207766d20312e20
[guest 0] Waiting for output data ...
[guest 0] data_out[2] (hex) = 0x54657374206d657373616765206f6620
[guest 1] Waiting for output data ...
[guest 1] data_out[2] (hex) = 0x54657374206d657373616765206f6620
[guest 0] Waiting for output data ...
[guest 0] data_out[3] (hex) = 0x746865206775657374207766d20302e20
[guest 1] Waiting for output data ...
[guest 1] data_out[3] (hex) = 0x746865206775657374207766d20312e20
[guest 0] Waiting for output data ...
[guest 0] data_out[4] (hex) = 0x54657374206d657373616765206f6620
[guest 1] Waiting for output data ...
[guest 1] data_out[4] (hex) = 0x54657374206d657373616765206f6620
[guest 0] Waiting for output data ...
[guest 0] data_out[5] (hex) = 0x746865206775657374207766d20307b7c
[guest 1] Waiting for output data ...
[guest 1] data_out[5] (hex) = 0x746865206775657374207766d20311c9d
[guest 0] tag (hex) = 0x3dda769317e930acb2151044f9ca26c8
[guest 1] tag (hex) = 0xa84a704cdf8b104bef0431b975faf694
[guest 0] decrypted msg = 'Test message of the guest vm 0. Test message of the guest vm 0. Test
message of the guest vm 0'
[guest 0] Done.
[guest 1] decrypted msg = 'Test message of the guest vm 1. Test message of the guest vm 1. Test
message of the guest vm 1'
[guest 1] Done.
```

3.5.2 Using the CORSSCON SoC

The best way to see how the CROSSCON SoC can be used is through the examples available on GitHub repository [11]. In the following section, the overview of the basic parts of a typical example that a developer needs to be aware of is provided.

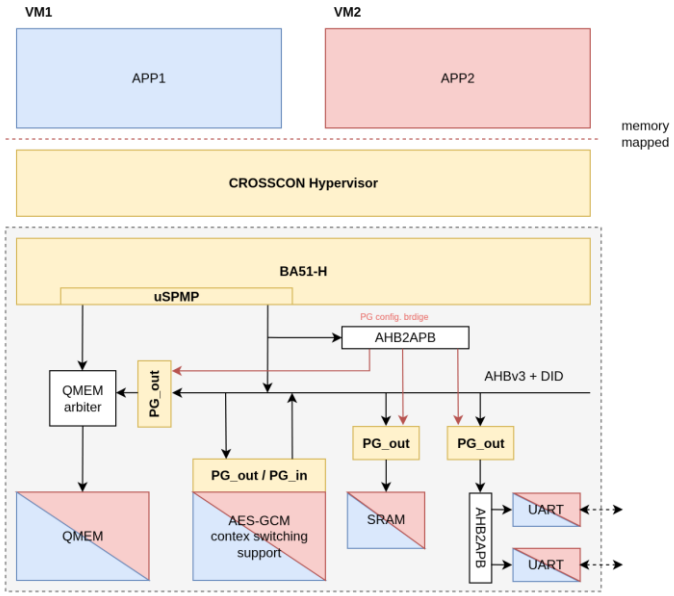


Figure 21: Crosscon SoC Architecture

Figure 21 shows a high-level architecture of the current version of the CROSSCON SoC. The CROSSCON SoC is built around the Beyond's BA51-H RISC-V core and includes all the necessary infrastructure that one needs to develop embedded applications (e.g. JTAG support and UART). It provides 512KB of main memory (QMEM), with an additional 512B of SRAM memory protected by PG in lock-release arbitration mode. Furthermore, it provides an AES-GCM accelerator with context switching support and two UART modules that can be assigned into separate domains by PG in restricted address space mode.

By leveraging CROSSCON Hypervisor, BA51-H and PG, one can partition the SoC resources between several VMs running on the CROSSCON Hypervisor, where each VM is running in its own domain. Due to the limited resources of the FPGA available on Arty-A7 100T board, the CROSSCON SoC currently supports only two domains, which means that only two VMs can be used on the Arty-A7 100T board. The software running in each of the domains can be found in `baremetal-guest` and `baremetal-guest2` directories in each of the examples that use the CROSSCON Hypervisor.

Address space: All the hardware modules external to the BA51-H processing core are connected between each other over the AHB interconnect with the following address space.

Table 9 shows the address space visible to all the master modules connected to the interconnect, including to the software running in individual VMs.

Table 9: Address space visible to all masters connected to te interconnect

Address range	Module
0x00000000 - 0x0FFFFFFF	QMEM (512KB)
0x10700000 - 0x107FFFFF	UART
0x20000000 - 0x20FFFFFF	SRAM (512B)
0x21000000 - 0x21FFFFFF	SRAM's PG lock-release interface
0x24000000 - 0x24FFFFFF	AES-GCM accelerator

Table 10 shows the address space visible only to the BA51-H core. The address space includes the configuration registers of all the PG modules available on the SoC. The PG modules are usually configured by the CROSSCON Hypervisor and thus its registers are not available to individual VMs.

Table 10: The address space visible only to BA51-H

Address range	Module
0x23100000 - 0x231FFFFF	BA51-H DID register - Drives HDID signal on the interconnect.
0x23200000 - 0x232FFFFF	QMEM's PG configuration interface
0x23300000 - 0x233FFFFF	APB subsystem's PG configuration interface
0x23400000 - 0x234FFFFF	SRAM's PG configuration interface

Note that the address ranges assigned to different modules are larger than the actual ranges used by the modules. The modules only use the lower bits when resolving which register is addressed. Thus, multiple addresses in the address range can map to the same register.

C header file: In order to simplify the use of the CROSSCON SoC, there is a C header file, named 'crosscon_soc.h', that includes all the definition of the CROSSCON SoC specific registers. The C header file can be found in `src` directory of the CROSSCON's GitHub repository [11] and is usually included into each example when `setup.sh` script is called.

Interconnect isolation: The CROSSCON SoC's interconnect leverages PG to obtain transfer isolation between domains / guest VM's and allows modules to be shared between domains in a safe manner. Table 11 lists the PG operation modes used for different modules connected to the interconnect:

Table 11: PG operation modes per protected hardware module

Module	PG operation mode
QMEM	Restricted address range mode with 8 address range entries
SRAM	Time-sharing with reset and lock-release arbitration mode
APB subsystem	Restricted address range mode with 8 address range entries
AES-GCM	Time-sharing with context switching

AES-GCM accelerator software interface: AES-GCM accelerator is a hardware module that allows one to encrypt or decrypt data using the AES-GCM accelerator. The accelerator supports context switching and can thus be used by multiple VMs at the same time without software arbitration. An example of how to use the AES-GCM example is available in the `cs_hypervisor_with_aes_gcm_example` of the GitHub repository [15] under `baremetal-guest/src/main.c` file. The code that encrypts or decrypts the data follows the workflow shoved below, which is the suggested way for using the accelerator:

