Ref. Ares(2025)2575086 - 31/03/2025

Cross-platform **O**pen **S**ecurity **S**tack for **Co**nnected Device

D2.3 CROSSCON Open Specification - Final

Document Identification			
Status	Final	**Due Date**	31/03/2025
Version	1.0	**Submission Date**	31/03/2025

Related WP	WP2	**Document Reference**	D2.3
Related Deliverable(s)	D2.1	**Dissemination Level(*)**	PU
Lead Participant	TUD	**Lead Author**	Nikhilesh Singh
Contributors	TUD, UMINHO, UWU, BEYOND, UNITN	**Reviewers**	João Miguel Costa Sousa (UMINHO), Mateusz Kusiak (3mdeb)

Keywords
CROSSCON Trusted Execution Environment, CROSSCON Hypervisor, CROSSCON System on Chip, CROSSCON Formal Specification

This document is issued within the frame and for the purpose of the CROSSCON project. This project has received funding from the European Union's Horizon Europe Programme under Grant Agreement No.101070537. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.
The dissemination of this document reflects only the author's view, and the European Commission is not responsible for any use that may be made of the information it contains. This deliverable is subject to final acceptance by the European Commission.
This document and its content are the property of the CROSSCON Consortium. The content of all or parts of this document can be used and distributed provided that the CROSSCON project and the document are properly referenced.
Each CROSSCON Partner may use this document in conformity with the CROSSCON Consortium Grant Agreement provisions.

(*) Dissemination level: (PU) Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page). (SEN) Sensitive, limited under the conditions of the Grant Agreement. (Classified EU-R) EU RESTRICTED under the Commission Decision No2015/444. (Classified EU-C) EU CONFIDENTIAL under the Commission Decision No2015/444. (Classified EU-S) EU SECRET under the Commission Decision No2015/444.

Document Information

List of contributors	
Name	**Partner**
Marco Roveri	UNITN
Alberto Tacchella	UNITN
Michele Grisafi	UNITN
Emanuele Beozzo	UNITN
Žiga Putrle	BEYOND
David Cerdeira	UMINHO
João Sousa	UMINHO
Tymoteusz Burak	UWU
Fabian Schmitt	UWU
Christoph Sendner	UWU
Nikhilesh Singh	TUD

Document history			
Ver.	**Date**	**Change editors**	**Changes**
0.1	10/12/2024	M. Roveri (UNITN)	Initial draft structure.
0.2	21/01/2025	D. Cerdeira (UMINHO)	Add D2.1 Content for UMinho relevant sections.
0.3	31/01/2025	J. Sousa (UMINHO)	Changes in "Overview", "High-level Architecture" and "Description of Architecture Components" sections.
0.4	07/02/2025	A. Tacchella, M. Grisafi, E. Beozzo, M. Roveri (UNITN)	Added content for Unitn relevant sections.
0.5	11/03/2025	Z. Purtle (BEYOND)	Finalizing the content of Hardware/Software Co-Design chapter.
0.6	11/03/2025	N. Singh (TUD)	Finalizing the contents of TUD relevant sections.
0.7	11/03/2025	N. Singh (TUD)	Finalizing the layout of API descriptions in all sections.
0.8	11/03/2025	N. Singh (TUD)	Added Section 1.3.1 listing the change w.r.t. D2.1.
0.9	14/03/2025	T. Burak, F. Schmitt, C. Sendner (UWU)	Added content for UWU relevant sections.
0.10	24/03/2025	Z. Purtle (BEYOND)	Addressed the review comments of Chapter 7 (Hardware/Software Co-Design).
0.11	24/03/2025	A. Tacchella, M. Grisafi, E. Beozzo, and M. Roveri (UNITN)	Addressed the respective review comments.
0.12	24/03/2025	T. Burak, F. Schmitt, C. Sendner (UWU)	Addressed the respective review comments.
0.13	27/03/2025	N. Singh (TUD)	Addressed the review comments of various sections and polished content.
0.14	28/03/2025	N. Singh (TUD)	Addressed the comments for QA in various sections.
0.15	28/03/2025	Juan Andrés Alonso (ATOS)	Final version based on quality control.
1.0	31/03/2025	H. Koshutanski (ATOS)	Final version submitted.

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Nikhilesh Singh	27/03/2025
Quality Manager	Juan Andrés Alonso (ATOS)	28/03/2025
Project Coordinator	H. Koshutanski (ATOS)	31/03/2025

Table of Contents

Document Information 2
Table of Contents 4
List of Tables 7
List of Figures 8
List of Abbreviations 9
Executive Summary 11
1 Introduction 12
1.1 Purpose of the Document 12
1.2 Relation to Other Project Work 12
1.3 Structure of the Document 12
1.3.1 Changes w.r.t. D2.1 12
2 Overview 13
2.1 Adversary Model 13
2.2 The Architectural Impact of Requirements 13
2.3 Assumptions and Constraints 14
2.3.1 Definitions of Key Concepts and Terminology 14
2.3.1.1 Trusted Execution Environment (TEE) 14
2.3.1.2 Rich Execution Environment (REE) 15
2.3.2 General Objectives 15
2.3.3 General Assumptions 16
2.3.4 General Constraints 16
3 High-level Architecture 17
3.1 Unified Abstract Model 17
3.2 System Stack Components 18
3.3 Architecture Description 18
3.3.1 Architectures 19
3.3.2 Hardware Security Primitives 19
3.3.3 Instantiation Options 22
3.3.4 Software Only Isolation Environment 22
3.3.5 Basic Memory Isolation Environment 23
3.3.6 Virtualization-less environment with TEE 23
3.3.7 TEE-less environment with Virtualization 23
3.3.8 Environment with TEE and Virtualization 24
3.3.9 Environment with Virtualization and FPGA 24
4 Description of Architecture Components 26
4.1 CROSSCON Stack Components 26
4.1.1 Firmware 26
4.1.2 Trusted Execution Environment (TEE) 26
4.1.3 CROSSCON Baremetal TEE 27
4.1.4 CROSSCON Hypervisor 28
4.1.5 Virtual Machines (VM) 28
4.1.6 Confidential VMs 29
4.1.6.1 Confidential VM Compatibility 29
4.2 CROSSCON Trusted Services 30
4.2.1 Context-Based Authentication 30
4.2.2 PUF-Based Authentication for IoT Devices 30

4.2.3 Trusted Application for Secure FPGA Partitioning ... 31
4.2.4 Control Flow Integrity ... 31
5 Specification of Proposed Architecture ... 32
5.1 CROSSCON Hypervisor ... 32
5.2 CROSCCON Hypervisor VM Configuration Interface ... 33
5.2.1 VM Configuration ... 34
5.2.2 Guest Image ... 36
5.2.3 Virtual Machine Configuration ... 37
5.2.3.1 Number of vCPUs ... 38
5.2.3.2 Memory Regions ... 38
5.2.3.3 Inter-Partition Communication (IPC) ... 39
5.2.3.4 Devices ... 40
5.2.3.5 Architectural-Specific Configurations ... 40
5.2.4 CPU Affinity ... 42
5.2.5 Coloring ... 43
5.2.6 Shared Memory Configuration ... 44
5.2.7 Configuration File Location ... 44
5.2.8 Static Partitioning Example ... 45
5.2.9 Virtual TZ TEE Example ... 48
5.2.10 Virtual SGX Example ... 52
5.2.11 Dynamic VM Example ... 53
5.2.12 Trap and Emulation ... 54
5.2.13 Hypercalls ... 55
5.3 CROSSCON Multi-VMM Support ... 55
5.4 CROSSCON Baremetal TEE ... 56
5.4.1 Baremetal nonMPU ... 57
5.4.2 Baremetal MPU ... 57
5.4.2.1 TEE Configuration ... 57
5.4.2.2 Trusted Application Configuration ... 57
5.4.2.3 Client Application Configuration ... 57
6 Specification of CROSSCON API ... 59
6.1 Hypervisor API ... 59
6.1.1 Inter Partition Communication (IPC) ... 59
6.1.2 Dynamic VM ... 62
6.1.3 Virtual TZ TEE ... 64
6.1.4 Virtual SGX TEE ... 64
6.2 Secure Storage APIs ... 64
6.3 Cryptographic operations ... 65
6.4 Additional measurements in attestation reports (besides default measurements) ... 65
6.5 The device's unique identifier ... 66
6.6 PUF ... 67
6.7 Remote attestation APIs ... 69
6.8 Context-Based Authentication APIs ... 71
6.9 Secure Update APIs ... 72
6.10 Secure FPGA Provisioning API ... 78
7 Hardware/Software Co-Design ... 81
7.1 High-level overview of CROSSCON SoC ... 81
7.2 Perimeter guard ... 81
7.2.1 Motivation ... 81
7.2.2 Perimeter guard architecture ... 82
7.2.3 The threat model ... 85

7.2.4 General consideration of PG attack surface 85
7.2.5 Configuring PG 86
7.2.6 Responding to access violation 86
7.2.7 Operation Mode: Exclusive access with external arbitration mode 86
7.2.8 Operation mode: Restricted address space with external arbitration mode 87
7.2.9 Operation mode: Time-sharing with reset and lock-release arbitration mode 89
7.2.10 Operation mode: Time-sharing with context switching mode 91
7.2.11 Integrating PG with the CROSSCON Stack 93
7.2.12 Comparison with other solutions 94
8 Conclusion 95
Bibliography 96

List of Tables

Table 1: Features of platforms selected for CROSSCON. *20*
Table 2: Configuration registers used in PG's Exclusive access with external arbitration mode. *87*
Table 3: Configuration registers available in the Restricted address space with external arbitration mode. *88*
Table 4: Status registers available in the Time-sharing with reset and lock-release arbitration mode. *90*
Table 5: Configuration registers available in the Time-sharing with context switching and Round-robin arbitration mode. *92*

List of Figures

Figure 1. The CROSSCON open security stack considered in the proposal. *17*
Figure 2. The refined CROSSCON stack. *18*
Figure 3. Multiple isolated environments on MSP architecture using software-only isolation. *22*
Figure 4. Multiple isolated environments on Armv6-M/v7-M architecture using basic memory isolation primitives. *23*
Figure 5. Multiple isolated environments in Armv8-M architecture using TrustZone-M assisted security primitive. *23*
Figure 6. Multiple isolated environment on both APU (Armv7-A/V8-A and CVA6) and RTU (Armv8-R and BA51-H) devices of Arm and RISC-V architectures. *24*
Figure 7. Multiple environment supporting multiple isolated execution contexts using hardware security primitives, including virtualization and TEE technologies on APU (Armv7-A/V8-A) devices. *25*
Figure 8. System stack representation of Armv6-M/v7-M architecture together with basic memory isolation primitives. *25*
Figure 9. CROSSCON Hypervisor CoVE Confidential VM. *30*
Figure 10. Guest Configuration *35*
Figure 11. VM Image Configuration *37*
Figure 12. Inter-Partition Communication (IPC) *39*
Figure 13. CPU Affinity Configuration *43*
Figure 14. LLC Coloring Configuration *43*
Figure 15. Shared Memory Configuration *44*
Figure 16. Comparision of typical virtualization scenario with one VMM and CROSSCON multi-VMM architecture. *56*
Figure 17. Current fully-featured architecture of the CROSSCON SoC *82*
Figure 18. A basic SoC with two masters, M1 and M2, connected to a HW module (MOD) through PG on a bus. *83*
Figure 19. How PG is used in the of CROSSCON SoC to restrict access to a HW accelerator. *84*
Figure 20. Format of $pg_addr_i_reg$ and $pg_acc_prm_i_reg$ where L equals $DID_WIDTH + 3$. *89*
Figure 21. The behavior of Time-sharing with reset and lock-release arbitration mode, where DX and DY represent domains. *90*
Figure 22. PG in context switching mode *91*
Figure 23. How stalling is used to provide time for the context switching mechanism to store the module's state of domain 1 (D1) and restore the module's state of domain 2 (D2). *92*
Figure 24. An example of how a cryptographic accelerator can be integrated within the CROSSCON SoC and Hypervisor through PG *93*

List of Abbreviations

Abbreviation / acronym	Description
APs	Access Points
APU	Application Processing Unit
CA	Client Application
CCA	Confidential Computing Architecture
CoVE	Confidential VM Extension
cVM	confidential VM
DMA	Direct Memory Access
DoS	Denial of Service
FPGA	Field Programmable Gate Array
GPOS	General-Purpose Operating Systems
GPUs	Graphics Processing Units
HW	Hardware
IOMMU	Input-Output Memory Management Unit
IOMPU	Input-Output Memory Protection Unit
IoT	Internet of Things
IPC	Inter-Partition Communication
ISA	Instruction Set Architecture
MCU	Micro Controller Unit
MFA	Multi-Factor Authentication
MMU	Memory Management Unit
MPC	Memory Protection Controller
MPU	Memory Protection Unit
OS	Operating System
PA	Physical Address
PG	Perimeter guard
PMP	Physical Memory Protection
PUFs	Physical Unclonable Functions
REE	Rich Execution Environment
RoT	Root of Trust
RTOS	Real-Time Operating System
RTU	Real-Time Processor Unit
SGX	Software Guard Extensions
SMPU	System Memory Protection Unit
SoC	System-on-Chip
SPI	Same-Privilege Isolation
SPMP	Second-stage Physical Memory Protection
SW	Software
TA	Trusted Application
TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TSM	Trusted Security Monitor
VA	Virtual Address

vFPGA	virtual FPGA
VM	Virtual Machine

Executive Summary

The purpose of the CROSSCON architecture is to enable the secure and isolated execution of security-sensitive tasks on a wide variety of IoT devices with very different levels of hardware support for security features. The goal of CROSSCON, therefore, is to define a flexible and adaptable set of architectural components that can provide a set of security features, maximally utilizing the capabilities of the underlying hardware platform to provide the best possible level of isolation.

This document (i) reviews the security assumptions and the threat model upon which the CROSSCON approach is built, (ii) provides a high-level overview of the components included in the CROSSCON stack, (iii) analyzes possible instantiation options depending on the security primitives of the underlying hardware and (iv) provides the configuration options associated with different CROSSCON components (v) presents details specifications for the APIs associated with different functional aspects of the CROSSCON stack. By using this approach, CROSSCON can support a very large variety of different hardware platforms from different architectures and classes. Besides existing hardware platforms, this document also explores the design of a CROSSCON-specific System on Chip design, which aims to provide the highest level of isolation guarantees with a minimal impact on the overall performance of the system.

1 Introduction

1.1 Purpose of the Document

The purpose of this document is to provide a general picture of the overall specification of the CROSSCON architecture, a description of the core stack components and instantiation options of CROSSCON. It also provides the API specifications for different services provided by the CROSSCON stack. This document will be used as a basis for the technical work of the project.

1.2 Relation to Other Project Work

The document is closely related to the initial version of the use case definitions in deliverable D1.4 [1] and the technical specification of corresponding overall requirements as documented deliverable D1.2 [2], as the use cases and requirements provide the setting that the CROSSCON architecture seeks to address. The work packages on the CROSSCON security stack (WP3) D3.1 [3] and the extensions to accommodate domain-specific hardware in CROSSCON (WP4) D4.1 [4] will use the draft specification laid out in this document as a starting point for their work. Thus, this document is related to deliverables D3.1 [3], D3.2, D4.1 [4], and D4.2, where the initial design implementation details belong to WP3 not WP2 related to the CROSSCON stack are detailed.

1.3 Structure of the Document

This document is structured into seven major sections. After this introductory section, Section 2 provides an overview of the threat and adversary model adopted by CROSSCON and discusses the impact of requirements on the CROSSCON architecture. After that, in Section 3, we describe the overall high-level architecture of CROSSCON and enumerate its different instantiation options based on the underlying hardware security primitives of the platforms on which CROSSCON is implemented. Section 4 then provides a detailed description of the individual architectural components comprising the CROSSCON architecture. In Section 5, we then provide the description of the interfaces of the CROSSCON's components, and in Section 6 describe the API specifications for CROSSCON. In Section 7, we present the considerations regarding Hardware (HW)/Software (SW) co-design, and in Section 8 conclusions of the document.

1.3.1 Changes w.r.t. D2.1

This document extends the draft presented in D2.1. Section 2 has been adapted to highlight the finalized specifications. The high-level architecture described in Section 3 is built primarily on D2.1. New details are added in Section 5 regarding the architecture specification of the CROSSCON stack. Section 6 is novel in its entirety as it provides the specifications of the CROSSCON APIs. Section 7 is newly added.

2 Overview

In the realm of secure computing, the Trusted Computing Base (TCB) serves as the critical foundation for ensuring system integrity and confidentiality. At its core lies the Underlying Hardware, which is pivotal in establishing trust and providing robust security features to fortify the entire system.

An innovative addition to the TCB is the CROSSCON Hypervisor, notable for its minimalist design and ability to integrate with the TCB seamlessly. Unlike traditional hypervisors, CROSSCON enhances security without compromising system performance. Its inclusion in the TCB improves the overall security posture by isolating critical components and enforcing strict separation between domains.

2.1 Adversary Model

Understanding the capabilities of the adversary is crucial in designing effective countermeasures. CROSSCON assumes a strong adversary that possesses considerable power to compromising system security. In particular, the adversary can inject malicious code into the kernel with full control over the system software. The adversary can also leverage the CROSSCON Trusted Execution Environment (TEE)Interface to set up malicious enclaves, taking advantage of multi-domain enclaves for sophisticated attacks.

In addition to software-based attacks, the adversary can exploit hardware peripherals to perform Direct Memory Access (DMA) attacks, attempting to gain unauthorized access to sensitive data. Furthermore, adversaries can employ cache side-channel attacks to extract information from the system's cache, potentially compromising confidentiality.

Despite these formidable capabilities, the Trusted Computing Base (TCB), backed by the robust Underlying Hardware and reinforced by the CROSSCON Hypervisor, is designed to be resilient against these adversarial threats. Through continuous innovation and diligent security practices, the TCB ensures a safe computing environment and bolsters confidence in the integrity of critical systems.

Excluded Adversaries. To streamline our security model, certain adversaries are explicitly excluded from posing threats to the TCB. Hardware-based attacks exploiting vulnerabilities in the underlying hardware are out of the scope of the adversary model as the underlying hardware is assumed to be trusted. Additionally, physical attacks, such as fault injection and side-channel exploits, are mitigated through hardware protections and robust countermeasures that lie outside the scope of CROSSCON and are, therefore, not considered here explicitly.

Denial of Service (DoS) Attacks are recognized as a potential threat. Adversaries who gain control over the Operating System could attempt to shut down the system, causing service disruptions. However, the TCB architecture and the CROSSCON mechanisms enable the system to withstand and recover from many DoS attacks, thereby reducing their impact and preserving system availability.

2.2 The Architectural Impact of Requirements

In the rapidly evolving realm of the Internet of Things (IoT), systems architecture must accommodate many requirements to ensure seamless interoperability across various devices. One of the central challenges in this landscape is the need for an architecture that embraces the heterogeneity of IoT devices. At the heart of this challenge lies the critical role of requirements, which establish the groundwork for a cohesive architecture capable of spanning various hardware architectures, capabilities, and trusted services.

Interoperability Across Heterogeneous IoT Devices. The focal point of architectural impact in the IoT landscape is the interoperability of devices that span a wide spectrum of form factors, computational capabilities, and hardware extensions. To address this, the architecture must be engineered to transcend the discrepancies in device capabilities and ensure consistent behavior across the device landscape. Requirements in this context serve as guiding principles that shape the architecture's ability to seamlessly integrate into diverse hardware ecosystems.

Diversity of Hardware and Architectures. IoT devices exhibit diversity not only in terms of their computational power but also in their underlying architectures and hardware extensions. The architectural impact of these requirements necessitates a design that is agnostic to the specifics of the hardware platform. This is especially critical in the case of Arm and RISC-V architectures, which dominate the IoT landscape. The architecture must provide an abstraction layer that enables the system to execute consistently and efficiently across these differing architectures.

Hardware Capabilities and Extensions. The variations in hardware capabilities across IoT devices, including features like cryptographic accelerators and security-oriented technologies (e.g. ARM TrustZone), demand an architecture that can accommodate such disparities. Architectural decisions must align with these requirements to ensure that the system can harness the full potential of the underlying hardware. At the same time, the architecture should avoid imposing unnecessary limitations that could hinder the system's scalability and adaptability.

Unified APIs for Global Interoperability. To harmonize the architecture in the presence of heterogeneous devices, globally accepted and standardized APIs are a cornerstone. These APIs serve as the lingua franca that enables different devices to communicate and interact seamlessly. In this context, the GlobalPlatform API [5] provides a solid foundation that can be extended and tailored to meet the specific needs of the IoT ecosystem. This approach promotes uniformity, reduces complexity, and enhances interoperability across the device landscape.

In summary, the architectural impact of requirements on the IoT landscape is profound and multifaceted. To realize a stack that can operate across a diverse range of IoT devices, designers must navigate the challenges posed by hardware heterogeneity, varied capabilities, and security considerations. By adhering to a set of unified APIs and leveraging established frameworks like the GlobalPlatform API, the architecture can strike a balance between flexibility and standardization, ultimately paving the way for a coherent and interoperable IoT ecosystem.

2.3 Assumptions and Constraints

2.3.1 Definitions of Key Concepts and Terminology

Specific terms can have multiple interpretations in the IoT domain. To ensure clarity, the following sections define the specific meanings of key concepts used in the CROSSCON context. The following sections introduce these concepts, serving as reference points throughout the document.

2.3.1.1 Trusted Execution Environment (TEE)

The notion of a Trusted Execution Environment (TEE) varies across the literature and industry and includes several different trusted components and architectural designs. In the research community, relevant examples include ReZone [6], Keystone [7], Komodo [8], and Sanctuary [9]. In the industry, implementations include Intel SGX [10], Hex Five MultiZone [11], SiFive WorldGuard [12], AMD SEV [13], Intel TDX [14], Arm Secure Partition Manager (SPM) [15], Arm CCA [16], and RISC-V CoVE [17].