```
void aes_gcm_cipher(
   // Input parameters
   bool encrypt, // True for encryption; false for decryption.
   uint8_t key[32],
   uint8_t iv[12]
   uint32_t ad_len,
   uint8_t *ad_data,
   uint32_t d_len,
   uint8_t *data_in,
  // Output parameters
  uint8_t *data_out,
  uint8_t *tag) {
...
//
// Pass initial parameters.
//

aes_mm_status_reg[AES_GCM_ENCRYPT_REG] = (uint32_t) encrypt;

aes_mm_status_reg[AES_GCM_KEYL_REG] = 0x2;

for (int i = 0; i < 8; i++) {
   aes_mm_status_reg[AES_GCM_KEY_REG + i] = *((uint32_t *) (key + (4*i)));
```

```
}
...
printf("[guest 0] Done passing initial parameters.\n");

//
// Pass additional data blocks.
//

for (int j = 0; j < ad_len; j += 16) {
  ...

    memcpy((void*) data_in_block_buf, (void*) (ad_data + j), csize);

    aes_mm_status_reg[AES_GCM_STATE_REG] = (uint32_t) (0x1) << AES_GCM_STATE_IN_D_RDY_FLAG;

    while ((aes_mm_status_reg[AES_GCM_STATE_REG] & (0x1 << AES_GCM_STATE_IN_D_RDY_FLAG))) {
        printf("[guest 0] Waiting for additional data to be processed ...\n");
    }

}

//
// Pass data blocks.
//

for (int j = 0; j < d_len;  j += 16) {

    ...
    memcpy((void*) data_in_block_buf, (void*) (data_in + j), csize);

    aes_mm_status_reg[AES_GCM_STATE_REG] = (uint32_t) (0x1) << AES_GCM_STATE_IN_D_RDY_FLAG;

    while (!(aes_mm_status_reg[AES_GCM_STATE_REG] & (0x1 << AES_GCM_STATE_OUT_D_RDY_FLAG))) {
        printf("[guest 0] Waiting for output data ...\n");
    }

    ...
    memcpy((void*) (data_out + j), (void*) data_out_block_buf, csize);
}

//
// Wait for tag.
//

while (!(aes_mm_status_reg[AES_GCM_STATE_REG] & (0x1 << AES_GCM_STATE_TAG_RDY_FLAG))) {
    printf("[guest 0] Waiting for tag ...\n");
}