As part of the current CROSSCON specification, a TEE is an isolated execution environment that guarantees:

1. **Authenticity** of the code executed inside of the environment,
2. **Integrity** of the assets (code, data, register file, stack, etc.) inside of the environment and
3. **Confidentiality** of the data (security key, processed data, etc.) inside of the environment.

Typically, as described in the TEE System Architecture of the GlobalPlatform standard document [1], a TEE is assumed to operate in a dual-environment system. One world hosts the Rich Execution Environment (REE), where the less security-critical operations take place, running anOperating System (OS) like Linux. The other world is dedicated to the execution of critical services. However, due to the Trusted Computing Base (TCB) size and other inherent limitations of a single trusted world, CROSSCON adopts a slightly different approach, leveraging a multi-world TEE concept. In CROSSCON, we adhere to the following assumptions:

- **We assume** as a **TEE technology** the underlining hardware that the device provides isolation among TEE software stack, i.e., the Arm TrustZone;
- **We assume** a TEE technology is primarily used to provide a secure execution environment for Trusted Application (TA)s and critical system assets essential to security and part of the RoT.
- **We assume** as a **TEE software stack** the environment composed by components like Trusted Applications (TAs), Trusted OSes and Firmware;
- **We assume** the components inside of a TEE software stack are trusted, meaning that they provide the expected behavior.
- **We assume** that the TAs running in a TEE software stack are isolated from each other by a trusted separation kernel or some other security mechanism.
- **We assume** that in CROSSCON we can have **multiple TEE software stacks** each running inside a Virtual Machine (VM) on top of CROSSCON Hypervisor. Therefore, a system could have several independent TEE software stacks so that if one of them is compromised, the other TEEs are not affected. See Section 4.1.2 for further explanation.

2.3.1.2 Rich Execution Environment (REE)

As part of the current CROSSCON specification, we define the Rich Execution Environment (REE) as a complex and feature-rich environment for general computation tasks, including OSes like Linux and FreeRTOS, along with the applications running on top. As mentioned above, REE is often viewed as a counterpart to the TEE, hosting system components designed for general-purpose computation. However, the CROSSCON stack aims to enable the creation of TEE software stacks with strong security guarantees within the REE. This is achieved through the CROSSCON Hypervisor and classical hardware security primitives (e.g., MMU, MPU).

2.3.2 General Objectives

The following are the general objectives for CROSSCON.

- **CROSSCON will** support the presence of a TEE. If it is present, we may decide to use it (optionally).
- **CROSSCON will** guarantee the integrity and confidentiality of data while supporting services operation;
- **CROSSCON will** support more than one REE SW stack executing on the same device, depending on the architecture and the hardware security primitives the device offers.
- **CROSSCON will** support more than one TEE SW stack running on the same device, depending on the architecture and hardware security primitives offered by the device.

- **CROSSCON will** support the presence of an Field Programmable Gate Array (FPGA). If it is present, we may decide to use it (optionally).

2.3.3 General Assumptions

The following are CROSSCON's general assumptions.

- **CROSSCON assumes** multiple TEE SW stack per architecture and device.
- **CROSSCON assumes** that Baremetal TEE should provide TEE software stacks on devices that either lack hardware security primitives or have only basic HW security primitives.
- **CROSSCON assumes** CROSSCON Hypervisor as the component responsible to provide multiple TEE SW stacks in a single platform (when hardware virtualization primitives available);
- **CROSSCON assumes** multiple TEE SW stacks running in REE by leveraging virtualization security primitives. Therefore, a software running in a TEE is trusted and can, therefore, coexist with the REE components;
- **CROSSCON assumes** a Hypervisor is primarily used to provide multiple isolated execution environments under the same device, supporting either General Purpose Operating Systems (GPOS), Real-Time Operating Systems (RTOS) and bare metal applications;
- **CROSSCON assumes** trusted services should follow a GP compliance;

2.3.4 General Constraints

The following are CROSSCON's general constraints. Specific constraints for the different specializations are reported in the respective section later on.

- Despite TEE technologies provided by different architectures we only use the trusted execution on top of CROSSCON Baremetal TEE or CROSSCON Hypervisor;
- CROSSCON components support architectures like RISC-V(BA51-H and CVA6), some versions of the Arm ISA (Armv6-M/v7-M/v7-A/v8-A/v8-M), and some versions of the MSP430 ISA.
- On bare metal systems, CROSSCON offers a limited set of security features compatible with the underlying architecture and device HW security primitives.

3 High-level Architecture

This chapter describes the high-level architecture of the CROSSCON stack, describing in detail the various instantiation options available to users depending on the capabilities of the underlying hardware.

3.1 Unified Abstract Model

In the project proposal, we considered the following high-level architecture for the CROSSCON stack. In Fig. 1. In the CROSSCON open security stack featured in the proposal, the new components (marked in green) extend interoperability across heterogeneous devices, offer a unified level of abstraction across multiple hardware platforms, and enrich existing security features by adding new trusted services.

During the execution of the project, this initial architecture has been further refined to create the following abstract model shown in Fig. 2.

We distinguish two high-level cases: i) deployment in the case of a Rich Execution Environment (REE) with different REE privilege levels, or ii) deployment in the case of the presence of a Trusted Execution Environment (TEE) also with different TEE privilege levels. In both cases, at the lower levels of the CROSSCON stack, we have the Instruction Set Architecture (ISA) that varies depending on the CPU architecture families (e.g., RISC-V, Arm with all their variants) considered, each equipped with possibly different set of capabilities (e.g., the presence of a TEE, privilege levels). On top of the ISA, we consider the possible presence of firmware (which may exist or not, depending on the CPU architecture and the needs of the above layers of the stack).

In the first case, the CROSSCON Hypervisor is software deployed at the hypervisor level, making it the highest privilege software in execution (apart from the firmware). Depending on the ISA and the class of the device, the Hypervisor leverages either a dedicated privileged level or the highest privileged level to operate. Each VM is isolated from the rest of the system via second stage memory isolation: it will have access to a separate set of system resources (e.g., memory, I/O, CPU). The hypervisor will then manage the physical resources, assigning them to the various VMs. This allows each VM to be isolated and secured from the rest of the system, which is incapable of accessing the resources allocated to the VMs. Notably, in the REE scenario, there is no substantial difference between a trusted application (TA) and an untrusted application since both enjoy the same level of protection, being completely isolated. On top of handling the physical resources, the hypervisor is entrusted with instantiating various VMs, some of which can be loaded dynamically upon request. For instance, an untrusted VM can request the allocation of a trusted application, which should be bootstrapped on demand by the hypervisor and then

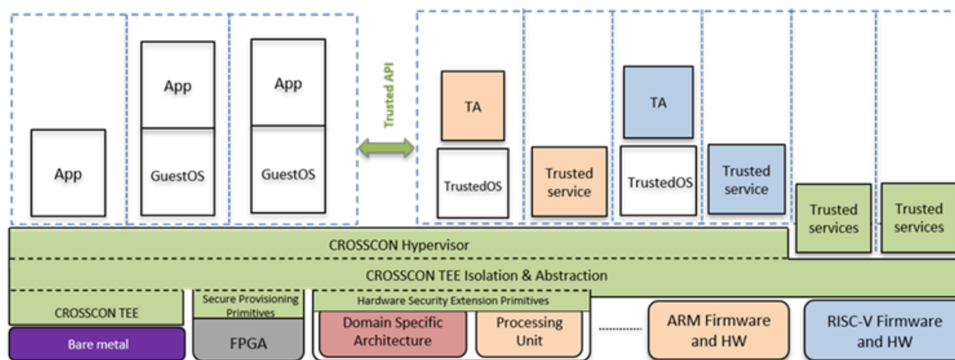


Figure 1: The CROSSCON open security stack considered in the proposal.

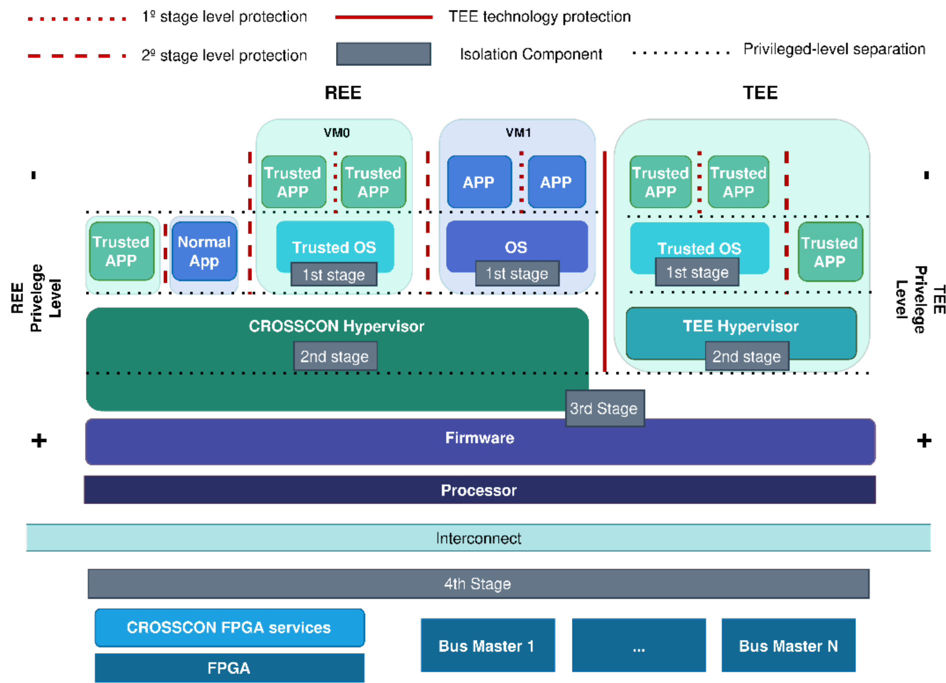


Figure 2: The refined CROSSCON stack.

destroyed when no longer needed. Finally, we support the presence of an FPGA on the system to provide specific security services through the CROSSCON FPGA service layer. This might act as a hardware RoT that can be leveraged by either the REE or the TEE.

Fig. 2 is a high-level representation of the CROSSCON architecture on a device fully equipped with the latest and most advanced security hardware. The following sections present an overview of how CROSSCON tackles real scenarios. We have discussed the objectives, assumptions, and constraints our work builds on in great detail in Section 2.

3.2 System Stack Components

The CROSSCON system stack could comprise several combinations of the components mentioned in Section 4, which could lead to different instantiation options. A single environment in the CROSSCON system stack could range from a simple application running in an Micro Controller Unit (MCU) platform encompassing just one privilege level to a complex system stack operating in an Application Processing Unit (APU) platform encompassing multiple trusted environments around four privilege levels. Section 3.3.3 demonstrates the possible instantiation options that CROSSCON will consider.

Every instantiation option hinges on the presence of the two most privileged components to support virtualization and TEE technologies - together or independently. In instantiations that include TEE or virtualization, 4th stage configuration should be configured with care to ensure that other bus masters cannot attack undue system components.

3.3 Architecture Description

The computing spectrum of IoT devices is very heterogeneous, encompassing several architectures of three main types of processors: the Micro-controller Unit (MCU), Real-Time Processor Unit (RTU) and Application Processing Unit (APU).

Micro-controller Unit (MCU). MCUs are bare-metal devices featuring limited resources, such as a few kilobytes of RAM and tens of kilobytes of Flash memory. They integrate processing cores, memory, and peripherals on a single chip, making them cost-effective and ideal for resource-constrained designs. Examples of IoT devices are: industrial controllers, drones or home appliances. Most MCUs operate using bare-metal programming or lightweight Real-Time Operating System (RTOS).

Real Time Processor Unit (RTU). RTUs are processors tailored for real-time operation in critical systems. RTUs are optimized for deterministic performance and low latency response, ensuring precise control in time-sensitive applications. They are commonly used in industrial automation, robotics, and control systems where real-time execution is essential.

APU (Application Processing Unit (APU). APUs are general-purpose processors tailored for complex and computation-heavy tasks. They integrate multi-core architectures, reconfigurable hardware and large number of memories. APUs are widely used in applications that require high computational power, such as automotive systems, industrial automation, and robotics. They typically run general-purpose operating systems like Linux or Android.

Such heterogeneity is also reflected in the architecture and the hardware security primitives the device provides, ranging from no security at all to implementation of Root of Trust (RoT), cryptographic engines or isolated execution environments.

3.3.1 Architectures

To cover this heterogeneity, the CROSSCON project encompasses various architectures, covering a wide computing spectrum of IoT devices. Table **??** lists all architectures considered by CROSSCON, along with their respective hardware security primitives. Starting with MCU architectures, the MSP represents the most constrained architecture, lacking any hardware security primitives.

Armv6-M/v7-M and RISC (M+U) provide basic hardware security primitives, MPU for Arm and Physical Memory Protection (PMP) for RISC-V. Armv8-M introduces TrustZone-M and Memory Protection Controller (MPC) to mediate non-CPU bus master accesses.

Moving to the RTU class of processors, Armv8-R has recently extended basic hardware security primitives by introducing virtualization primitives such as the second-stage MPU (for the Arch64 architecture) and protection against non-CPU bus masters through System Memory Protection Unit (SMPU). Similar security primitives are present in RISC-V devices with BA51-H processors, where SPMP corresponds to the MPU, PMP to the second-stage MPU, and Perimeter Guard to the SMPU.

Finally, in the APU class of processors, Armv7-A and Armv8-A include Arm TrustZone-A, Memory Management Unit (MMU) Memory Management Unit (MMU), second stage MMU, and Input-Output Memory Management Unit (IOMMU). For APU RISC-V architectures, such as CVA6 processors, the available primitives include PMP, MMU, second-stage MMU, and IOMMU. In addition to the hardware security primitives provided by the respective architectures, some devices offer FPGA, which can be leveraged to implement missing security primitives.

3.3.2 Hardware Security Primitives

The following is a description of each hardware security primitives that CROSSCON addresses. It is important to note that other architectures could include additional hardware security primitives.

Software-based isolation

Memory isolation can be achieved through software by instrumenting and verifying the code. Whenever an instruction accesses memory, either to read, write or execute, the software itself must check

Table 1: Features of platforms selected for CROSSCON.

Type	Architecture	HW Security Primitives
MCU	MSP430	(No HW Primitives)
	Armv6-M	MPU
	Armv7-M	MPU
	RISC-V(M+U)	PMP
	Armv8-M	Arm TrustZone-M MPU MPC
RTU	Armv8-R	MPU (2nd Stage) MPU SMPU MMU
	RISC-V(BA51-H)	SPMP PMP Perimeter Guard
APU	Armv7-A	Arm TrustZone-A MMU (2nd Stage) MMU IOMMU
	Armv8-A	Arm TrustZone-A MMU (2nd Stage) MMU IOMMU
	RISC-V(CVA6)	PMP MMU (2nd Stage) MMU IOMMU

whether this operation adheres to a certain access policy. If the check passes, then the memory access is performed; otherwise, the software must throw an error before carrying out the memory access.

To achieve software isolation, each memory access performed by the software must be verified, either statically or dynamically. In particular, we can classify memory operations into two types: static and dynamic. Static memory operations, i.e., instructions that target an hard-coded memory region, can be vetted at compile-time or prior being executed by a binary verifier module. This will ensure that a piece of code is executed only if it does not contain illicit static operations. Dynamic memory operations, i.e., instructions that target a memory region computed at run-time, require run-time checks. Specifically, an explicit check is performed before the memory operation. The check will ensure that the target memory region can be accessed, throwing an error if that is not true.

Memory Protection Unit (MPU) and Physical Memory Protection (PMP)

The Memory Protection Unit (MPU) is a hardware-based security primitive commonly found in modern processor architectures. It is used to restrict processor access permissions based on the execution context (e.g., privileged vs. unprivileged). The MPU enforces access rules to specific memory regions, preventing unauthorized or unintended physical memory accesses by defining and enforcing permissions for read, write, and execute operations.

The Physical Memory Protection (PMP) is a hardware security feature commonly found across the spectrum of RISC-V architectures, from MCUs to APUs. It is an MPU, controlling and restricting processor access to memory regions based on predefined permissions.

Memory Management Unit (MMU)

The MMU is a hardware security primitive commonly found in modern processors, including APUs. Its primary function is to manage virtual memory translation, enabling programs to operate without requiring knowledge of the physical memory layout. By abstracting physical memory, the MMU enables

efficient memory utilization and provides isolation between applications and the underlying hardware. Typically, the Operating System (OS) leverages the MMU to enforce memory isolation, ensuring secure and independent operation of multiple processes at low-privileged software levels.

Second stage MMU

A second-stage MMU is a hardware security primitive that provides memory virtualization capabilities. By adding extra layers of memory access control and translation, they improve both security and flexibility, enabling multiple operating systems to run on a single platform. For example, the second-stage MMU in the Cortex-A53 architecture allows a hypervisor to isolate the memory of each VM from the others.

Second stage MPU and S-Mode PMP (SPMP)

Second-stage MPUs are hardware security primitives that provide enhanced memory access control capabilities. By adding extra layers of memory access control, they improve both security and flexibility, enabling multiple operating systems to run on a single platform. For example, the second-stage MPU in the ARM Cortex-R52+ architecture allows a hypervisor to isolate a VM memory from others, ensuring protection.

In RISC-V, the Second-stage Physical Memory Protection (SPMP) implements a secondary MPU layer. The current proposal introduces two distinct SPMP units: one for restricting memory access in virtualized modes (VS-mode) and another for user mode (U-mode). As part of the CROSSCON project, we proposed a unified SPMP model that consolidates these into a single SPMP unit, capable of restricting memory access for both virtualized and non-virtualized modes. This unified approach simplifies the design and reduces hardware overhead, particularly when virtualization is not in use. The extension is implemented in hardware as part of the BA51-H architecture.

Arm TrustZone

Arm TrustZone is a hardware-based security technology designed to create and manage a Trusted Execution Environment (TEE). It employs a dual-world architecture, dividing system execution into secure and non-secure worlds. This isolation ensures the protection of sensitive data and secure operations, protecting against unauthorized access and potential attacks. TrustZone is widely implemented across Arm devices, including MCUs (in Armv8-M architectures) and APUs (in Armv7 and Armv8 architectures).

Input-Output Memory Management Unit (IOMMU) and Input-Output Memory Protection Unit (IOMPU)

The IOMMU and Input-Output Memory Protection Unit (IOMPU) play a crucial role in securing and optimizing data transfers between non-CPU bus masters, such as DMA controllers, Graphics Processing Units (GPUs), network adapters or system memory. The IOMMU extends virtual memory concepts to peripheral devices by translating their memory addresses into physical addresses, similar to how an MMU operates for the CPU. In contrast, the IOMPU focuses solely on access control rather than address translation. It enforces security policies by restricting memory access to authorized devices, ensuring system integrity and protection.

Perimeter Guard (PG)

The Perimeter guard (PG) is a hardware security primitive for RTU devices that enables the secure sharing of hardware modules between VMs while maintaining isolation. The PG module is positioned between the SoC interconnect and the shared hardware module and controls access by determining which VMs and bus masters can interact with the module. Additionally, PG ensures security by resetting or switching

the hardware module's state before granting access to a different VM or master, preventing any residual state-related information flow.

PG supports multiple operational modes, although the current prototype only implements time-sharing with reset mode and lock-release arbitration. In the CROSSCON SoC, PG is used to manage access to hardware modules, for example, a cryptographic accelerator, allowing secure sharing between VMs without compromising isolation.

3.3.3 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 - are assumed to support the execution of multiple trusted applications.

We consider the following CROSSCON instantiation options:

1. Software Only Isolation Environment
2. Basic Memory Isolation Environment
3. Virtualization-Less Environment with TEE
4. TEE-less Environment with Virtualization
5. Environment with TEE and Virtualization
6. Environment with Virtualization, and FPGA

3.3.4 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 3 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 Subsection 4.1.3.

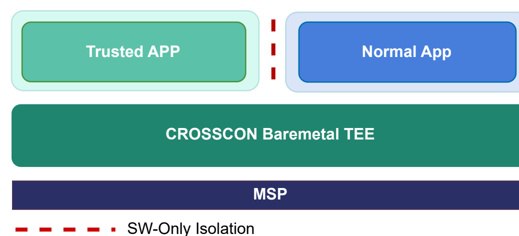


Figure 3: Multiple isolated environments on MSP architecture using software-only isolation.

3.3.5 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 4, 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.

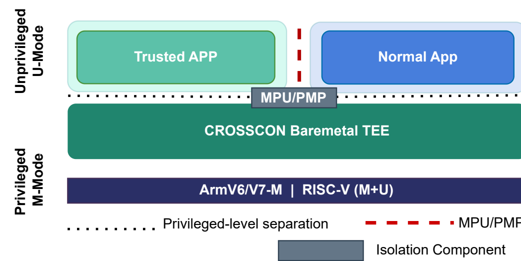


Figure 4: Multiple isolated environments on Armv6-M/v7-M architecture using basic memory isolation primitives.

3.3.6 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 5 shows an example of a system stack featuring multiple isolated environments on an Armv8-M architecture using TrustZone-M.

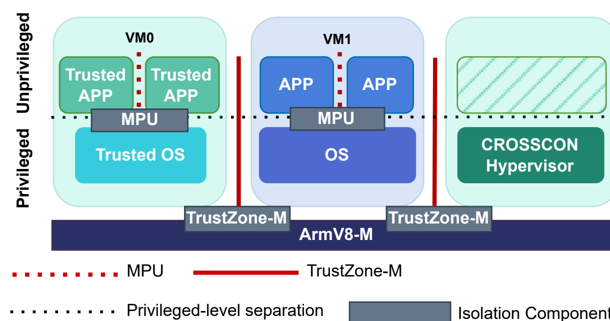


Figure 5: Multiple isolated environments in Armv8-M architecture using TrustZone-M assisted security primitive.

3.3.7 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 6) 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.

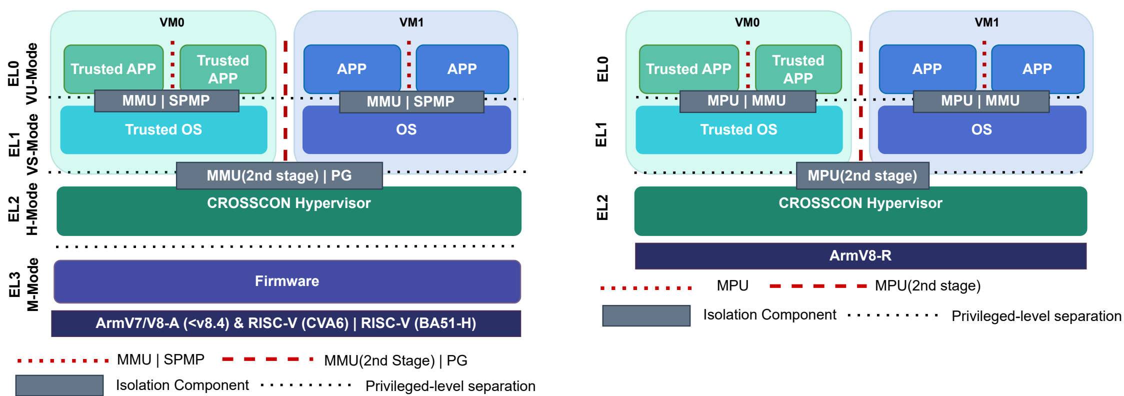


Figure 6: Multiple isolated environment on both APU (Armv7-A/V8-A and CVA6) and RTU (Armv8-R and BA51-H) devices of Arm and RISC-V architectures.

3.3.8 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 7 illustrates the hypervisor's role in managing multiple trusted services within isolated VMs.

3.3.9 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 8. 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 in the WP1 classification.

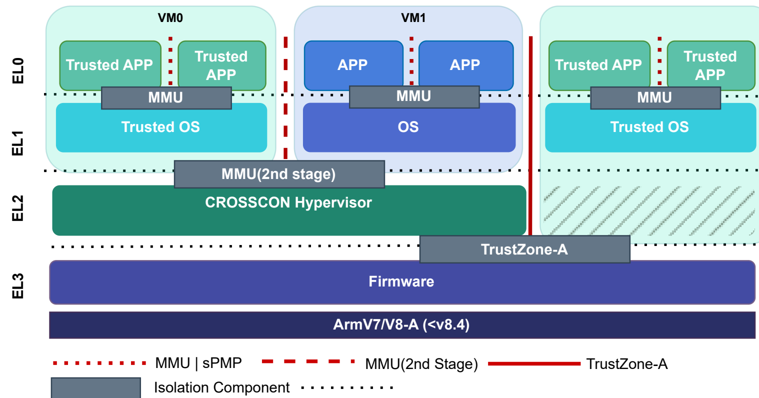


Figure 7: Multiple environment supporting multiple isolated execution contexts using hardware security primitives, including virtualization and TEE technologies on APU (Armv7-A/V8-A) devices.

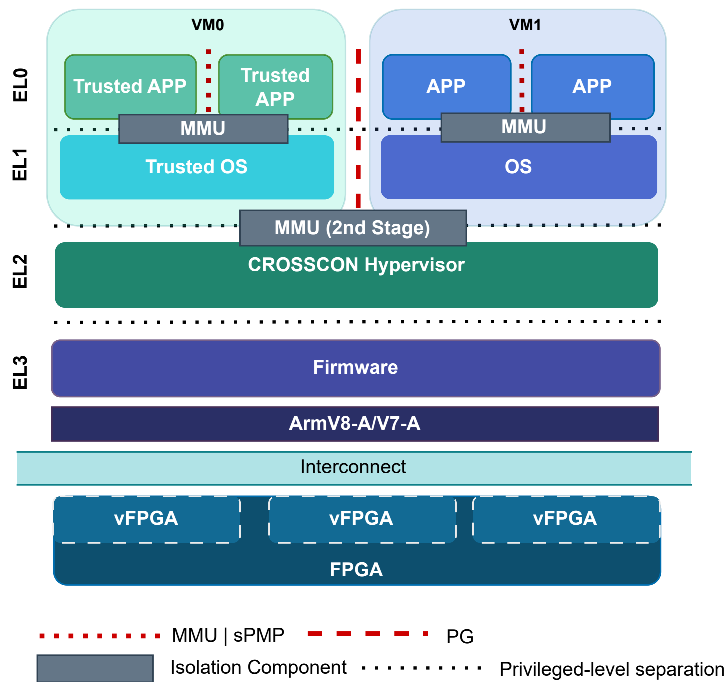


Figure 8: System stack representation of Armv6-M/v7-M architecture together with basic memory isolation primitives.

4 Description of Architecture Components

The CROSSCON stack comprises a number of components that are used to realise the different instantiation options of the CROSSCON architecture outlined in Subsection 3.3.3. The following sections describe each possible stack component in more detail. Also, in Section 4.2, we provide a description of the new CROSSCON trusted services providing higher-level security functionalities to applications utilizing the CROSSCON architecture.

4.1 CROSSCON Stack Components

This section describes each potential architecture component within the CROSSCON system stack. CROSSCON system stack components are presented in a bottom-up order, starting from the component in the highest privileged level, the Firmware component, which may exist or not, depending on the CPU architecture and the needs of the above layers of the stack. Proceeding to the privileged levels above, we encounter the Hypervisor component for managing virtual environments, the respective OS Guests, and their applications. TEE SW components are placed either as a guest and above the hypervisor, i.e., running inside a VM, or in some platforms above the firmware to support trusted services operations. Lastly, the confidential computing technology components as potential supplementary and optional elements of CROSSCON architecture at the designing time. Before going into the CROSSCON Stack Component's details, we describe the objectives, assumptions, and constraints related to CROSSCON stack components.

4.1.1 Firmware

Firmware components refer to the most privileged software layer in the system stack, typically embedded into the non-volatile memory of hardware devices, such as microcontrollers or SoC devices. The firmware acts as an intermediary between the hardware and the OS or the hypervisor, providing a layer of abstraction that facilitates interaction between hardware and software, e.g., providing necessary services and implementing device-specific functionalities.

In the realm of Arm technology, TF-A (Trusted Firmware-A) and TF-M (Trusted Firmware-M) [18] are the reference firmware implementations. This firmware implementation is a universal reference for the Trusted Board Boot Requirements (TBBR). TF-A and TF-M offer a standardized approach to hardware initialization, memory setup, secure boot support, and mechanisms for secure firmware updates. TF-A also acts implements secure switching between the TEE and the REE for APU devices.

When delving into RISC-V technology, the prevailing choice for open firmware implementation is OpenSBI [19]. OpenSBI provides features like processor initialization, memory configuration, and security features such as Physical Memory Protection (PMP). Additionally, OpenSBI provides runtime services to allow secure execution of sensitive operations restricted to the higher privileged layer, e.g., configuration of timer registers, TLB flushing, etc.

4.1.2 Trusted Execution Environment (TEE)

TEE technologies are hardware security primitives designed to protect application integrity and confidentiality. They allow to run security-sensitive applications in isolated execution, distinct from the host platform's OS. The resulted TEE software stack includes components such as: trusted applications (TAs) and Trusted OSes (TOSes).

Trusted Application (TA)

As the name suggests, TAs are software programs that run at user privileged level, within the secure world, and have access to the secure services provided by the TEE. They handle sensitive data, cryptographic operations, and other critical functions, ensuring the confidentiality and integrity. For example, secure mobile payment applications and digital rights management systems are often implemented using trusted applications, leveraging the TEE to safeguard sensitive user information.

Client Application (CA)

A Client Application (CA) is simply any application running outside the TEE making use of the TEE Client API to access the facilities provided by the TAs inside the TEE. Every untrusted application running on the device should be considered as a CA.

Trusted OS

Trusted OSes operate at the supervisor privilege level (OS privilege level within the Secure World), providing TA management and trusted peripheral access through dedicated device drivers. They offer TEE primitives such as remote attestation, trusted I/O, and secure storage, ensuring a secure environment for critical operations and data protection.

For Arm devices, CROSSCON leverages Arm TrustZone, a widely adopted technology in mobile [20], industrial [21, 22], server [23], and low-end devices [20]. Since 2004, Arm TrustZone has been integrated into APU devices as TrustZone-A. In 2016, it was extended to resource-constrained devices, such as microcontrollers, with TrustZone-M in the Cortex-M23 and Cortex-M33 processors [20].

Both TEE technologies follow a dual-world architecture, comprising a non-secure world (REE) and a secure world (TEE software stack). This design mandates a single TEE software stack to manage all trusted services on a single TEE kernel. However, this approach increases the Trusted Computing Base (TCB) and expands the attack surface, potentially undermining the intended security benefits. Additionally, many Trusted OS implementations are proprietary and follow distinct architectural choices (e.g., TEEGRIS, QSEE, Kinibi, mTower), limiting interoperability between their security services. Unsurprisingly, vulnerabilities in these TEE implementations have been exploited, compromising system confidentiality and integrity.

4.1.3 CROSSCON Baremetal TEE

TEE technologies are gaining traction in recent architectures, and are often deployed on top of mid- to high-end devices. Unfortunately, older architectures and lower-end devices are not equipped with such strong security primitives. Instead, these devices may rely on simple Memory Protection Units (MPUs) and a few privilege levels. On the one hand, if these features enable memory isolation primitives, on the other hand, their security guarantees are far from those offered by TEE technologies. To exacerbate this picture, the more constrained devices can lack both MPU and privilege levels, thus making memory isolation a most challenging task.

The CROSSCON Baremetal TEE is a software-based solution designed to provide TEE-like features and security guarantees to these constrained devices. The Baremetal TEE sits between the baremetal user-level application and the hardware, enforcing a minimal yet comprehensive memory isolation between the two. This creates two virtual worlds: the REE, where untrusted applications are executed, and the TEE, where applications are well isolated. Interestingly, given the limited resources of these devices, the REE must be a baremetal application. Inside our software TEE, various TAs can operate safely and interact with the untrusted application through APIs. These TAs, in most cases, can be used to enable the interaction between the different CROSSCON devices.

To enforce the access control policy that isolates the memory between the REE and the TEE, the Baremetal TEE uses the MPU (if present) or software virtualization. This ensures that the TEE domain is protected against unregulated accesses from the untrusted application. Moreover, the Baremetal TEE can be executed at a higher privilege level with respect to the application, or at the same level. In the latter scenario, a more extensive instrumentation of the application is required.

4.1.4 CROSSCON Hypervisor

Hypervisors have proven to be effective in managing trusted components, extending their oversight even into REE environments. Research demonstrates that hypervisors can enable lightweight VMs in the REE, allowing developers to offload complex functionality from the Secure World. This reduces the TCB while maintaining system security. [24, 9]

In the new CROSSCON abstract model, Figure 2, the hypervisor sits immediately above the firmware. The hypervisor's ultimate objective is to host isolated VMs, ensuring they can run as if they were operating independently on separate hardware. The hypervisor operates within a dedicated layer, with higher privileges than the OS, managing access to hardware resources by leveraging various isolation mechanisms.

For effective resource isolation, including shared architectural resources, such as last-level caches, interconnects, and memory controllers, the hypervisor supports various security features according to the underlying hardware and ISA (see Section 3.3.3). For example, in APU processors, the hypervisor leverages virtualization extensions, including 2nd stage translation and other virtualization techniques, to establish isolation between VMs. In contrast, in Armv8-M featuring TrustZone, the hypervisor employs the TEE security primitives to restrict access to physical memory for each guest without the need for virtualization mechanisms.

The CROSSCON hypervisor is designed to ensure strict resource isolation across VMs, preventing hardware resource sharing while enforcing strong architectural and micro-architectural isolation through built-in mechanisms such as cache coloring. This approach enhances TEE architectures by integrating a thin, static partitioning hypervisor, inspired by a microkernel-like design, to strengthen security guarantees. The hypervisor enforces access control policies, ensuring VMs can securely execute security-sensitive workloads, such as running a Trusted OS and TAs. Additionally, the hypervisor addresses the limitations of static partitioning hypervisors, specifically their lack of flexibility in dynamically creating and managing new VMs and services [3].

4.1.5 Virtual Machines (VM)

As previously mentioned, the REE is a complex, feature-rich environment designed for general computing tasks, including general-purpose OSes and applications. When using the CROSSCON Hypervisor, REEs run within VMs. Without a hypervisor, they function as stand-alone components, operating without hypervisor-enforced access control. These components may include both bare-metal applications and OSes.

The OS operates at the supervisor-privilege level, either as a guest (under a hypervisor) or standalone. It handles hardware communication, implements device drivers, thread management, networking, etc.

Without an OS, bare-metal applications run at the supervisor- or user-privilege level (depending on the ISA) and provide basic services, either as stand-alone applications or within a VM.

CROSSCON supports various scenarios where OSes and bare-metal applications coexist, integrating different CROSSCON instantiation options (see Section 3.3.3).

Beyond these scenarios, the CROSSCON Hypervisor also enables trusted execution within VMs. VMs can

be paired with one or more TEE software stacks, including a Trusted OS and trusted applications. When an OS requests a trusted service, the request is securely redirected to the appropriate VM, ensuring isolation and secure processing.

4.1.6 Confidential VMs

Typically, memory is assigned to TEEs statically during the boot phase, and a limited number of entries to mark memory regions as part of the TEE is available. New technologies such as Arm Realms and RISC-V Confidential VM Extension (CoVE) allow resources to be transitioned between untrusted and trusted components dynamically and flexibly.

Confidential Computing Architecture (CCA) in Armv9 introduces Realms, a feature that provides isolation between sensitive workloads with the resource management flexibility of virtualization-based solutions. Within Realms, memory and CPU resource allocation are managed by a hypervisor operating within the non-secure world rather than being overseen by the platform manufacturer, allowing for larger and more complex workloads to be isolated. The transition of resources from the normal world to the realm world (and even a secure world) is overseen by the monitor, which executes in a world of its own (root world). A dedicated monitor executes with hypervisor-level privileges in the realm world and is responsible for the overall management of the lifecycle and operation of confidential VMs.

A similar concept to Realm exists in the RISC-V landscape, CoVE. CoVE aims to isolate applications or VMs from potential attacks launched via higher-privileged components, such as OSs and hypervisors. CoVE builds on the concept of a TEE Virtual Machine, i.e., confidential VM (cVM), which mirrors the approach taken by Realm, where non-TEE elements (hypervisor components) continue to manage confidential components. This management is performed by the Trusted Security Monitor (TSM), a component located at the hypervisor privileged level of the TEE side, acting as an intermediary between the hypervisor and cVMs. Importantly, it is endowed with the capability to manage, execute, and destroy cVMs, thereby ensuring effective and secure management of these entities.

Both Realm and CoVE present an advancement by affording lower-privileged software entities, such as applications or virtual machines, the means to safeguard their operations. This is achieved by preventing attacks originating from higher-privileged layers of the system stack, including OSes and hypervisors. It is important to note, however, that these technologies are relatively new, and as a result, there is a lack of availability and compatibility with COTS products.

4.1.6.1 Confidential VM Compatibility

In APUs, the CROSSCON Hypervisor operates in hypervisor mode, presenting unique challenges when integrating with software stacks that include other hypervisors, particularly in scenarios involving TEEs that implement confidential VMs (cVMs). This model pairs an untrusted hypervisor with a trusted hypervisor to ensure the confidentiality and integrity of VMs.

To integrate into such stacks, the CROSSCON Hypervisor must coexist with both untrusted and trusted hypervisors. This requires support for nested virtualization, enabling the CROSSCON Hypervisor to provide the infrastructure needed to host the trusted hypervisor or TSM, ensuring secure cVM management. The CROSSCON Hypervisor can address this by hosting and creating guest hypervisors through nested virtualization techniques, such as para-virtualization, trap-and-emulation, or hardware-assisted virtualization, depending on platform capabilities. This allows the CROSSCON Hypervisor to coexist with other hypervisors, enabling flexible, multi-layered virtualization. For example, it can integrate with RISC-V CoVE, where the TSM acts as a guest hypervisor, as illustrated in Figure 9. Untrusted guest hypervisors or guest OSes in the CROSSCON Hypervisor invoke the TSM to interact with CoVE's cVMs. By supporting nested virtualization and additional hypervisors, the CROSSCON Hypervisor provides a versatile platform for complex, multi-layered scenarios, ensuring adaptability across diverse use cases.

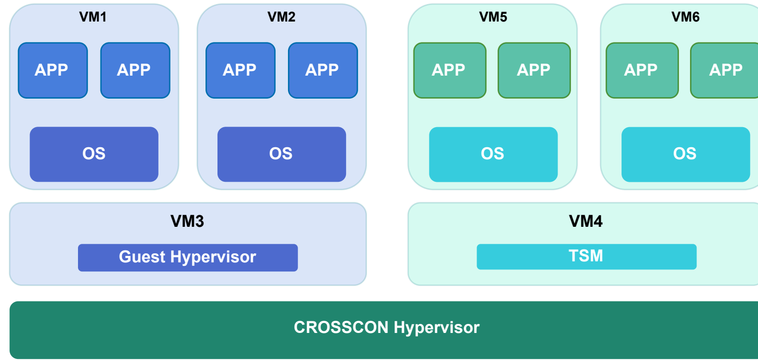


Figure 9: CROSSCON Hypervisor CoVE Confidential VM.

4.2 CROSSCON Trusted Services

In the following sections, we discuss the various Trusted Services provided by CROSSCON. We begin by exploring context-based authentication, which leverages the device's environment to create unique fingerprints, enhancing security, especially for second-factor authentication. Subsequently, we introduce Physical Unclonable Functions (PUFs) as a promising approach that aligns with multi-factor authentication principles and offers both supplementary and standalone security. The discussion then shifts to Secure FPGA Provisioning, aiming to enable resource-constrained IoT devices to securely offload complex tasks to FPGA accelerators within our trusted execution environment, emphasizing the preservation of confidentiality, safeguarding against malicious workloads, and ensuring FPGA design security. Finally, we describe the control-flow integrity primitives integrated in the CROSSCON Baremetal TEE.

4.2.1 Context-Based Authentication

We aim to facilitate the possibility of utilizing the environment of a device for context-based authentication. This authentication may be deployed as an additional layer to existing techniques. In our case, we plan to leverage this approach to enable a second-factor authentication. Specifically, we target surrounding Wi-Fi Access Points (APs) to collect metadata to create a fingerprint of the Wi-Fi landscape around a device. During the enrollment phase, a device captures this metadata, collects the information to constitute a fingerprint, and finally sends it to an authoritative party, e.g., an authentication server. This party then utilizes the composed fingerprint as an input to a Siamese network, which is well-suited to determine the similarity of fingerprints. Later on, during the authentication phase, a device reiterates the fingerprinting process. Eventually, the authoritative party can distinguish between different device identities by comparing the similarity of the fingerprints collected during the enrollment and authentication phases. The proposed API for the Context-based authentication trusted application can be found in Section 6.8.

4.2.2 PUF-Based Authentication for IoT Devices

A Physical Unclonable Function is a unique physical characteristic or property of a device that can be leveraged for authentication purposes. Unlike traditional cryptographic methods that rely on secret keys stored in software or hardware, PUFs extract their security from the inherent randomness and complexity of physical structures or processes. This makes PUF-based authentication particularly robust against various attacks, including those attempting to clone or replicate the authentication mechanism. The focus of our exploration lies in developing a PUF-based authentication scheme that fosters secure communication and interaction between mutually mistrusting parties within the IoT ecosystem. In this context, mutual mistrust refers to the absence of a trusted central authority or a shared secret key

between the parties involved. The objective is to establish a reliable authentication framework that safeguards against unauthorized access, data breaches, and other malicious activities.

The proposed authentication scheme aligns with Multi-Factor Authentication (MFA) principles, which enhances security by requiring users to provide multiple verification forms before gaining access to a system or device. In this scenario, the PUF-based authentication serves as a potential second factor, supplementing the traditional username-password combination with a distinct and tamper-resistant layer of protection. This approach significantly raises the bar for potential attackers, as they would not only need to compromise the user's credentials but also navigate the intricacies of the PUF-based authentication mechanism. Furthermore, the versatility of the PUF-based authentication scheme extends beyond being a supplementary factor within multi-factor authentication. It is designed to stand alone as a robust authentication method, providing a solid foundation for securing IoT devices and systems even when multi-factor authentication is not implemented. This is particularly advantageous for scenarios where the user experience needs to be streamlined or when implementing additional authentication factors is not feasible.

In conclusion, our exploration into PUF-based authentication for IoT devices seeks to address the pressing security concerns that arise in an interconnected world. By leveraging the unique and unclonable physical attributes of devices, we aim to establish a trustworthy and tamper-resistant authentication scheme that not only enhances security in IoT interactions, but also provides a potential second factor for multi-factor authentication or operates effectively as a standalone authentication method. The proposed API for the PUF-based authentication trusted application can be found in Section 6.6.