...
memcpy(tag, ((uint8_t*) data_out_block_buf + 16), 16);
```

4 Conclusion

This deliverable consolidates the installation and usage procedures for all components of the CROSSCON stack, ensuring that the security architecture can be effectively deployed in heterogeneous environments.
The guidelines provide detailed references for the configuration of the CROSSCON Hypervisor, the Bare-Metal TEEs, the TEE Toolchain, and trusted applications, covering both software and hardware implementations across Arm and RISC-V platforms.

The main outcomes achieved through this work include:

- A detailed installation and configuration process for all CROSSCON components.
- A validated set of usage models and deployment environments for different classes of IoT devices.
- Demonstrated interoperability between CROSSCON's hypervisor, TEEs, and trusted applications across multiple architectures.
- A set of tested demo scenarios showcasing dynamic virtualization, per-VM TEE support, and secure FPGA provisioning.

These results position the CROSSCON stack as a foundational enabler for secure, scalable, and trustworthy IoT systems. Future work will focus on optimizing performance across heterogeneous devices, strengthening automation in deployment workflows, and expanding validation through large-scale demonstrators. The document thus represents both a technical reference and a roadmap for continued integration, validation, and exploitation of CROSSCON's open security technologies within industrial and research contexts.

References

[1] CROSSCON, A. García, "D1.5 Final Requirements and Technical Specifications," 2024.

[2] CROSSCON, N. Singh, "D2.3 CROSSCON Open Specification (Final)" 2025.

[3] CROSSCON, M. Götz, "D3.3 CROSSCON Open Security Stack Documentation - Final," 2025.

[4] CROSSCON, A. Tacchella, "D3.4 CROSSCON Open Security Stack - Final," 2025.

[5] CROSSCON, M. Götz, "D4.3 CROSSCON Extensions to Domain Specific Hardware Architectures Documentation - Final," 2025.

[6] CROSSCON, Z. Putrle, "D4.4 CROSSCON Extension Primitives to Domain Specific Hardware Architectures - Final," 2025.

[7] [7] https://gitlab.arm.com/research/ietf-suit/suit-parser

[8] [8] https://github.com/angr/angr

[9] https://dst.trust.informatik.tu-darmstadt.de/drive/d/s/14PasofbNNYT7NHzMbB1KY2iDN2qr1XX/KLWTtjN_RO1Y0hiMZenoEh6rYj1lms4O-z7lg0JsEegw

[10] https://digilent.com/shop/arty-a7-100t-artix-7-fpga-development-board/

[11] https://github.com/crosscon/crosscon_soc

[12] https://digilent.com/reference/reference/programmable-logic/arty-a7/reference-manual

[13] https://github.com/riscv-collab/riscv-gnu-toolchain

[14] https://www.beyondsemi.com/beyond-debug-key/

[15] https://github.com/bao-project/bao-hypervisor

[16] https://github.com/crosscon/baremetal-tee/tree/main/nonMPU-version

[17] https://www.ti.com/tool/CCSTUDIO

[18] https://www.ti.com/tool/MSP-EXP430F5529LP

[19] https://www.iar.com/embedded-development-tools/iar-embedded-workbench#containerblock_3096

[20] https://github.com/crosscon/baremetal-tee/tree/main/nonMPU-version#trusted-application-apis

[21] https://github.com/crosscon/baremetal-tee/tree/main/MPU-version#global-platform-core-api-available-to-trusted-applications-tas

[22] https://github.com/crosscon/baremetal-tee/tree/main/MPU-version#fortifying-an-application

[23] https://github.com/crosscon/secure_update_consumer

[24] https://github.com/crosscon/secure_update_infrastructure

[25] https://github.com/anchore/grype

[26] crosscon/ZK-PUF-Zephyr-Demo

[27] crosscon/UC1.1-Manifest

[28] https://euc-word-edit.officeapps.live.com/we/ENROLLMENT_APP

[29] crosscon/crosscon_puf_authentication

[30] https://github.com/crosscon/context-based-auth-remote/

[31] https://github.com/crosscon/context-based-auth-crosscon-demo

[32] https://github.com/crosscon/remote-attestation-remote-verifier

[33] https://github.com/crosscon/CROSSCON-Hypervisor-and-TEE-Isolation-Demos

[34] https://github.com/crosscon/UC5-IP_Protection_for_Secure_Multi-Tenancy_on_FPGA

[35] https://dst.trust.informatik.tu-darmstadt.de/drive/d/s/14PVNjmZJKDlmeWfA0xJqPPCGNpRVx4C/6_8JlNLfIG4INMEVYGvh_6X0CfvIdQm_-SrsA9VX2eQw

[36] https://dst.trust.informatik.tu-darmstadt.de/drive/d/s/14PVQyO0IDiJkzf3KD2DEd2qRKaDIDVC/RYZsnljZ1vaD8VrB8fwuV9nYnFNFeVmF-KLyAte_2eQw

[37] https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/ek-u1-zcu102-g.html

[38] CROSSCON, A. García, "D5.6 Security Testing and Validation Results of the CROSSCON Stack in Use Cases - Final Report" 2025.

[39] https://github.com/crosscon/CROSSCON-Hypervisor-and-TEE-Isolation-Demos/tree/master?tab=readme-ov-file#demo-5

[40] https://github.com/crosscon/CROSSCON-Hypervisor-and-TEE-Isolation-Demos/blob/master/README.md