4.2.3 Trusted Application for Secure FPGA Partitioning

The Secure FPGA provisioning service aims to allow CROSSCON-enabled client devices to provision and run workloads like AI inference tasks on remote FPGA accelerators. Especially resource-constrained IoT devices can, therefore, run potentially complex and compute-intensive tasks by offloading such tasks to the cloud. However, for the secure use of such remote accelerated workloads by CROSSCON clients, a number of requirements must be met. Firstly, as the client workload code may potentially contain proprietary, sensitive information that is the intellectual property of the client, the service must make sure that the confidentiality of the workload is preserved during provisioning so that neither the cloud-based FPGA acceleration service provider nor any other party may obtain access to the plaintext code of the workload. Secondly, the FPGA acceleration service must be protected from malicious workloads that could disrupt other clients or even damage the FPGA hardware. To accommodate these requirements, the Secure FPGA provisioning service will provide two main sub-services: the FPGA Root of Trust (RoT) Setup service and the Secure FPGA Configuration/Bitstream Scanning service. Section 6.10 documents APIs exposed to clients using the Secure FPGA provisioning feature on the CROSSCON stack.

4.2.4 Control Flow Integrity

A novel Control Flow Integrity service is integrated within the CROSSCON Baremetal TEE. This service protects the applications from run-time attacks that exploit the corruptibility of the stack. By ensuring that the return addresses are stored in a secure memory area, it prevents ROP attacks from hijacking the application.

This secure service is integrated in the TEE and is automatically enforced on every application by instrumenting the binary. In particular, the `call` and `return` instructions are replaced by virtual calls, namely `safe_call` and `safe_ret`, that take care of storing the correct return address in the safe stack and fetching it, respectively. At run-time, the application will invoke these virtual functions transparently.

5 Specification of Proposed Architecture

This chapter defines the CROSSCON architecture, detailing the system interfaces and their role in enabling secure and efficient virtualization. It establishes how developers and runtime components interact with the CROSSCON stack, in order to provide clear guidelines for system configuration and deployment.

5.1 CROSSCON Hypervisor

This section specifies the CROSSCON Hypervisor interface. The CROSSCON Hypervisor is based on the Bao static partitioning hypervisor. Bao is a security and safety oriented, lightweight bare-metal hypervisor that prioritizes fault containment and real-time performance. Bao's implementation strives for simplicity, consisting of a minimal layer of privileged software that utilizes ISA virtualization support to create a static partitioning hypervisor architecture. In Bao, resources including CPU, memory, and I/O are allocated and partitioned when VMs are created. Memory allocation employs a two-stage translation process, while I/O operations are strictly pass-through. Virtual interrupts are directly linked to their physical counterparts, and Bao maintains a one-to-one mapping of virtual to physical CPUs, eliminating the need for a scheduler. This design approach results in a smaller TCB, enhancing security.

Additionally, Bao enables inter-VM communication through a static shared memory mechanism and asynchronous notifications using inter-VM interrupts triggered via hypervisor calls. Notably, Bao operates independently of external dependencies, such as privileged VMs running large, untrusted, monolithic General-Purpose Operating Systems (GPOS). Unlike typical hypervisors that may adapt to changing configurations while running, Bao's system configuration is established prior to its execution. The configuration is performed via editing a configuration source file, to setup hypervisor for certain behavior.

The configuration file allows developers to define various parameters, such as the number of VMs, the number of CPUs assigned to each VM, which interrupts the VMs can access, and the memory allocated to each VM. To ensure system security, developers need to have a good understanding of how to set up this configuration.

VMs interact with Bao in two different ways. First, some interactions occur through exceptions, which the hypervisor handles without the VM being aware of it. Second, VMs may use hypervisor calls to request services directly from the hypervisor.

The CROSSCON Hypervisor, extends Bao's capabilities supporting additional use cases and configurations. Bao offers only static partitioning capabilities for OSes and baremetal applications. CROSSCON Hypervisor, however, additionally supports: hosting TrustZone trusted OSes in VMs in various configurations, execution of Software Guard Extensions (SGX)-like enclaves, dynamic VM instantiation and offer multi-VMM execution support.

Static Partitioning: Bao's core functionality static partitioning, is maintained. By design, resources such as CPUs, memory, and I/O devices are allocated during the system's initialization. This approach is ideal for real-time systems and safety-critical applications, where deterministic behavior and minimal runtime variability are critical. Static partitioning ensures that no resource contention occurs between VMs, providing strong isolation and predictable performance.

TrustZone TEE Support: CROSSCON Hypervisor can emulate a TrustZone-based TEE system. This allows a VM to act as a TEE for running sensitive applications, such as cryptographic operations or secure key storage, without requiring direct hardware support. This allows deploying TrustZone software stacks,

even on platforms lacking native TrustZone support, for example the RaspberryPi4. Execution model is additionally extended to allow multiple TrustZone VMs, offering better system isolation. CROSSCON offers the ability for a single OS to benefit from two TrustZone VMs simultaneously, and for independent OSes to each feature their own independent TrustZone VMs.

Intel SGX Emulation: The CROSSCON Hypervisor can emulate Intel's Software Guard Extensions (SGX) functionality by creating secure, isolated execution environments for trusted applications. This allows developers to build and run secure enclaves for sensitive workloads on platforms that lack native SGX support. Emulating SGX provides an additional layer of security for applications requiring secure computation while preserving compatibility across hardware platforms.

Dynamic VM Instantiation and Management: The CROSSCON Hypervisor introduces extensions for dynamic VM instantiation and management, enabling the creation, configuration, and destruction of VMs at runtime. Unlike static partitioning, where VMs are predefined at boot time, this dynamic approach provides greater flexibility allowing VMs to use part of their resources to create new VMs at runtime.

Multi-VMM Support: Supporting multiple VMMs/hypervisors in a single system architecture effectively bridges the gap between high security and flexibility, which is crucial for modern mixed-criticality systems. By decoupling partitioning from virtualization and leveraging offline system orchestration, each partition can run a customized VMM tailored to its specific workload requirements. This allows safety-critical partitions to use lightweight, formally verified VMMs for stringent isolation, while less critical partitions can employ feature-rich VMMs to support dynamic resource management and advanced functionalities like device sharing or VM introspection.

The combination of these use cases ensures that the CROSSCON Hypervisor is versatile enough to support a wide range of applications, from static real-time systems to more dynamic and adaptive environments. Developers must carefully choose and configure described options in the configuration file, to ensure the solution is secure.

5.2 CROSCCON Hypervisor VM Configuration Interface

The CROSSCON Hypervisor's configuration is managed through a dedicated configuration in the form of a C source file. This section provides an in-depth description of the various configuration options available.

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`.

```
#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 },
        ...
    }
};
```

Warning. Inconsistencies between the specified list sizes (`shmemlist_size` and `vmlist_size`) and the actual sizes of their respective lists may result in unpredictable behavior. Ensure that any changes made to the configuration lists' number of elements are reflected in the respective list size.

When embedding VM images into the hypervisor binary it is necessary to declare the VM images using the `VM_IMAGE` macro. 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 (see Guest Image).
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.

5.2.1 VM Configuration

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. It also defines the guest image that each VM will execute. In CROSSCON Hypervisor, all resources are exclusively allocated, ensuring strict isolation between VMs. Additionally, inter-VM communication is explicitly configured through shared memory regions or dedicated communication links, allowing controlled data exchange while maintaining security and isolation.

Each entry in the `vmlist` mentioned earlier is a `vm_config` struct, which defines the configuration of each individual guest:

Each entry in this list represents a unique VM configuration, defining its image, memory address, CPU

```
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;
};
```

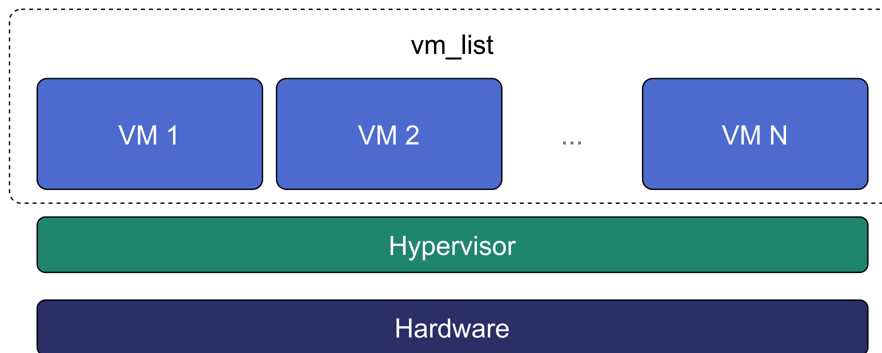


Figure 10: Guest Configuration

affinity, color mapping, and platform details. For each VM, the following parameters must be specified:

- `image` [mandatory] - a structure containing information about guest image loading (see details in Guest Image).
- `entry` [mandatory] - defines the entry point address in the guest's address space.
- `platform` [mandatory] - a description of VM platform: resource assignments and requirements (see details in Virtual Machine Configuration).
- `cpu_affinity` [optional] - defines the affinity of VM's vCPUs to the physical CPUs assigned to the virtual platform. The affinity will be respected to achievable extent, but direct match is not guaranteed (see details in CPU Affinity).
- `colors` [optional] - assignment of shared LLC cache colors (or partitions) to this VM (see details in Coloring).
- `type` - Describes the type of VM. Types include `CROSSCON_VM_ORDINARY`, `CROSSCON_VM_SDTZ`, `CROSSCON_VM_SDSGX`, and `CROSSCON_VM_DYNAMIC`. If no value is defined, `type` is

CROSSCON_VM_ORDINARY.

- children_num - Number of child VMs.
- children - VMs can be configured in a execution control hierarchy, establishing parent<->child relationship. This is used for sdTZ (Sec. 5.2.9), sdSGX (Sec. 5.2.10) and dynamic VM (Sec. 5.2.11) management. This field enables VMs to statically declare multiple children in an array.

5.2.2 Guest Image

The guest image comprises a structure that describes the image configuration running on the guest side. It encompasses the following options:

- image [mandatory] - definition of the image to run on a given VM. The image corresponds to the following structure:

```
struct vm_image {
        vaddr_t base_addr;
        paddr_t load_addr;
        size_t size;
        bool separately_loaded;
        bool inplace;
} image;
```

where:

- 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.

To ensure accurate and efficient configuration of VM images, 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. Attempting to manually configure image details without utilizing these macros may result in errors or undefined behavior.

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. For more details, refer to Figure 11.

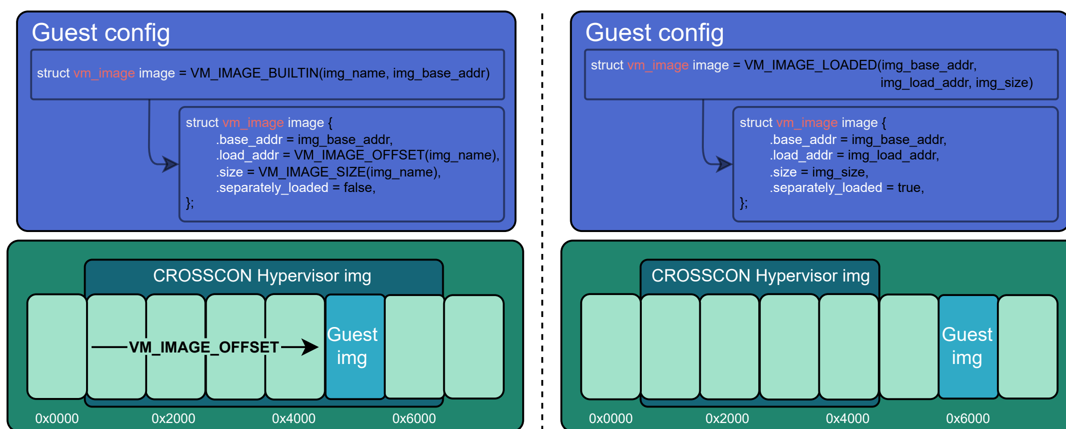


Figure 11: VM Image Configuration

5.2.3 Virtual Machine Configuration

The VM configuration enables users to define the characteristics of each virtualized platform. The virtual machine configuration is performed by populating the structure struct vm_platform, outlined below:

```
struct vm_platform {
    size_t cpu_num;
    size_t region_num;
    struct vm_mem_region* regions;
    size_t ipc_num;
    struct ipc* ipcs;
    size_t dev_num;
    struct vm_dev_region* devs;
    bool mmu;
    struct arch_vm_platform arch;
};
```

By customizing this configuration, users can set up virtual platform to suit specific workload requirements and application needs for their virtual machines. The configuration includes the definition

of:

- Number of CPUs - see details in Number of vCPUs.
- Memory regions - see details in Memory Regions.
- Inter-Partition Communication (IPC) - see details in Inter-Partition Communication (IPC).
- Devices - see details in Devices.
- Architectural-Specific Configurations - see details in Architectural-Specific Configurations.

5.2.3.1 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.

5.2.3.2 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.

5.2.3.3 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.

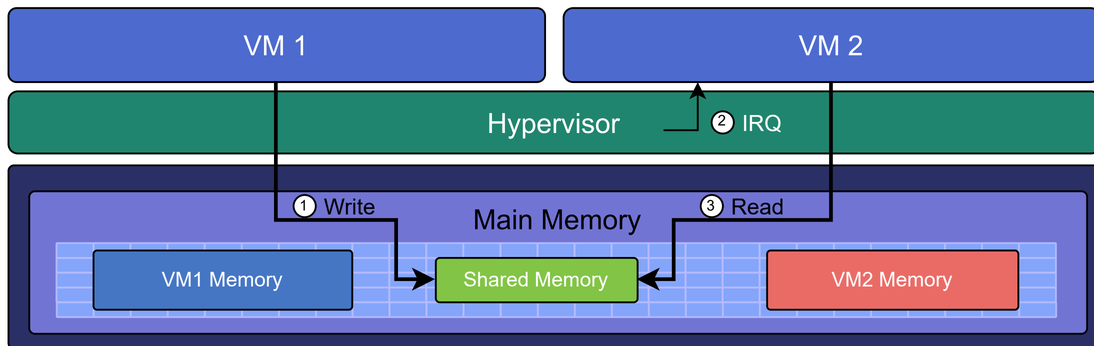


Figure 12: Inter-Partition Communication (IPC)

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] - 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 behavior.

5.2.3.4 Devices

- dev_num [mandatory] - corresponds to the number of device MMIO regions assigned to the VM.
- devs [mandatory if dev_num > 0] - corresponds to the specification of the VM's devices and is configured through the following 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 physical 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 virtual address (va) must match the physical address (pa).

- interrupt_num [optional] - corresponds to the number of assigned interrupts. 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.

5.2.3.5 Architectural-Specific Configurations

- arch - allows defining architecture-dependent configurations and is configured through the following structure:

```
struct arch_vm_platform {
    // Architecture-specific fields
};
```

The specific fields depend on the architecture being used (e.g., Arm, RISC-V). Refer to the architecture-specific documentation for details.

Arm Architecture

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 schema.

RISC-V Architecture

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

```c
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 incorrect behavior, as they are typically managed by Bao 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.

5.2.4 CPU Affinity

The configuration file structure of the CROSSCON Hypervisor also enables the definition of core affinity, which involves selecting the physical core where the guest should run.

This functionality is achieved through the following configuration parameter:

- `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

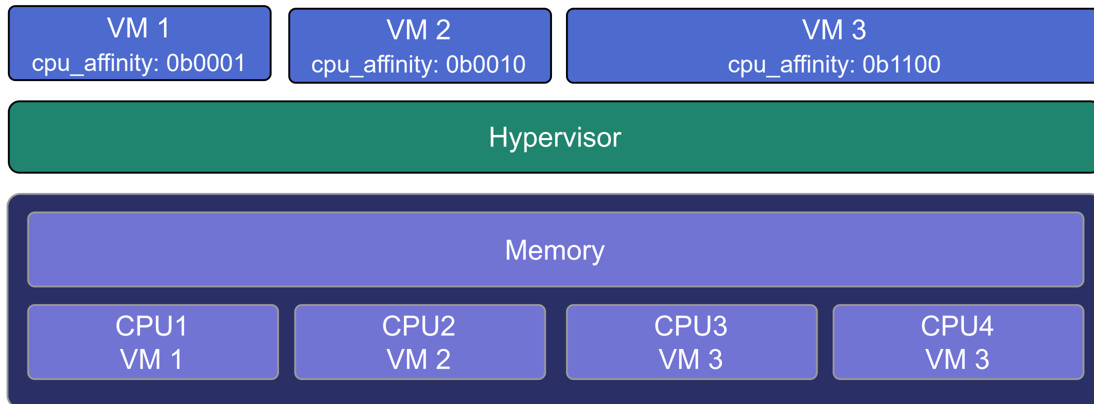


Figure 13: CPU Affinity Configuration

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.

5.2.5 Coloring

Cache coloring is a technique used to partition shared Last Level Cache (LLC) sets among different guests (i.e., VMs). The main goal is to minimize cache conflicts and enhance overall system performance by carefully assigning specific colors to different entities. Each color represents a certain cache set. CROSS-CON Hypervisor supports cache coloring, which can be configured using the `colors` field within the `vm_config` struct.

- `colors` [optional] - 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.

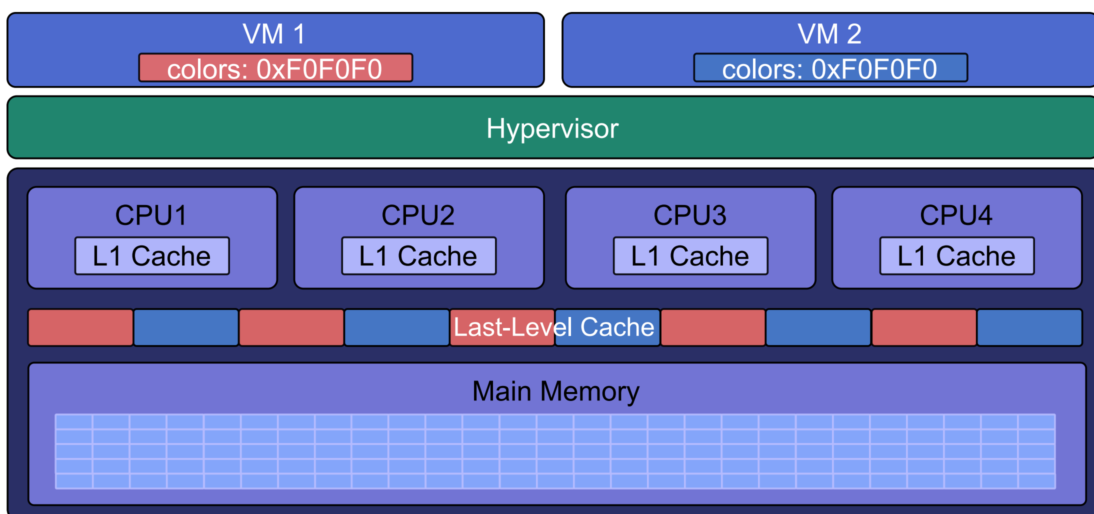


Figure 14: LLC Coloring Configuration

It is important to note that 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.

5.2.6 Shared Memory Configuration

In CROSSCON Hypervisor's configuration, one can set multiple shared memory regions (e.g., `shmem0`, `shmem1`, ...) or none at all. An ID is assigned to each shared memory object. Later, this ID can be linked to an IPC in the multi-guest configuration.

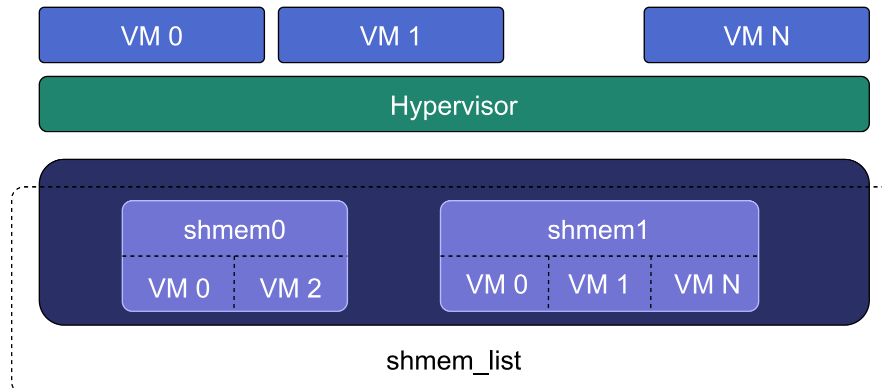


Figure 15: Shared Memory Configuration

The configuration of shared memory is done by populating a struct called `shmem`. The struct definition is as follows:

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

The struct contains the following parameters:

- `size` [mandatory] - defines the total size of the shared memory. The size of the shared memory must always be multiples of 4KB (0x1000).
- `place_phys` [optional] - This flag determines the mapping of shared memory on a virtual memory. When `place_phys` is set to `false` (default value), the base address is a guest physical address. When set to `true`, the guest address is the same as the physical address (PA) of the shared memory.
- `base / phys` [optional] - When `place_phys` is set to `false`, the `base` parameter should be used to specify the guest virtual address of the shared memory. When `place_phys` is `true`, the `phys` parameter should be used to specify the physical address of shared memory.

5.2.7 Configuration File Location

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-name
```

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.

5.2.8 Static Partitioning 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.

```c
#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 VM1 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.

5.2.9 Virtual TZ TEE 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>

// Linux Image
VM_IMAGE(linux_image, "<path-to-linux>/linux.bin");

// Linux VM configuration
struct vm_config linux = {
    .image = VM_IMAGE_BUILTIN(linux_image, 0x40200000),
    .entry = 0x40200000,

    .platform = {
        .cpu_num = 1,
        .region_num = 1,
        .regions = (struct mem_region[]) {
            {
                .base = 0x40000000,
                .size = 0x30000000,
            }
        },
        .ipc_num = 1,
        .ipcs = (struct ipc[]) {
            {
                .base = 0x70000000,
                .size = 0x00200000,
                .shmem_id = 0,
            },
        },
        .dev_num = 2,
        .devs = (struct dev_region[]) {
            {
                .pa = 0x9000000,
                .va = 0x9000000,
                .size = 0x10000,
                .interrupt_num = 1,
                .interrupts = (irqid_t[]) {33}
            },
            {
                .interrupt_num = 1,
                .interrupts = (irqid_t[]) { 27 }
            },
        },
        .arch = {
            .gic = {
                .gicc_addr = 0x8010000,
                .gicd_addr = 0x8000000,
                .gicr_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, // 15 MB
            }
        },
        .ipc_num = 1,
        .ipcs = (struct ipc[]) {
            {
                .base = 0x70000000,
                .size = 0x00200000,
                .shmem_id = 0,
            }
        },
        .dev_num = 2,
        .devs = (struct dev_region[]) {
            {
                // PL011
                .va = 0x9040000,
                .pa = 0x9000000,
                .size = 0x10000,
            },
            {
                // Arch timer
                .interrupt_num = 1,
                .interrupts = (irqid_t[]) {27,40}
            }
        },
        .arch = {
            .gic = {
                .gicc_addr = 0x8010000,
```

```
                .gicd_addr = 0x8000000,
                .gicr_addr = 0x80A0000,
            }
        }
    },
};

struct config config = {

    CONFIG_HEADER
    .shmemlist_size = 1,
    .shmemlist = (struct shmem[]) {
        [0] = { .size = 0x00200000, },
    },
    .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), and 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.

5.2.10 Virtual SGX Example

The following configuration file defines a secure enclave VM designed to run an isolated workload using a single VM on an Arm-based platform. The setup is structured to support an environment, in which an SGX-like enclave operates in an isolated memory region with strict access controls. This configuration ensures that sensitive computations are protected from the host system and any potentially untrusted software.

```
#include <config.h>

VM_IMAGE(enclave, "<path-to-enclave-binary>/crosscon_sgx_enclave.bin");

struct vm_config enclave_vm = {

    .image = VM_IMAGE_BUILTIN(enclave, 0x40000000),
    .entry = 0x40000000,

    .type = CROSSCON_VM_SDSGX,

    .platform = {
        .cpu_num = 1,

        .region_num = 1,
        .regions = (struct mem_region[]) {
            {
                .base = 0x40000000,
                .size = 0x07fe0000
            },
        },
    },
};

struct config config = {
    .vmlist_size = 1,
    .vmlist = { &enclave_vm },
};
```

The configuration specifies a single VM image, named `enclave`, which is loaded from `<path-to-enclave-binary>/crosscon_sgx_enclave.bin`. The binary is integrated using the VM_IMAGE macro, ensuring that it is properly linked into the VM. The enclave VM is assigned a base address of 0x40000000, and the macros VM_IMAGE_OFFSET and VM_IMAGE_SIZE are used to determine the image's load address and size. Execution begins at 0x40000000, making it the entry point for the enclave. A distinctive feature of this configuration is the VM type, which is set to CROSSCON_VM_ENCLAVE, indicating that it operates in a specialized enclave mode. This configures the VM to support an SGX-like environment. The enclave VM is allocated one virtual CPU. It is also assigned a single memory region of approximately 127.9MB (0x07FE0000), starting at 0x40000000. Device access is restricted, as no devices are accessible by the enclave.

This config file must be compiled into a config file. The resulting binary (e.g., enclave.bin) is then used by an application linked with the modified SGX SDK to instantiate the enclave, in the same way it is done for typical SGX untrusted applications.

5.2.11 Dynamic VM Example

The following configuration defines a VM to be created dynamically.

```
#include <config.h>

VM_IMAGE(dynamic, "<path-to-enclave-binary>/dynamic-linux.bin");

struct vm_config dynamic_vm = {

    .image = VM_IMAGE_BUILTIN(dynamic, 0x40000000),

    .entry = 0x40000000,

    .type = CROSSCON\_VM\_DYNAMIC,

    .platform = {
        .cpu_num = 1,

        .region_num = 1,
        .regions = (struct mem_region[]) {
            {
                .base = 0x40000000,
                .size = 0x20000000
            },
        },

        .ipc_num = 1,
        .ipcs = (struct ipc[]) {
            {
                .base = 0x70000000,
                .size = 0x00200000,
                .shmem_id = 0,
            }
        },

        .dev_num = 1,
        .devs = (struct dev_region[]) {
            {
                .pa = 0x09000000,
                .va = 0xFF010000,
                .size = 0x10000
            },
        },
    },
};

struct config config = {
    .vmlist_size = 1,
    .vmlist = { &dynamic_vm },
};
```

This configuration defines a single VM, named `dynamic_vm`, which loads its binary from `<path-to-enclave-binary>/dynamic-linux.bin`. The image is mapped to memory at 0x40000000, and execution begins at the same address. The VM type is specified as CROSSCON_VM_SDSGX, indicating that it is a dynamic VM, potentially supporting runtime resource allocation or secure interactions with other system components.

The VM is assigned a single virtual CPU and a dedicated memory region of 128MB (0x08000000), starting from 0x40000000. The execution entry point is set to 0x40200000, ensuring that the VM starts execution from within its protected memory space.

The VM has access to a 2MB shared memory region, mapped at 0x70000000, which allows it to exchange data securely with other system components. The shared memory is identified as `shmem_id = 0`, indicating that it is the first shared memory region allocated.

Device access is restricted to one mapped device region. The VM is given access to a PL011 device at physical address 0x09000000.

This config file must be compiled (e.g., dynamicvm.bin) and passed as an argument to the hypervisor, for example, by using the Dynamic VM CROSSCON Hypervisor API 6.1.2.

5.2.12 Trap and Emulation

A trap occurs when a virtualized guest OS executes a privileged instruction or attempts to access memory regions of the VM's range. Instead of allowing the VM to directly execute the instruction or perform the memory access, the hypervisor steps in and manages the execution of the privileged operation in a controlled manner. This process is referred to as emulation. It involves simulating or replicating functionality that is not directly available in a virtualized environment. The hypervisor emulates the required functionality, enabling the VM to function as if it had direct access to the resource or interface. Through trap and emulation, the hypervisor transparently interrupts the VM execution to mediate interaction with the system. In the case of the CROSSCON Hypervisor, three scenarios require trap and emulation: firmware calls, system register access, and MMIO, e.g., to emulate devices.

Firmware Call Emulation - Arm and RISC-V platforms feature an operation mode that provides firmware-level functionality (e.g., power management). In Arm systems, this mode is often referred to as "Secure Monitor," while in RISC-V, the firmware mode is known as "Machine Mode." The firmware modes on Arm and RISC-V serve similar goals in that certain platform-specific details are accessible through their respective interfaces. Secure Monitor Call (SMC) [25] in the case of Arm and the Supervisor Binary Interface (SBI) accessed through an "ECALL" in RISC-V [26].

- Arm SMC emulation - Arm SMC [25] emulation is a crucial aspect of the hypervisor's role in managing virtualized environments on Arm-based platforms. The SMC instruction allows a virtual machine to request services from a secure monitor. The hypervisor intercepts these calls from the VMs and emulates or mediates their interaction with the underlying firmware if appropriate. The CROSSCON Hypervisor supports transparent handling of the Arm's Power State Coordination Interface (PSCI) [27].
- RISC-V SBI emulation - In the case of RISC-V, the firmware mode is known as "Machine Mode", and it hosts firmware often referred to as the "SBI" (Supervisor Binary Interface) that, similarly to the secure monitor, serves as the interface between machine mode and supervisor or hypervisor modes. When a VM attempts to execute an SBI call, the hypervisor intervenes, mediating or replicating the behavior and functionality of the requested SBI operation. The following interfaces are supported: Base, Time, IPI, RFence, and HSM. For more details see [26].

System Register Emulation - System registers serve specific control, configuration, and status-monitoring purposes for CPU cores and other hardware components. System registers provide a way for

software, including OSes, to manage various aspects of the processor's behavior and operation. They often control features like memory management, interrupt handling, cache configuration, power management, and mode switching. The content of system registers is accessed by privileged software, such as the OS or the hypervisor, in order to manage the system's behavior and maintain security and isolation between different software components or virtual machines. Some system registers can only be accessed at specific privilege levels, ensuring that critical settings are not easily altered by user-level applications or guest OSes. Accesses to system registers need to be decoded for the type of operation (read or write) and must be handled accordingly.

- Arm System Register emulation - In Arm, the CROSSCON Hypervisor controls access to GICv3 system registers ICC_SGIR and ICC_SRE. For more information, see [9] [28].
- RISC-V System Register emulation - In RISC-V, the CROSSCON Hypervisor does not trap any system registers.

MMIO emulation - In MMIO emulation, the hypervisor traps memory accesses to unmaped MMIO regions to implement virtualization functionality (e.g., sharing a device between guests). Same as for system register emulation, memory accesses need to be decoded for the type of operation (read or write).

- Arm MMIO emulation - The hypervisor emulates the interrupt controller in Arm platforms. The emulation of Arm's Generic Interrupt Controller (GIC) [29] enables the hypervisor to manage and process interrupts by driving the behavior of the physical GIC without VMs being able to interfere with each other's interrupts. Currently, there is support for GICv2 and GICv3 interrupt controllers.
- RISC-V MMIO emulation - The hypervisor emulates the interrupt controller in RISC-V platforms. Similar to Arm's GIC emulation, emulating RISC-V's Platform-Level Interrupt Controller (PLIC) [30] and Advanced Interrupt Architecture [31] enables the hypervisor to manage and process interrupts without VMs being able to access each other's interrupts. Currently, there is support for the PLIC interrupt controller.

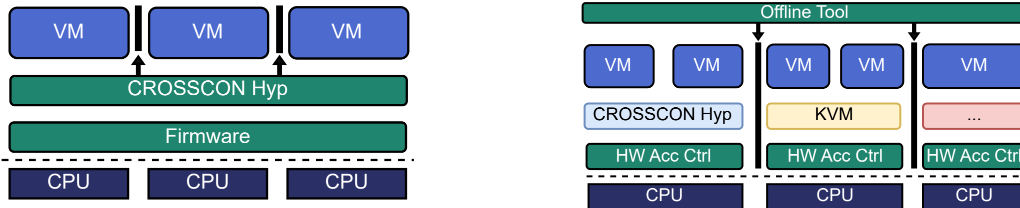
5.2.13 Hypercalls

The hypervisor call interface, henceforth the hypercall interface, enables VMs to directly request services from the hypervisor. A VM will issue hypercalls using dedicated instructions. In some systems (ArmV7-A ArmV8-A and RISC-V), the hypercall interfaces are similar to system calls issued by an application to the OS. In other systems (Armv8-M) this is achieved using secure gateway entrypoints.

Hypercalls provide a controlled pathway for VMs to access hypervisor functionalities. In the CROSSCON Hypervisor, the available hypercall interface provides access to IPC communication, i.e., communication between VMs, SGX like features, and dynamic VM instantiation. The ABI for issuing hypercalls in Armv7-A/Armv8-A is defined in SMCCC[32], in the RISC-V the interface is defined in SBI Specification [19], while for Armv8-M we define our own to be described in Deliverable D3.3 .

5.3 CROSSCON Multi-VMM Support

Supporting multiple VMMs or hypervisors within a single system architecture is a critical step toward addressing the dual needs of high security and flexibility in modern Mixed-Criticality Systems(MCSs). Traditional SPHs provide strong isolation and real-time guarantees, essential for safety-critical applications, including automotive systems or industrial control. However, their rigid resource allocation and lack of dynamic management capabilities make them inefficient for handling feature-rich, lower-criticality workloads, such as infotainment systems or general-purpose computing tasks. By enabling multiple VMMs, each tailored to specific partitions or subsystems, the system can achieve a balance between



(a) Typical virtualization architecture with one VMM.

(b) CROSSCON Multi-VMM architecture.

Figure 16: Comparision of typical virtualization scenario with one VMM and CROSSCON multi-VMM architecture.

stringent safety requirements and the flexibility needed for diverse applications.

Figure 16 presents two architectures. One with a common hypervisor scheme, i.e., single VMM based architecture 16a, where the hypervisor and firmware must be implemented correctly to guarantee isolation between VMs. The other 16b, where a multi-VMM architecture allows multiple VMMs to execute isolated from each other. Critically, VMs operating under one VMM cannot be affected by other compromised VMMs. The key to supporting multiple VMMs lies in decoupling the partitioning functionality from the virtualization layer. In our proposed architecture, Figure 16b, the partitioning is handled by an offline tool, which pre-computes and optimizes resource allocation for each partition. At runtime, hardware access control layers enforce these partitions, ensuring spatial and temporal isolation. This approach allows each partition to run its own VMM, which can be independently customized and certified according to the specific requirements of the workload it hosts. For example, a safety-critical partition running an ASIL-D workload could use a lightweight, formally verified VMM like seL4, while a less critical partition hosting a QM workload could employ a more feature-rich VMM like KVM. This heterogeneity ensures that each subsystem operates with the appropriate level of security and functionality without compromising the overall system's safety guarantees. Moreover, the use of multiple VMMs enhances fault containment and system resilience. By isolating each VMM within its own partition, the system prevents potential faults or vulnerabilities within a VMM from impacting others. This is particularly important in MCSs, where a failure in a non-critical subsystem should not jeopardize the operation of safety-critical components. Additionally, the ability to run different VMMs on different partitions allows for the integration of advanced features such as device sharing, dynamic resource management, and VM introspection, which are typically precluded in traditional SPHs. These features are essential for supporting complex, feature-rich applications while maintaining the system's overall security and real-time performance.

To implement this architecture, Same-Privilege Isolation (SPI) techniques on Arm and hardware primitives like ePMP on RISC-V can be used. If used correctly, these mechanisms ensure that VMMs running at the hypervisor level are isolated from one another, even though they operate at the same privilege level. This approach eliminates the high latencies associated with microkernel-based VMMs, where frequent context switches between the microkernel and VMMs can degrade performance. By allowing VMMs to execute directly in hypervisor mode, the architecture achieves low-latency, high-performance virtualization while maintaining strong isolation guarantees.

5.4 CROSSCON Baremetal TEE

In this section we describe the configuration process for the Baremetal TEE, in both the available versions (with and without MPU), the specific TA configuration, and the configuration of the CAs.

5.4.1 Baremetal nonMPU

The BareMetal TEE nonMPU is a lightweight security solution designed for low-end devices that lack an MPU. As a minimal TEE, it operates with a fixed, static configuration that cannot be modified. Deploying an application is straightforward: the source code must be compiled using the provided custom toolchain. No additional configuration is required. The compilation process is simple, involving a single Makefile invocation.[1]

5.4.2 Baremetal MPU

The BareMetal TEE MPU is a solution for memory isolation and supervised execution aimed at increasing the security of resource-constrained devices in a way that is mostly transparent to application developers. The architecture is fully compliant with the Global Platform TEE Client API specification [33] and with a subset of the TEE Core API. The BareMetal TEE MPU supports the deployment of up to two custom TAs compliant with the Global Platform TEE Core API and up to one untrusted CA.

5.4.2.1 TEE Configuration

The TEE itself does not need any configuration. When the deployed CA needs network connectivity, the only requirement is to provide a configuration for network access. More specifically, the Wi-Fi SSID and password must be set to establish a connection. These data can be specified in the file `microvisor_config.h` which can be found in the `OS/Microvisor` folder.

5.4.2.2 Trusted Application Configuration

The TAs that can be included in the Baremetal TEE for use by Client Applications are fully customizable. However, since TAs are compiled alongside TEE, it is necessary to perform multiple steps to have them successfully deployed. As a prerequisite, each TA must implement the TEE Client API specified by Global Platform [33].

Among the configurations required for the TA, the following must be set.

1. Insert the TA code inside the `OS/TA` directory of the TEE.
2. Modify the `Makefile` of the TEE by specifying the TA entry point and the TA source files.
3. Modify the linker script `BOOTLOADER.ld` to map the TA's code and BSS data sections to their designated memory regions.
4. Include in the header files of the TA the correct directives to support the Global Platform Core and Client API specification (e.g., `tee_client_api.h` and `tee_core_api.h`).
5. Recompile and redeploy the Baremetal TEE to apply the modifications and use the updated TA.

These steps can be repeated for both TA slots available in the Baremetal TEE MPU.

5.4.2.3 Client Application Configuration

Building a Client Application (CA) that runs under the supervision of the Baremetal TEE and utilizes the services provided by the TAs and the TEE core requires several steps and configurations. The essential ones are presented below.

1. Modify the `Makefile` of the CA to enable the instrumentation of the application sources during the compilation process (using the `INSTRUMENTER_PATH` variable). This ensures that the application runs without privileges and under the supervision of the TEE.

[1] The official repository includes a README.md file with detailed Makefile commands.

2. Edit the linker script of the CA (STM32L475VGTX_FLASH.ld) to align with the memory map defined for the TEE, ensuring that the code and data are stored and positioned in the predefined segments.
3. Edit the debug configuration of the CA to launch the TEE and TAs binaries before switching to the execution of the unprivileged client application.

After completing these steps, the CA can be executed. To interact with the TAs included in the TEE, both the tee_client_api.c, tee_client_api.h, and tee_common.h files should be included and compiled along with the CA.

6 Specification of CROSSCON API

This section describes the specification for CROSSCON APIs. It explores topics such as the APIs for the CROSSCON Hypervisor, secret key management, cryptographic operations, measurements, device identification, PUF, remote attestation, control flow integrity, secure update and secure FPGA provisioning.

6.1 Hypervisor API

This section documents the APIs that applications use to interact with the Hypervisor.

6.1.1 Inter Partition Communication (IPC)

IPC enables inter-partition and inter-VM information exchange. This communication mechanism allows VMs to interact and exchange data with each other while maintaining strong isolation boundaries.

API Overview

API Name: IPC

API Version: 1.0

Description: This API, allows CROSSCON Hypervisor guests which have been configured with IPC capabilities to notify other guests participating in the same IPC channel about an IPC event.

API Specification

1. **Function: Write**

   Function ID: crosscon_ipc_write

   Description: Writes to an offset on the shared memory.

   Input Parameters

   Input Parameter 1

   Name: ipc_id

   Type: unsigned long

   Direction: IN

   Description: The shared memory ID for an IPC channel

   Constraints: Value must be one of the shared memory ids defined in the config file.

   Input Parameter 2

   Name: buffer

   Type: char *

   Direction: IN

   Description: Buffer write to.

   Input Parameter 3

Parameter Name: n

Type: size_t

Direction: IN

Description: number of bytes to write.

Return Value

Type: unsigned long

Description: Error code

Possible Values: `SUCCESS`,`ERROR_ACCESS_DENIED`, or `ERROR_BAD_PARAMETERS`.

Preconditions

Guest OS must have the CROSSCON Hypervisor IPC driver loaded and must be configured with IPC channels.

2. **Function: Read**

Function ID: crosscon_ipc_read

Description: Reads shared memory region from an offset.

Input Parameters

Input Parameter 1

Parameter Name: ipc_id

Type: unsigned long

Direction: IN

Description: The shared memory ID for an IPC channel

Constraints: Value must be one of the shared memory ids defined in the config file.

Input Parameter 2

Parameter Name: buffer

Type: char *

Direction: IN

Description: Buffer to read shared memory contents to.

Input Parameter 3

Parameter Name: n

Type: size_t

Direction: IN

Description: number of bytes to read

Return Value

Type: unsigned long

Description: Error code

Possible Values: `SUCCESS`,`ERROR_ACCESS_DENIED`, or `ERROR_BAD_PARAMETERS`.

Preconditions

Guest OS must have the CROSSCON Hypervisor IPC driver loaded and must be configured with IPC channels.

3. **Function: Notify**

Function ID: crosscon_ipc_notify

Description: Notify IPC channel participants.

Input Parameters

Input Parameter 1

Parameter Name: ipc_id

Type: unsigned long

Direction: IN

Description: The shared memory ID for an IPC channel

Constraints: Value must be one of the shared memory ids defined in the config file.

Input Parameter 2

Parameter Name: event_id

Type: unsigned long

Direction: IN

Description: What event to use to notify IPC channel participants

Constraints: The value must be in the range $[0, N-1]$, where N is the number of available events (i.e., interrupts) in the IPC channel.

Return Value

Type: unsigned long

Description: Error code

Possible Values: `SUCCESS`,`ERROR_ACCESS_DENIED`, or `ERROR_BAD_PARAMETERS`.

Preconditions

Guest OS must have the CROSSCON Hypervisor IPC driver loaded and must be configured with IPC channels.

Error Codes

`SUCCESS`: Operation completed successfully

`ERROR_ACCESS_DENIED`: Permission denied - accessing the driver failed or the driver does not exist

`ERROR_BAD_PARAMETERS`: One or more parameters are incorrect.

Security Considerations

Privileges Required: This API requires access to the `/dev/crossconhyp@ipc<n>` filesystem nodes.

Data Sensitivity: No sensitive data is accessed by the API.

Potential Threats: By abusing this API could an attacker may be able to trigger high number of interrupts, which might disrupt other virtual machines and cause a Denial of Service (DoS).

6.1.2 Dynamic VM

The CROSSCON Hypervisor adds new features to that allow virtual machines to be created, configured, and removed while the system is running, making VM management more dynamic and flexible. The Dynamic VM API allows a host VM to create, invoke and destroy dynamic VMs. Communication between host and dynamic VMs is established using the hypervisor's IPC mechanism.

API Overview

API Name: VM Host

API Version: 0.1

Description:This API allows CROSSCON Hypervisor guests to create and manage dynamic VMs.

API Specification

1. **Function: VM Create**

   Function ID: crosscon_vm_create

   Description: Dynamically create a VM according to a configuration file.

   Input Parameters

   Input Parameter 1

   Parameter Name: config_file

   Type: void*

   Direction: IN

   Description: The address of config struct.

   Constraints: When creating a new VM, both the config file and the memory assigned to the VM must be placed in a single, continuous block within the guest's physical address space.

   Return Value

   Type: unsigned long

   Description: Result

   Possible Values: vm_handl number, INVALID_ARGUMENTS: One or more parameters are incorrect.

   Preconditions

   VM config must reside in contiguous guest physical address space.

   Post-conditions

   The memory occupied by the config binary and the subsequent memory allocated for the VM will be unmapped from the host VM. The VM image and memory will be mapped to the enclave VM.

2. **Function: VM INVOKE**

   Function ID: crosscon_vm_invoke

   Description: Invoke a dynamically created VM.

   Input Parameters

   Input Parameter 1

Parameter Name: vm_handle

Type: (void*)

Direction: IN

Description: Handle the VM to execute

Constraints: The handle must correspond to a VM.

Return Value

Type: unsigned long

Description: Return.

Possible Values: INVALID_ARGUMENTS, SUCCESS

Preconditions

VM config must reside in contiguous guest physical address space. Guest must have a permission to create VMs.

Post-conditions

The memory occupied by the config binary and the subsequent memory allocated for the VM will be unmapped from the host VM. The VM image and memory will be mapped to the enclave VM.

3. **Function: VM Delete**

Function ID: crosscon_vm_delete

Description: Delete a dynamic VM from memory and reclaim access to the memory lent to the dynamic VM.

Input Parameters

Input Parameter 1

Parameter Name: vm_handle

Type: unsigned long

Direction: IN

Description: The handle of the dynamic VM to delete.

Constraints: The handle must correspond to a VM.

Return Value

Type: unsigned long

Description: Success of the operation

Possible Values: SUCCESS, INVALID_ARGUMENTS.

Preconditions

Enclave VM config must reside in contiguous guest physical address space.

Error Codes

INVALID_ARGUMENTS

SUCCESS

Security Considerations

Privileges Required: This API is exposed to the Guest OS.

Data Sensitivity: No sensitive data is accessed by the API.

6.1.3 Virtual TZ TEE

Armv8-A. On Arm8-A there are no CROSSCON Hypervisor specific interfaces provided to interact with Virtual TZ TEE. Guests should use the same SMC interface [32] as they would use to interact with the platform's TEE. In practice, this means that registers x0-x5 are used to interact with the TEE. For multiple TEEs CROSSCON Hypervisor differentiates between TEE instances based on the SMCCC owner bits [32]. Currently CROSSCON Hypervisor supports two OP-TEE instances with one OP-TEE SMCCC Owner between 0x32 and 0x3f, and the second with SMCCC Owner between 0x12 and 0x1F.

RISC-V (MHvSvU). On RISC-V (MHvSvU) the same ABI as on Arm is being adapted for interacting with the TEE. The only difference is that registers A7 and A6 are used to identify a TEE specific call.

6.1.4 Virtual SGX TEE

Armv8-A. On Arm8-A there are no CROSSCON Hypervisor specific interfaces provided to interact with enclaves. This is because a modified version of the SGX SDK is used. Additionally, for the enclave a custom enclave runtime has been developed. This allows for running unmodified enclaves and corresponding untrusted applications apart from architecture specific details.

6.2 Secure Storage APIs

To securely store secret keys, the GlobalPlatform TEE Internal Core API Specification was utilized, as it is a widely recognized standard for trusted execution environments. The standardized API set is described in Section 5, "Trusted Storage API for Data and Keys", of the GlobalPlatform official documentation [5] and includes, among others, the following API calls.

- Create a persistent object, given an initial datastream and attributes (TEE_CreatePersistentObject, section 5.7.2). The object is created in the secure private storage of the Trusted Application. Access rights and permissions can be set using flags.
- Open a persistent object, given the handle of an already created object (TEE_OpenPersistentObject, section 5.7.1).
- Close a persistent object, given the handle of an existing object (TEE_CloseObject, section 5.5.5) or close and delete an object given the handle (TEE_CloseAndDeletePersistentObject1, section 5.7.4).
- Read the content of the object associated with that handle (TEE_ReadObjectData, section 5.9.1).
- Write (or overwrite) the content of the object specified by the handle with the content provided (TEE_WriteObjectData, section 5.9.2).

Other APIs for object handling are included in the standard, but fall outside the scope of secret key storage.

All these functions pertain to the use of persistent objects, meaning that the data is stored persistently, and its lifecycle is independent of the CA that created it. Secret keys typically rely on these types of objects, as their value is needed across multiple sessions. Additionally, the standard provides APIs for transient objects, which are stored in memory and have a limited lifecycle.

6.3 Cryptographic operations

To provide an interface for cryptographic operations, the GlobalPlatform TEE Internal Core API Specification was selected, as it is a widely recognized standard for trusted execution environments. The APIs are defined in Section 6, "Cryptographic Operations API", of theGlobalPlatform official documentation [5] and are classified based on the type of operation to be performed. Additionally, each category requires a specific set of operations to properly handle cryptographic processes.

In particular, before being able to use any cryptographic primitives, the operation should be allocated (TEE_AllocateOperation, section 6.2.1) and the key initialized (if needed, with TEE_SetOperationKey, section 6.2.6). After the operation is complete, the allocated operation should be freed (TEE_FreeOperation, section 6.2.2).

The different types of operation available are listed below.

1. Symmetric Cipher Functions: the functions are used for both encryption and decryption
    - 1.1. Initialize the cipher suite with the proper initialization vector, based on the algorithm used (TEE_CipherInit, section 6.4.1)
    - 1.2. Iterate on the data and compute the ciphertext, updating it one piece at the time (TEE_CipherUpdate, section 6.4.2)
    - 1.3. Finalize the result of the ciphertext (TEE_CipherDoFinal, section 6.4.3)
2. MAC functions
    - 2.1. Initialize the MAC data structure with the proper initialization vector, if needed (TEE_MACInit, section 6.5.1)
    - 2.2. Iterate on the data and compute the MAC, updating it one piece at the time (TEE_MACUpdate, section 6.5.2)
    - 2.3. Finalize the result of the MAC computation (TEE_MACComputeFinal, section 6.4.3)
3. Message Digest functions
    - 3.1. Compute the digest of a given text (TEE_DigestUpdate, section 6.3.1)
    - 3.2. Finalize the digest computation of the text (TEE_DigestDoFinal, section 6.3.2)
    - 3.3. Extract the current digest without finalizing the digest computation (TEE_DigestExtract, section 6.3.3)
4. Asymmetric functions for "Sign and Verify" operations
    - 4.1. Sign a message digest with an asymmetric encryption operation (TEE_AsymmetricSignDigest, section 6.7.2)
    - 4.2. Verify a message digest signature within an asymmetric operation. (TEE_AsymmetricVerifyDigest, section 6.7.3)

6.4 Additional measurements in attestation reports (besides default measurements)

The Remote Attestation (RA) implementation of the CROSSCON stack will be configured to attest certain predefined memory regions of a selected target VM. As such, no distinct differences between "measurement types" can be made, as the RA service does not make a distinction between what type of data is

stored within the defined memory range. As such, a distinction between different measurement types can no longer be made, as adapting the to-be-attested memory range will imply a change in measurement.

A detailed discussion and example implementations on how different measurement types can be achieved by changing the memory ranges will be explored as part of deliverables D5.5 and D5.7, which will detail the integration of the Use Cases and the CROSSCON stack integration and usage guidelines, respectively.

6.5 The device's unique identifier

API Overview

API Name: CROSSCON Unique ID

API Version: 0.1

Description: This API allows Apps to request the platforms unique ID, i.e., TA UUID;

API Specification

1. **Function: Get CROSSCON Unique ID**

   TA Function ID: TA_CROSSCON_GET_UNIQ_ID

   Description: A clear and concise explanation of what this API function does, including its use cases.

   Module/Service Name: Name the specific module or service this API is part of (e.g., Trusted Application (TA) Management, Cryptographic Services, etc.).

   Input Parameters

   Input Parameter 1

   Name: unique_id

   Type: TEE_PARAM_TYPE_MEMREF_OUTPUT

   Direction: OUT

   Description: Unique ID in memory

   Constraints: Must be a pointer to address in shared memory buffer.

   Return Value

   Type: unsigned long

   Description: Error Code

   Possible Values: INVALID_ARGUMENTS, SUCCESS.

   Preconditions

   Enclave VM config must reside in contiguous guest physical address space.

   Post-conditions

   The memory occupied by the config file and the subsequent memory allocated for the VM will be unmapped from the host VM. The VM image and memory will be mapped onto the enclave VM.

Error Codes

INVALID_ARGUMENTS

SUCCESS

Security Considerations

Privileges Required: This API is exposed to the Guest OS.

Data Sensitivity: No sensitive data is accessed by the API.

Potential Threats: Abusing this API may allow an attacker to interfere with the execution of other guests by causing too many interrupts to occur, potentially causing a Denial of Service (DoS).

6.6 PUF

API Overview

API Name: PUF 2FA

API Version: 0.1a

Description: This API allows Apps to trigger a second factor authentication via PUF.

Note: This API is in development with UC1.

API Specification

1. **Function: Initialize the necessary variables and store them in nonvolatile memory**

   TA Function ID: crosscon_zkp_init

   Description: This function initializes the necessary variables for the ZKP proof.

   Module/Service Name: TA for prover authentication.

   Return Value

   Type: int

   Description: Status of the initialization process.

   Possible Values:

   ZKP_SUCCESS: Successful initialization.

   ZKP_ERR_FUSED: Initialization has already been done for this device and is fused.

   Non-zero error codes: Indicate potential issues such as elliptic curve computation failures, PUF access errors, or random number generation issues.

2. **Function: Enroll the device with Challenge Pairs**

   TA Function ID: crosscon_zkp_enroll

   Description: This function enrolls the Challenge Response Pairs and commits to them using the Pedersen commitment. The function deletes the challenge values after creating the commitment.

   Module/Service Name: TA for prover authentication.

   Input Parameters

   Input Parameter 1

   Name: challenge

Type: uint8_t challenge[32]

Direction: IN

Description: A 256-bit challenge given by the enrollment server.

Input Parameter 2

Name: challenge_response

Type: uint8_t *array[32]

Direction: OUT

Description: A committed 256-bit hash of the PUF response and the challenge.

Return Value

Type: int

Description: Status of the enrollment process.

Possible Values:

ZKP_SUCCESS: Enrollment successful.

ZKP_ERR_INIT_NOT_RUN: crosscon_zkp_init hasn't been ran before calling the enrollment.

ZKP_ERR_NO_MEMORY: No memory available to store the committed value.

ZKP_ERR_FUSED: The device has been fused.

Non-zero error codes: Indicate potential issues such as elliptic curve computation failures, PUF access errors, or random number generation issues.

3. **Function: Provide a challenge response**

TA Function ID: crosscon_zkp_prove_challenge

Description: This function generates a proof based on challenge request.

Module/Service Name: TA for prover authentication.

Constraints: ZKP must be initialized, and the device must be enrolled before calling the function.

Input Parameters

Input Parameter 1

Name: challenge

Type: uint8_t challenge[32]

Direction: IN

Description: A 256-bit challenge given by the authentication server.

Input Parameter 2

Name: nonce

Type: uint8_t array

Direction: IN

Description: The `nonce` guarantees the recency of the authentication process by introducing a unique, one-time value that prevents replay attacks. It is typically a sequence of randomly generated values or counters to guarantee that each authentication attempt is distinct.

Constraints: The size of the `nonce` array should match the expected size for the cryptographic operation (e.g., 64-bit or 512-bit).

Input Parameter 3

Name: challenge_response

Type: uint8_t *array[32]

Direction: OUT

Description: A committed 256-bit hash of the PUF response and the challenge.

Return Value

Type: int

Description: Status of the enrollment process.

Possible Values:

ZKP_SUCCESS: Response generation successful.

ZKP_ERR_INIT_NOT_RUN: crosscon_zkp_init hasn't been ran before calling the enrollment.

ZKP_ERR_NO_CHALLENGE: No challenge response for this challenge.

ZKP_ERR_NONCE_SIZE: Nonce too big.

Non-zero error codes: Indicate potential issues such as elliptic curve computation failures, PUF access errors, or random number generation issues.

Error Codes

TBD with UC1.

Security Considerations

Privileges Required: This API is exposed to the Guest OS.

Data Sensitivity: API doesn't expose sensitive data but accesses PUF and CRP data internally.

Potential Threats: Abusing this API may allow an attacker to interfere with the execution of other guests by causing too many interrupts to occur, potentially causing a Denial of Service (DoS).

API Dependencies

Related APIs: CROSSCON HV

External Resources: PUF of the platform. In our case, LPC's S-RAM PUF.

6.7 Remote attestation APIs

API Overview

API Name: Remote Attestation

API Version: 0.1a

Description: This API allows a 'remote' verifier to trigger attestation.

Note: This API is in development with UC4.

API Specification

1. **Function: Get Attestation**

TA Function ID: crosscon_get_attestation

Description: This function returns the attestation of the requested challenge.

Module/Service Name: TA for RA.

Input Parameters

Input Parameter 1

Parameter Name: challenge

Type: unsigned long / struct

Direction: OUT

Description: challenge is given by the verifier. Could be VM number or specific memory regions -- still TBD with UC4.

Return Value

Type: unsigned long

Description: Returns the measurement result.

Preconditions

VM to be attested needs to be running. Measurement service by the Crosscon Hypervisor needs to be available.

Error Codes

TBD with UC4.

Security Considerations

Privileges Required: This API is exposed to a remote verifier.

Data Sensitivity: Measurements for attestation might contain sensitive data.

Potential Threats: Abusing this API may allow an attacker to interfere with the execution of other guests by causing too many interrupts to occur, potentially causing a Denial of Service (DoS).

API Lifecycle

Initialization Requirements: Challenges need to be defined prior to init. Secure communication between TA and verifier needs to be established.

Termination/Cleanup: Secure communication channel between TA and verifier needs to be closed.

API Dependencies

Related APIs: Measurements from the Crosscon Hypervisor

External Resources: Remote Verifier to attest and verify the outcome of the measurement

6.8 Context-Based Authentication APIs

API Overview

API Name: Context-based Authentication

API Version: 0.1a

Description: This API allows VMs to use WiFi as a 2FA.

Note: This API is in development with UC1.

API Specification

1. **Function: Authenticate**

   Function ID: crosscon_wifi_auth

   Description: Returns the WiFi measurement for authentication.

   Input Parameters

   Input Parameter 1

   Name: challenge

   Type: struct

   Return Values

   Type: struct

   Description: Returns the Wifi measurement.

   Preconditions

   Available WiFi chip. Disconnect from AP.

   Relation between Pre-Post conditions (optional/nice to have):

   TBD with UC1.

   Post-Conditions

   Reconnection to AP.

2. **Function: Enroll**

   Function ID: crosscon_wifi_enroll

   Description: Enrolls the current WiFi environment.

   Input Parameters

   Input Parameter 1

   Name: TBD with UC1.

   Type: struct

   Description: Structure required for enrolling the device.

   Return Values

   Type: struct

   Description: Returns the enrollment result.

Error Codes

ERROR_INVALID_PARAMS, ERROR_HARDWARE_ISSUE

Security Considerations

Privileges Required: This API is exposed to a VM.

Data Sensitivity: Measurements might contain sensitive data.

Potential Threats: Abusing this API may allow an attacker to interfere with the WiFi connection of other VMs, potentially causing a Denial of Service (DoS).

API Lifecycle

Initialization Requirements: Possible challenges need to be defined. Needs complete access to Wifi chip.

Termination/Cleanup: Reconnect to WiFi.

API Dependencies

External Resources: WiFi chip compatible with Nexmon firmware patch.

6.9 Secure Update APIs

API Overview

API Name: Secure Update

API Version: 1.0

Description: This API allows Trusted Applications to perform a Secure Update procedure.

API Specification

1. **Function: Fetch Update Manifest**

   Function ID: TA_CROSSCON_FETCH_MANIFEST

   Description: Fetch update manifest from a specified remote server

   Module Name: Secure Update

   Input Parameters

   Input Parameter 1

   Name: firmware_server

   Type: const char *

   Direction: IN

   Description: A string containing the address of the firmware server

   Constraints: Must be a valid HTTPS address

   Input Parameter 2

   Parameter Name: manifest

   Type: char **

   Direction: OUT

Description: On success, the pointed location will contain a pointer to the buffer where the manifest has been stored

Constraints: n/a

Input Parameter 3

Name: manifest_length

Type: size_t *

Direction: OUT

Description: On success, the pointed location will contain the length of the fetched manifest

Constraints: n/a

Input Parameter 4

Parameter Name: embedded_image

Type: bool *

Direction: OUT

Description: On success, the pointed location will be set to true when the update image is embedded in the manifest, false otherwise

Constraints: n/a

Return Value

Type: int

Description: Return code

Possible Values: SUCCESS if the operation has been completed successfully; ERROR_NET in case of network error; ERROR_MEM in case of memory allocation error

2. **Function: Validate Update Manifest**

Function ID: TA_CROSSCON_VALIDATE_MANIFEST

Description: Validate an update manifest

Module Name: Secure Update

Input Parameters

Input Parameter 1

Name: manifest

Type: const char *

Direction: IN

Description: A pointer to the start of the stored manifest.

Constraints: Must point to a valid update manifest.

Input Parameter 2

Name: manifest_length

Type: size_t

Direction: IN

Description: The length of the manifest.

Constraints: Must be larger than zero.

Return Value

Type: int

Description: Return code

Possible Values: SUCCESS if the manifest has been validated correctly; ERROR_INVALID_SIGNATURE if the signature is incorrect; ERROR_INVALID_PROOF if a proof verification has failed

Preconditions

An update manifest has been obtained.

3. **Function: Get Update Version**

Function ID: TA_CROSSCON_GET_VERSION

Description: Obtain the version of an update manifest

Module Name: Secure Update

Input Parameters

Input Parameter 1

Name: manifest

Type: const char *

Direction: IN

Description: A pointer to the start of the stored manifest.

Constraints: Must point to a valid update manifest.

Input Parameter 2

Name: manifest_length

Type: size_t

Direction: IN

Description: The length of the manifest.

Constraints: Must be larger than zero.

Return Value

Type: char *

Description: A pointer to a string containing the version of the update stored in the manifest

Possible Values: A pointer to a NULL-terminated string

Preconditions

An update manifest has been obtained.

4. **Function: Get Update SBOM**

Function ID: TA_CROSSCON_GET_SBOM

Description: Obtain the Software Bill of Materials of an update manifest

Module Name: Secure Update

Input Parameters

Input Parameter 1

Name: manifest

Type: const char *

Direction: IN

Description: A pointer to the start of the stored manifest.

Constraints: Must point to a valid update manifest.

Input Parameter 2

Name: manifest_length

Type: size_t

Direction: IN

Description: The length of the manifest.

Constraints: Must be larger than zero.

Return Value

Type: char *

Description: A pointer to a string containing the Software Bill of Materials of the update stored in the manifest

Possible Values: A pointer to a NULL-terminated string

Preconditions

An update manifest has been obtained.

5. **Function: Get Update Properties**

Function ID: TA_CROSSCON_GET_PROPERTIES

Description: Obtain the list of Property IDs of an update manifest

Module Name: Secure Update

Input Parameters

Input Parameter 1

Name: manifest

Type: const char *

Direction: IN

Description: A pointer to the start of the stored manifest.

Constraints: Must point to a valid update manifest.

Input Parameter 2

Parameter Name: manifest_length

Type: size_t

Direction: IN

Description: The length of the manifest.

Constraints: Must be larger than zero.

Input Parameter 3

Parameter Name: properties

Type: char **

Direction: OUT

Description: A pointer to an array of strings, or NULL if the manifest contains no properties

Constraints: n/a

Return Value

Type: size_t

Description: The length of the properties array

Possible Values: From zero to the maximum number of properties supported by the manifest

Preconditions

An update manifest has been obtained.

6. **Function: Get Update Image**

Function ID: TA_CROSSCON_GET_IMAGE

Description: Obtain the update image

Module Name: Secure Update

Input Parameters

Input Parameter 1

Name: manifest

Type: const char *

Direction: IN

Description: A pointer to the start of the stored manifest.

Constraints: Must point to a valid update manifest.

Input Parameter 2

Name: manifest_length

Type: size_t

Direction: IN

Description: The length of the manifest.

Constraints: Must be larger than zero.

Input Parameter 3

Name: image

Type: char **

Direction: OUT

Description: On success, the pointed location will contain a pointer to the buffer where the update image has been stored

Constraints: n/a

Input Parameter 4

Name: image_length

Type: size_t *

Direction: OUT

Description: On success, the pointed location will contain the length of the update image

Constraints: n/a

Return Value

Type: int

Description: Return code

Possible Values: SUCCESS if the operation has been completed successfully; ERROR_NET in case of network error; ERROR_MEM in case of memory allocation error

Preconditions

An update manifest has been obtained.

7. **Function: Install Update Image**

Function ID: TA_CROSSCON_INSTALL_IMAGE

Description: Install an update image

Module Name: Secure Update

Input Parameters

Input Parameter 1

Name: image

Type: const char *

Direction: IN

Description: A pointer to the buffer containing the update image.

Constraints: Must be non-NULL and refer to a valid memory region.

Input Parameter 2

Name: image_length

Type: size_t

Direction: IN

Description: The length of the buffer containing the update image.

Constraints: Must be larger than zero.

Return Value

Type: unsigned long

Description: Error code

Possible Values: SUCCESS if the operation has been completed successfully; ERROR_INST in case of an installation error

Preconditions

An update image has been obtained.

Post-conditions

On success, the code for the affected component is updated.

Error Codes

SUCCESS, ERROR_NET, ERROR_MEM, ERROR_INVALID_SIGNATURE, ERROR_INVALID_PROOF, ERROR_INST

Security Considerations

Privileges Required: This API should be exposed to any Trusted Application.

Data Sensitivity: This module does not access sensitive data.

Potential Threats: Abusing this API may allow a malicious TA to launch resource-intensive processes on device (e.g. checking of big proof certificates), potentially causing a Denial of Service (DoS).

API Lifecycle

Initialization Requirements: none

Termination/Cleanup: Memory allocation for the manifest is performed by the Fetch Update Manifest function. It is the responsibility of the caller to free the `manifest` pointer at the end of the update process.

If the update image is embedded in the manifest, the Get Update Image function must not make any further allocations. Otherwise, the `image` pointer is allocated by the Get Update Image function and must also be properly freed at the end of the update process.

API Dependencies

Related APIs: none

External Resources: Firmware server, Verification server (only when proof verification is not performed on-device).

6.10 Secure FPGA Provisioning API

The FPGA resources in the CROSSCON stack are partitioned into two virtual sections. Different clients can use these APIs to select and use the available resources in any of these virtual FPGAs.

API Overview

API Name: FPGA Resource Provisioning

API Version: 0.4

Description: This API allows clients to use the CROSSCON stack to allocate a virtual FPGA.

API Specification

1. **Function: Configure**

   Function ID: crosscon_vFPGA_configure

   Description: Allows a client to configure a virtual FPGA on the FPGA.

   Input Parameters

   Input Parameter 1

   Name: clientID

   Type: int

   Direction: IN

   Description: A numeric ID for each client.

   Input Parameter 2

   Name: vFPID

   Type: int

   Direction: IN

   Description: The virtual FPGA ID to be configured.

   Input Parameter 3

   Name: enc_bit_stream_addr

   Type: char*

   Direction: IN

   Description: Starting memory address of the encrypted bitstream to configure.

   Input Parameter 4

   Name: enc_bit_stream_size

   Type: int

   Direction: IN

   Description: Size of the encrypted bitstream to configure in MB.

   Constraint: The bitstream can not be greater than the shared memory region of the underlying FPGA. Forenvironment ZCU102, it is 256MB.

   Return Values

   Type: int

   Description: Result of the FPGA provisioning.

   Possible Values: FAIL_NO_vFP: if there are no more resources available; FAIL_BITSTREAM: if the bitstream is not applicable or can not be decrypted for use; SUCCESS: the given bitstream is configured on the assigned virtual FPGA.

Preconditions

The FPGA resources need to be virtually divided between at least two virtual FPGAs.

7 Hardware/Software Co-Design

Most modern-day IoT platforms and hardware (HW) architectures provide some security-related HW primitives (MMU/MPU unit, random number generator, cryptographic accelerator, etc.) that can be used by software to enforce security-related guarantees. However, different HW primitives can have different impact on the system. Therefore, having the platform with the right primitives is important as it can be the only way to obtain strong security guarantees needed for a specific use cases and, at the same time, not sacrifice the performance of the system.

One aspect of the system's performance is utilization of it's resources. If a system's resource is limited, we want to utilize it as much as possible. This is often hard to do in a secure way as the resource needs to be shared between different security domains, but usually the hardware is not designed with this in mind. In the following chapters, we describe a HW primitive, called Perimeter guard, that addresses this problem and allows HW accelerators to be shared across different security domains in a secure way by preserving the isolation between the domains. We start by briefly introducing a CROSSCON SoC: a System-on-Chip (SoC) that utilizes Perimeter guard and BA51-H core to provide a platform that, when used with the CROSSCON Stack, can guarantee strong isolation between security domains running on the system.

7.1 High-level overview of CROSSCON SoC

CROSSCON SoC is a system on a chip that provides secure RISC-V execution environment for mixed criticality IoT devices that require strong security guarantees, flexibility, small code size and low power consumption. CROSSCON SoC is based around Beyond Semiconductor's BA51-H (RISC-V) core with HW extensions that enable isolation of software by running it inside of hardware-enforced software-defined virtualization-based TEEs. Furthermore, CROSSCON SoC allows sharing of HW modules connected to the SoC interconnect between TEEs without compromising isolation by using Perimeter guard: a hardware module that, when placed between SoC interconnect and another HW module, allows better access control to the HW module. Figure 17 shows the current fully featured high-level architecture of the CROSSCON SoC. Further details about the CROSSCON SoC are provided in [34]. Note that the architecture of the CROSSCON SoC is not fixed and could change in the future.

7.2 Perimeter guard

In this section, we describe a HW module called PG that can be used to share HW modules across different security domains of a system while preserving the isolation guarantees between the domains. We begin the chapter with an overview and motivation for PG and continue with its description.

7.2.1 Motivation

Imagine that we have an SoC that divides its resources into multiple isolated domains (i.e. contexts or VMs). We want to integrate a new HW module into the SoC in such a way that it can be used by multiple domains, but at the same time we want to preserve the isolation between the domains. For example, we want to be sure that a domain does not gain access to other domains using the newly added module. There does not appear to be a straightforward way to achieve this for arbitrary hardware modules, as modules are typically not designed with multi-domain isolation in mind.

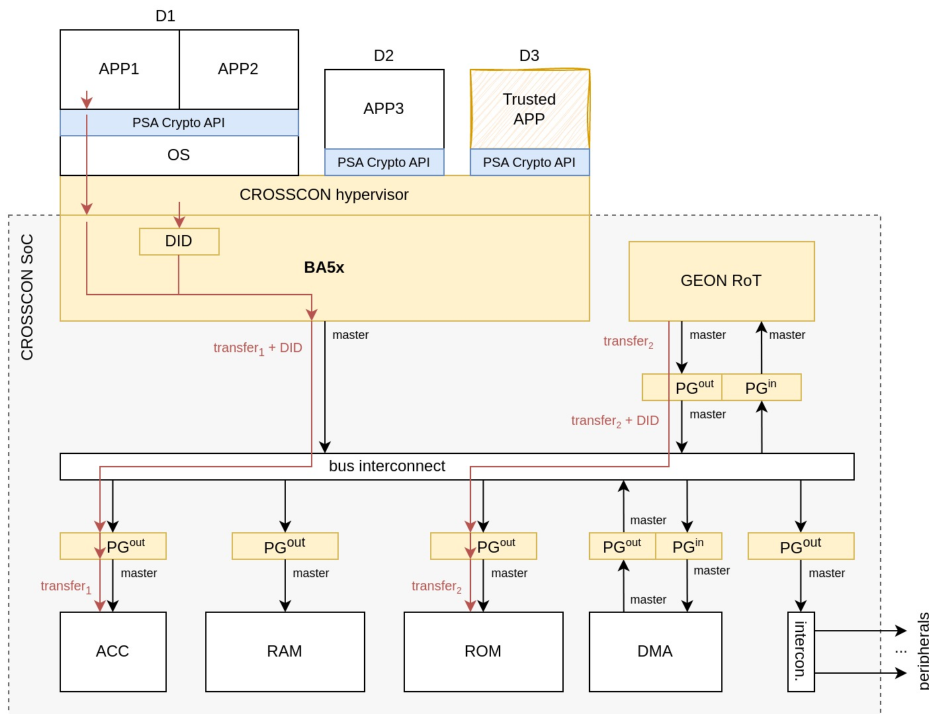


Figure 17: Current fully-featured architecture of the CROSSCON SoC

To determine if an arbitrary module is safe to use by multiple domains, we would need to look at the design of the module and make sure that no information flows from one domain to another. However, finding all possible information flows that pass through a module is often hard and time-consuming; because of that, it is usually not done in practice. This leads to a situation where we are forced to limit a HW module to a single domain (the safe way) or allow it to be shared and possibly introduce unwanted information flows (the unsafe way).

To address this situation, we want a solution that:

- allows us to share a HW module between domains without compromising the isolation,
- can be applied to existing HW modules, and
- does not require major HW module changes.

In the following chapter, we describe one such solution, called Perimeter guard.

7.2.2 Perimeter guard architecture

Perimeter guard (PG) is a HW module that can be placed in front of a slave module on a bus to control access to the module according to the provided policy. For example, Figure 18 shows a setup where there are two masters, M1 and M2, that are connected to the HW module (MOD) through PG over a bus where the PG's policy only allows M1 to access the module.

PG controls access to the module by restricting which (store and load) requests (i.e. transfers) can pass through to the module. We place PG in front of the module by connecting the PG's master interface directly with the module and the PG's slave interface with the bus interconnect. For PG to have full control over which master can access the module, the module should only be connected to the bus interconnect through PG. No other direct connections are allowed.

PG decides whether a request is passed through by considering the information provided in the request.

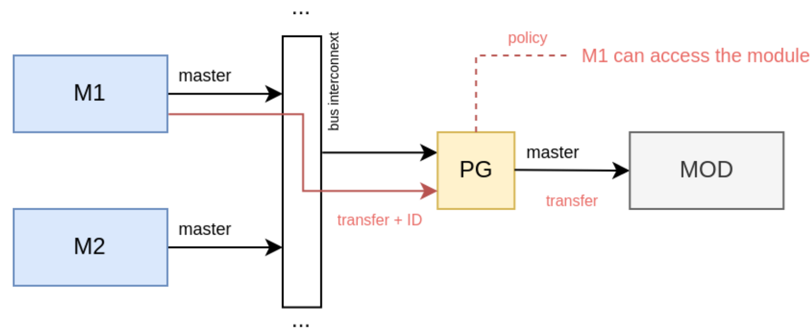


Figure 18: A basic SoC with two masters, M1 and M2, connected to a HW module (MOD) through PG on a bus.

The information can range from a simple master ID, that identifies the master that sent the request, to a set of tags that provide more specific information. Figure 18 illustrates the use of IDs. Which information is needed to make a decision depends on the setup and use case. Because of that, we do not limit the information that PG can use to decide whether to pass through a request; It is only required that master can provide that information. How the information is provided to PG usually depends on the bus protocol that connects the HW modules.

To facilitate further reading, it should be assumed that that all the SW and HW resources of an SoC (e.g. code, data, threads of execution, computing cores, RAM, cryptographic accelerators, IO devices, etc.) are grouped into domains. A domain is a collection of SW and HW resources. For example, in case of the CROSSCON Hypervisor (Seciton 4.1.4), we say that each VM and related resources it uses are grouped in its own domain. Two domains are isolated if no information is shared between the resources of the domains in other way than through allowed communication channels. Note that what belongs to a specific domain needs to be explicitly defined for each setup that is considered. Having the idea of domains in mind, we say that PG is used to restrict access to a HW module to a specific domain, while keeping the solution generic. It is always possible to choose domains in a way that allows appropriate granularity for a specific setup.

Although we think PG can be applied to a broad range of HW modules, currently our main focus is on how it can be applied to hardware accelerators, such as cryptographic accelerators. Figure 19 shows a simplified setup of CROSSCON SoC [4] where PG is used to restrict access to HW accelerator (ACC). In this setup, it is assumed that each VM is part of a domain: VM1 is part of a domain D1 and VM2 is part of a domain D2. Each time a "store / load" instruction is performed in one of the VMs, the domain ID (DID) of the domain that the VM is a part of is passed along the transfer performed on behalf of the instruction. PG then uses the DID and other information provided in the transfer to decide if the transfer is passed along to the HW module.

For the initial implementation of PG, it is assumed that masters and slaves communicate using the AMBA 3 AHB-Lite protocol (version A) [35]; where, in the future, PG can be adapted for other bus protocols. Furthermore, it is assumed that each transfer over an AHB bus is performed on behalf of a domain. A master transmits the DID using a dedicated signal called HDID. We assume that all masters drive the HDID in an honest way, i.e. the HDID reflects the domain that caused the transfer.

Thus far, we have identified several ways that PG can restrict access to a HW module:

- **Exclusive access mode** – Only one domain can access the module during a runtime of a system.
- **Restricted address space mode** – Each domain has access only to a specific part of the address space of the module. PG in this mode can be viewed as a module specific MPU.

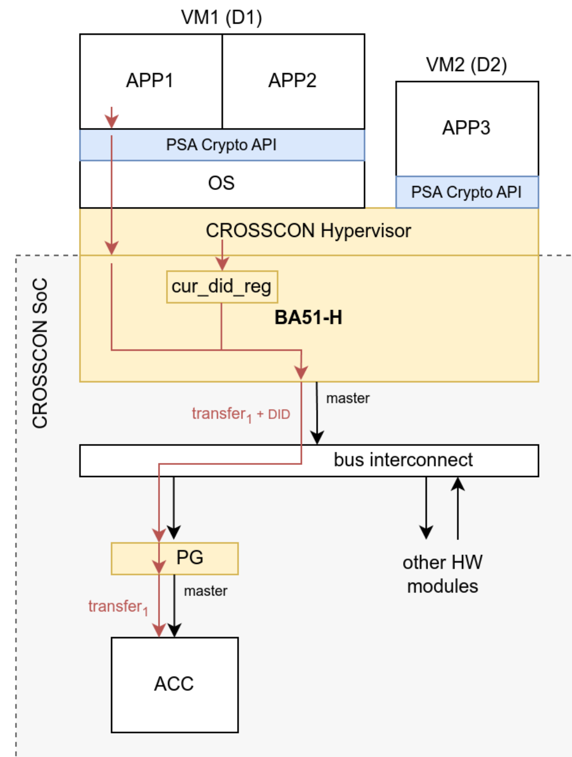


Figure 19: How PG is used in the of CROSSCON SoC to restrict access to a HW accelerator.

- **Time-sharing with reset mode** – A HW module is time-shared between domains. Every time another domain gets access to the module, the HW module is reset to a predefined state.
- **Time-sharing with context switching mode** – A HW module is time-shared between domains. Every time the module is passed on to a different domain, the state of the module is preserved and new domain's state is loaded into the module.

We call the way PG behaves its *operation mode*. In general, we can distinguish between different operation modes based on how a mode decides which master can access the HW module:

- **External arbitration mode** – PG allows a master that has access to PG configuration interface to decide which master can access the HW module; PG just enforces the decision. This is useful when we want to delegate a decision of which master can access the module to an external master.
- **Lock-release arbitration mode** – PG provides a lock-release mechanism that allows a master to lock and release a module which allows a master to access the module for a specific time. This differ from the Extrnal arbitration mode as an external master can only request an access to the module for itself.
- **Round-robin arbuitration mode** - PG uses a Round-robin scheduling mechanism to decide which master can access the module.

We can further describe PG's operation mode by specifying how it handles arbitration, for example, PG can be in time-sharing with reset and external arbitration mode.

PG can be configured using a set of configuration registers that are available through a separate *configuration interface* (Section 7.2.5). Only masters and domains that have access to that interface can configure PG. Each operation mode has a different set of configuration registers which are described alongside each operation mode.

In the following chapter, we provide a threat model specific to PG in the context of the CROSSCON SoC and describe the operation modes.

7.2.3 The threat model

The following is a specific threat model for PG in the context of CROSSCON SoC (Section 7.1).

Setup: We assume a similar architecture of the CROSSCON SoC as shown in Figure 19. The BA51-H core and the CROSSCON Hypervisor are used to establish several isolated VMs that are allowed to interact with each other according to the configuration of the CROSSCON Hypervisor. We consider each VM to be in its own separate domain. Furthermore, each of the masters connected to the AHB MLIC is a part of some domain and drives the HDID signal for each transfer it initiates. We assume that only a privileged domain has access to the configuration interface of all PGs and can configure them.

Goal: Our goal is to allow a new HW module to be added to the system so that the module can be shared between domains while preserving existing isolation guarantees.

We assume the following threat model: The attacker has full control of one or several domains (VMs and masters), excluding the privileged domain. The attacker can execute arbitrary instructions, start new programs, influence existing programs, modify OS, control peripheral devices and HW modules, etc., inside of the domains they control.

The BA51-H, interconnect, the CROSSCON Hypervisor, the privileged domain configuring PG, and PG itself are **trusted** by default. It is assumed that all the components that are trusted, are functionally correct, have no bugs, and behave as expected.

In the following chapters, we argue that PG used together with the newly added module can be used to preserve existing isolation guarantees and, at the same time, allow the module to be shared between domains while assuming previously described threat and trust models.

We assume that physical attacks, such as fault injection and bus sniffing, are out of scope. We consider physical attacks as an orthogonal problem that can be mitigated by using separate solutions.

7.2.4 General consideration of PG attack surface

In order to preserve the isolation between domains, we need to ensure that no information flows from one domain to another unless it is explicitly allowed. We have identified the following ways of how information can flow from one domain to another: (1) through the (internal) state of the module and (2) through module's availability (i.e. when and how long the module is used).

State attacks: A malicious program could try to use module's (internal) state to obtain information about the previous domain that used the module, or influence the domain that will use the module after it. This is a general class of attacks that can be avoided by being able to modify / control the module's state before passing it to another domain, for example, by resetting it to a predefined state.

Note that the observation that state-related attacks can be avoided by resetting HW modules state is significant as it provides a general mechanism for how HW modules can be shared without leaking any information through the module's state.

Timing attacks: Another class of attacks are timing attacks, where the malicious program can use the information of when and for how long other domains use the module to learn something new about them (i.e. timing side channel attacks). For example, if a program knows how long a module was used, it could learn which operation was performed by the module. We can address this kind of attacks by having a "fair" arbitration or by hiding the timing information in some way.

7.2.5 Configuring PG

Each PG exposes its configuration registers through a separate APB [36] interface, called *configuration interface*. For each setup, it is assumed that there exists a privileged domain that has access to the configuration registers. How the access to the registers is isolated and which domains have access to them is defined at the time PG is integrated into an SoC.

In case of the CROSSCON SoC, the PG's configuration interfaces are connected only to BA51-H and are memory-mapped to the execution environment, which means that only software running on BA51-H can access the registers. The CROSSCON Hypervisor can then further be used to limit which VMs can access the registers by using memory protection unit; for example, the CROSSCON Hypervisor could allow only VM 0 with HDID 0 to access the registers, which makes domain 0 a privileged domain.

7.2.6 Responding to access violation

PG rejects a transfer if the domain that triggered the transfer does not have the right permissions. If transfer is rejected, PG does not forward the transfer to the protected module but notifies the configured domains by raising an interrupt or by signaling error over a bus (e.g. in case of AHB, by driving HRESP signal to high). Because transfers need to be completed, PG completes the transfers by responding to the master with pre-defined values.

The behavior of how and which domains are notified, is configured at the integration of PG into an SoC. For example, in the CROSSCON SoC all the access violations are signaled by raising an interrupt which is connected to BA51-H.

When a VM (or a domain) receives an interrupt, the VM can identify which domain has caused the interrupt and reset it by doing the following:

- First, VM must identify the PG that raised the interrupt, by reading the `pg_interrupt_source_reg` of all PGs. If `pg_interrupt_source_reg` is set to 1, PG has raised the interrupt.
- Next, the VM can check which domain caused the interrupt by reading the `pg_interrupt_cause_reg` register.
- Finally, the VM can reset the interrupt by writing 1 to the `pg_interrupt_source_reg` register.

Note that the interrupt can only be reset by a domain that has access to the PG's configuration registers.

7.2.7 Operation Mode: Exclusive access with external arbitration mode

The *Exclusive access* mode is one of the most straightforward behaviors PG can provide. The mode allows to restrict access to a module to a single domain. Only a domain with the same DID as the value of the mode's `pg_acc_did_reg` register can access the module. For example, as considered in Figure 18, the privileged domain can set the `pg_acc_did_reg` to DID of M1 only to allow domain M1 to access MOD. That is, assuming M1 and M2 belong to different domains.

The `pg_acc_did_reg` register can be set at any point during the run-time of the SoC, which means that the Exclusive access mode can also be used with external arbitration; for example, when the privileged domain performs the arbitration.

Configuration registers: Table 2 shows the PG configuration registers used in Exclusive access with external arbitration mode, where `HDID_WIDTH` is the size of HDID. HDID max size is 4 bytes. The privileged domain can configure which domain can access the protected HW module by writing the domain's ID to the `pg_acc_did_reg` register. The register is exposed through a configuration interface.

Table 2: Configuration registers used in PG's Exclusive access with external arbitration mode.

Register	Size (in bits)	Address[8:2] (in hex)	Description
pg_acc_did_reg	DID_WIDTH	0x0	The DID of the domain that can access the HW module. Reset value of pg_acc_did_reg is 0x0.

Threat analysis: In case where only one domain can access the module, attacks related to module's internal state and availability are not an issue because the module is not shared between domains. Note that a module could be reassigned to a different domain also during the runtime if a privileged domain reconfigures PG. This should only be done when no information is leaked through the internal state and availability of the module.

7.2.8 Operation mode: Restricted address space with external arbitration mode

In *Restricted address space mode*, PG can restrict access to an address range to a specific set of domains. Which domains can access the address range is determined by a set of *address range entries* composed of:

- start address - start of the address range,
- end address - end of the address range,
- did - ID of the domain that can access the address range, and
- access permissions - permissions of the domain that can access the address range.

Note that because the address ranges can overlap, several domains might be able to access the same range. Therefore, one needs to be careful when configuring the address ranges to get the desired isolation of VMs.

In case there is no address range entry that allows a domain to access the desired address range, PG prevents the access and considers this an access violation. See Chapter 7.2.6 for how PG treats access violations.

PG can have an arbitrary number of entries in the address range. The number of entries is set at the integration. We assume that PG has M address range entries.

Configuration registers: Table 3 lists configuration registers available in the restricted address space mode, it is assumed that DID_WIDTH + 3 is smaller or equal to 32. The registers are exposed through a configuration interface.

The i-th address range configuration is provided to PG through the following registers:

- pg_addr_i_reg contains the start address of the i-th address range,
- pg_addr_i+1_reg contains the end address of the i-th address range, and
- pg_acc_prm_i_reg contains the did and the access mode of the i-th address.

Table 20 shows the format of pg_addr_i_reg and pg_acc_prm_i_reg registers where L is the size of the pg_acc_prm_i_reg that is equal to DID_WIDTH + 3. Register pg_addr_i_reg is a HDIDW_WIDTH-bit WARL (Write-Any-Read-Legal) register that contains the start and end address of an address range. Register pg_acc_prm_i_reg is an (DID_WIDTH + 3)-bit WARL register that contains A, DID, W and R fields. Field A indicates if the address range entry is active, field DID is the domain to which the entry applies to, field W indicates that the domain can write to the address and field R indicates that domain can read from the address.

Table 3: Configuration registers available in the Restricted address space with external arbitration mode.

Register	Size (in bits)	Address[8:2](in dec)	Description
pg_addr_0_reg	HADDR_WIDTH	0	The start address of 1st address range.
pg_addr_1_reg ...	HADDR_WIDTH ...	1 ...	The start address of 2nd address range and the end address of 1st address range.
pg_addr_M_reg	HADDR_WIDTH	M	The end address of M-th address range.
pg_acc_prm_0_reg	DID_WIDTH + 3	M + 1	The DID of the domain that can access the 1st address range according to the specified permissions.
pg_acc_prm_1_reg ...	DID_WIDTH + 3 ...	M + 2 ...	The DID of the domain that can access the 2nd address range according to the specified permissions.
pg_acc_prm_M-1_reg	DID_WIDTH + 3	M + M	The DID of the domain that can access the M-th address range according to the specified permissions.

Workflow: In Restricted address space mode, PG supports the following workflow:

1. Upon reset, no domain has access to the protected HW module. All configuration registers are set to 0.
2. The privilege domain can grant access to a specific address range to a domain by:
   - writing the address range into the i-th pair of address registers (pg_addr_i_reg and pg_addr_i+1_reg) and
   - writing the domain's ID and permissions to the pg_acc_prm_i_reg.
3. When a read or write transfer is received by PG:
   - PG checks if the transfer's address is in one of the configured address ranges.
   - If a transfer's address is in one of the address ranges, PG checks if a domain has access permissions for the specified address range; otherwise, PG rejects the transfer (Chapter 7.2.6).
   - If the domain has access permissions for a specified address range, the transfer is forwarded to the HW module; otherwise, PG rejects the transfer (Chapter 7.2.6).

Threat analysis: The Restricted address space mode allows multiple domains to access the protected HW module simultaneously. Given that, we should consider the module's behavior to analyze which guarantees are provided. For example, a relevant use case is given a cryptographic accelerator we want to restrict the handling of the cryptographic keys to only a single domain, but at the same time, using the cryptographic accelerator. By using Restricted address space mode, PG can be configured to allow only the domain that handles the cryptographic keys to supply the keys to the accelerator. If the cryptographic accelerator prevents the keys from being extracted from the internal state of the module or through timing, the domain handling the cryptographic keys remains isolated from other domains.

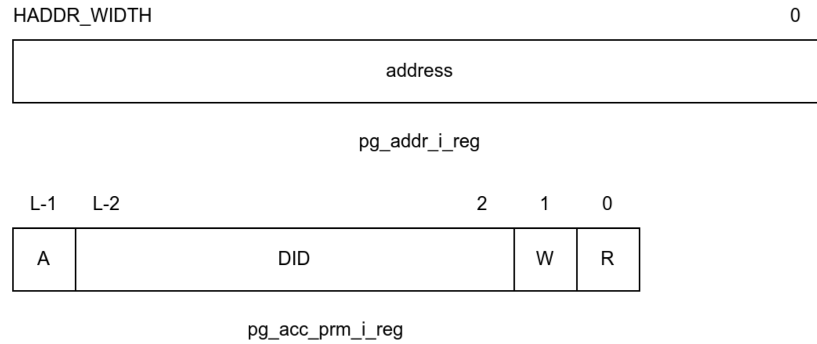


Figure 20: Format of pg_addr_i_reg and pg_acc_prm_i_reg where L equals DID_WIDTH + 3.

7.2.9 Operation mode: Time-sharing with reset and lock-release arbitration mode

In *Time-sharing with reset and lock-release arbitration mode*, PG restricts access to the protected HW module to one domain at a time and allows the module to be accessed from other domains only after the module is reset. In order for PG to be used in this mode of operation, the module needs to support a reset mechanism. The reset mechanism sets the internal state of the module to a predefined state that does not contain any information from the domain that previously used the module. The access to the module is arbitrated through a lock-release mechanism where a domain needs to lock the module before using it and release it afterwards. A module can be locked and released by reading from `pg_lock_reg` and `pg_release_reg` register available to each domain through a separate AHB [35] interface. Figure 21 describes the behavior of the mode.

Workflow: In Time-sharing with reset and lock-release arbitration mode, PG supports the following workflow:

1. Upon reset, the module is not used by any domain.
2. For a domain to use the module, it needs to lock it. A module can be locked by reading the `pg_lock_reg` register. If the returned value is 1, the module was successfully locked; otherwise, 0 is returned, and the domain needs to retry to lock the module before using it.
3. When a domain has successfully locked the module, it can perform an arbitrary sequence of read and write requests that will be forwarded to the module.
4. If a domain does not hold a lock on the module, all requests to the module will be rejected (Chapter 7.2.6).
5. After the domain is done using the module, it can release the module by reading the `pg_release_reg` register. If the returned value is 1, the module was successfully released.
6. When the module is released, the module is reset and then it can be locked by other domains.

Note that if module is being reset and a domain tries to lock the module, a read from `pg_lock_reg` will return 0 for the duration of the reset, which indicates that the module has not been locked.

Configuration registers: The Lock-release arbitration mode does not have any specific configuration registers.

Status registers: Table 4 lists the status registers available in the Time-sharing with reset and Lock-release arbitration mode. The status registers are available through a separate AHB interface [35] and should be mapped to each domain that wants to use the protected module.

Reset mechanism: In this mode, PG provides two additional signals, `is_rst_o` and `is_rst_ack_i`,

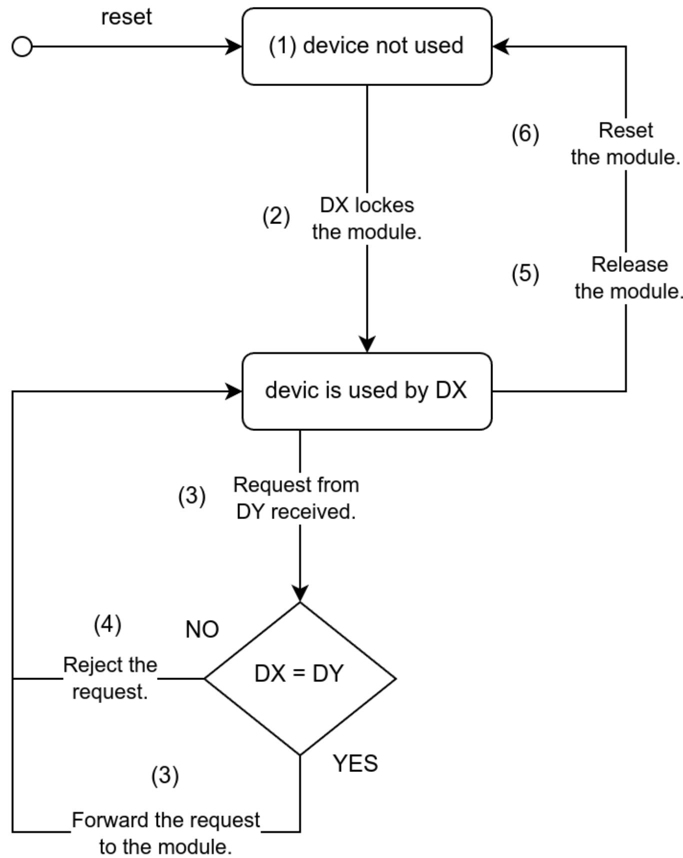


Figure 21: The behavior of Time-sharing with reset and lock-release arbitration mode, where DX and DY represent domains.

Table 4: Status registers available in the Time-sharing with reset and lock-release arbitration mode.

Register	Size (in bits)	Address[8:2](in dec)	Description
`pg_lock_reg`	32	0	Read to lock the module. Returns 1, if the module was successfully locked; otherwise returns 0.
`pg_release_reg`	32	1	Read to release the lock on the module. Returns 1, if the module was released; otherwise returns 0.

that allow PG to trigger a reset and wait for the confirmation that the module has been reset. PG sets `is_rst_o` to HIGH to indicate that the module should be reset. After the module has performed the reset, it needs to drive `is_rst_ack_i` signal HIGH to let PG know that the reset was completed. Note that the reset can take several cycles to complete.

Threat analysis: The Time-sharing with reset and lock-release arbitration mode guarantees that there are no explicit information flows between domains using the module, but it does not protect against timing attacks as domains can monitor when the module is used by other domains. Furthermore, the mode allows DoS attacks as a domain can deny access to the module by never releasing the lock.

Note that other arbitration modes could be used with the Time-sharing with reset mode; for example, external arbitration or reset on the first write request. By using different arbitration modes, it might be possible to prevent timing attacks or DoS attacks, but extra caution is needed when triggering a reset. If the reset is performed, before a domain has finished using the module, the information on the module

would be lost, which could hinder the operation of the domain using the module. This issue is addressed in Time-sharing with context switching mode.

7.2.10 Operation mode: Time-sharing with context switching mode

The *Time-sharing with context switching mode* is similar to Time-sharing with reset mode (Section 7.2.9) as it only allows one domain to access the HW module at the time, but it differs in how it modifies the module state when switching from one domain to another. In Time-sharing with context switching mode, PG preserves the internal state of the module instead of resetting it. Each time a new domain obtains access to the module, PG replaces the module's internal state with the state corresponding to that domain. We call this *context switching* as the same principle is used when switching threads of execution on a CPU core. Figure 22 shows a diagram of PG in context switching mode.

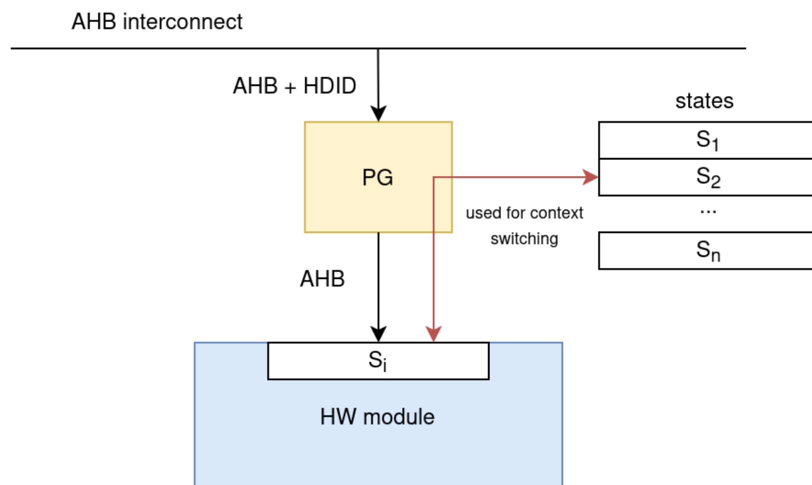


Figure 22: PG in context switching mode

The main advantage of context switching is that it preserves the state of the module while other domains are using it, and, at the same time, prevents any state-related information flows, as each domain can only access the state of the module that belongs to that domain. The downside of context switching is that the HW module needs to support it, which requires additional resources and complicates the design of the module. Furthermore, the process of switching can take a lot of cycles to perform, witch reduces the availability of the module.

Module's interface: The registers exposed by a HW module are memory-mapped to each domain and can be accessed on the same address. From the domain's perspective, the interface is the same as the interface of the module without the context switching support, except that the results of the operation performed by the module are available with a delay as the module is shared.

Arbitration mode: Context switching provides a way to share a HW module without losing the module's state. This allows us to switch the module from one domain to another at arbitrary points during the module's operation; therefore, allowing for a wider range of arbitration modes. For example, the Lock-release arbitration mode could be used or some form of "fair" scheduling; for example, Round-robin scheduling where the module is assigned to each domain in order for a fixed number of clock cycles.

Configuration registers: Table 5 lists the configuration registers available in the Time-sharing with context switching mode and Round-robin arbitration, where `DID_WIDTH` is smaller or equal to 32.

Context switching mechanism: In order for a HW module to be used with context switching, it needs to

Table 5: Configuration registers available in the Time-sharing with context switching and Round-robin arbitration mode.

Register	Size (in bits)	Address[8:2](in dec)	Description
pg_num_of_dom_reg	32	0	The number of domains using the module. The number should be smaller or equal to 16.
pg_did_0	DID_WIDTH	1	The DID of 1st domain that can use the module.
pg_did_1 ...	DID_WIDTH ...	2 ...	The DID of 2nd domain that can use the module. ...
pg_did_15	DID_WIDTH	16	The DID of 16th domain that can use the module.
pg_switch_period_reg	32	17	The number of clock cycles the module performs computation on behalf of a domain each time the time on the module is given to that domain.

expose its internal state as a set of registers accessible from the outside of the module. This allows the context switching mechanism to read and modify the module's state, which allows storing and restoring the module's context. Furthermore, the module needs to support *stalling*; when a stall signal is set to HIGH, HW module should not perform any computation and modification of registers in the next clock cycle. This is needed as the context switching mechanism needs to stall the module for one or more cycles so that it has time to modify the registers. Figure 23 shows how the stalling is used.

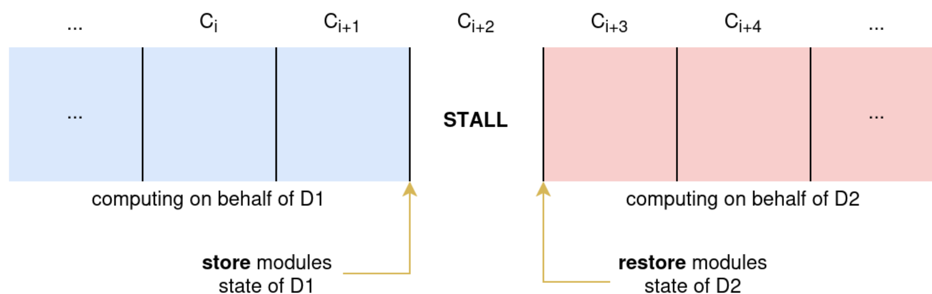


Figure 23: How stalling is used to provide time for the context switching mechanism to store the module's state of domain 1 (D1) and restore the module's state of domain 2 (D2).

Threat analysis: By using context switching with Round-robin arbitration, both state and availability related attacks can be prevented. By having a different module state for each domain, any state-related information flows are prevented. By giving each domain the same amount of time to use the module in a fixed order with Round-robin scheduling, any timing information flows are avoided, as other domains are not able to gather any information about the amount of time other domains are using the module. The primary drawback of using Round-robin with fixed intervals for each domain is the inefficient utilization of the module, as it is allocated to a domain even when the domain does not actively use it.

7.2.11 Integrating PG with the CROSSCON Stack

All masters connected to the interconnect must drive the HDID signal according to the domain they belong to. Because majority of masters are not aware of HDID, we add an additional module in front of them that is responsible to drive the HDID signal. We call the module PG^{in}. Figure 17 shows a CROSSCON SoC setup where each master on a bus, except the BA51-H core, has a PG^{in} module that drives HDID for each transfer. To avoid confusion, all PGs are renamed to PG^{out}. By using PG^{in} and PG^{out}, we can construct a bus interconnect in which all the transfers over the interconnect are marked with DID of the domain that caused the transfer.

Figure 24 shows an example of how a cryptographic accelerator (AES / RSA) can be integrated using PG throughout the entire HW-SW stack.

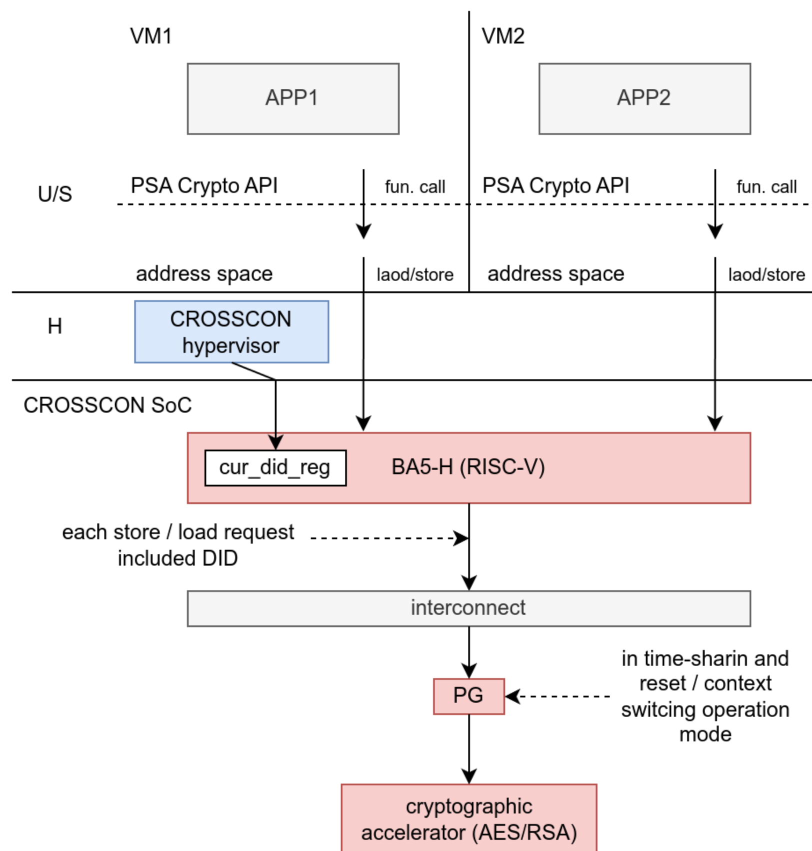


Figure 24: An example of how a cryptographic accelerator can be integrated within the CROSSCON SoC and Hypervisor through PG

The accelerator interface is memory-mapped to the address space of each VM so that applications running inside the VM can interact with the accelerator. Ideally, the applications can interact with the module, as specified by the module's interface, and no additional steps are needed. This depends on the PG operation mode; for example, in the case of Lock-release arbitration mode, the application should additionally lock and release the module. To avoid adding unnecessary complexity to the application, usually the module's interface is exposed through a library that has a standard interface; for example, for the cryptographic accelerator, the PSA Crypto API [37] could be used. This makes the application easier to port as it can be used by any library that exposes the same interface.

Note that the CROSSCON Hypervisor does not arbitrate which VM has access to the module as it is done by PG or privileged VM. The only thing that the CROSSCON Hypervisor needs to do is to let the BA51-H know which VM is currently running by writing the VM's ID to `cur_did_reg` register on each

VM context switch. The ID stored in `cur_did_reg` is then forwarded by BA51-H through HDID signal on each store and load request so that PG knows which domain / VM is trying to access the module.

7.2.12 Comparison with other solutions

The idea of dividing resources into different domains on a SoC level is not new and is already supported by some solutions. For example, SiFive's WorldGuard [38] and Arm's TrustZone [39] divide SoC resources into two or more banks that are isolated from each other. The difference between WorldGuard, TrustZone and solution proposed here is that the solution focuses on mechanisms of how to share a module across domains while preserving existing separation guarantees, where:

- WorldGuard puts more emphasis on how to divide HW modules into different banks on a level of modules connected over a bus and not on mechanisms of how a module can be shared between domains with certain guarantees and
- Arm with TrustZone provides a broader solution that extends Arm's ISA, processor, and bust protocol (AHB-lite and AXI) and also provides several IP cores that can be used to implement an SoC with TrustZone support.

As far as we know, WorldGuard and TrustZone do not focus on a general problem of how a HW module can be shared between domains while preserving the existing isolation guarantees, although both solutions address this problem in some (less general) way specific for their setup.

8 Conclusion

In this document, we have presented the final version of the open specification of CROSSCON. Starting from the overall high-level architecture and related adversary model, we have presented the individual architectural components of CROSSCON and presented possible implementation alternatives of the CROSSCON stack in order to accommodate a wide variety of different HW platforms on which CROSSCON can be instantiated. The architecture specification provided in this deliverable is used as a basis for the technical work in work packages WP 3 and WP 4, in which the security stack components and support for domain-specific hardware architectures are specified and developed.

References

[1] Ainara García, David Purón, Bruno Crispo, Hristo Koshutanski, Krystian Hebel, Maciej Pijanowski, Michał Żygowski, and Rafał Kochanowski. *Deliverable D1.1: Use Cases Definition Initial Version*. 2023.

[2] David Purón, Ainara García, Rafał Kochanowski, Ziga Putrle, Yacine Felk, Emna Amri, Akos Milankovich, Gergely Eberhardt, Sandro Pinto, Bruno Crispo, Marco Roveri, Michele Grisafi, and Peter Ten. *Deliverable D1.2: Deliverable D1.2: Requirements Elicitation Initial Technical Specification*. 2023.

[3] Sandro Pinto, David Cerdeira, João Sousa, Luís Cunha ND Bruno Crispo, Michele Grisafi, Marco Roveri, Alberto Tacchella, Tommaso Zoppi, Lukas Petzi, Peter Ten, Hristo Koshutanski, and Shaza Zeitouni. *Deliverable D3.1: CROSSCON Open Security Stack Documentation - Draft*. 2024.

[4] Žiga Putrle, Shaza Zeitouni, Richard Mitev, Marco Chilese, and Lukas Petzi. *Deliverable D4.1: CROSSCON Extensions to Domain Specific Hardware Architectures Documentation - Draft*. 2024.

[5] GlobalPlatform. *TEE Internal Core API 1.3.1*. [Online]. Available at: https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[6] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. ``ReZone: Disarming TrustZone with TEE Privilege Reduction''. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022. ISBN: 978-1-939133-31-1.

[7] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. ``Keystone: An open framework for architecting trusted execution environments''. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020.

[8] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. ``Komodo: Using verification to disentangle secure-enclave hardware from software''. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017.

[9] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. ``SANCTUARY: ARMing TrustZone with User-space Enclaves.'' In: *NDSS*. 2019.

[10] Victor Costan. ``Intel SGX explained''. In: *IACR Cryptol, EPrint Arch* (2016).

[11] Renesas Eletronics. *HEX FIVE MULTIZONE® SECURITY FAQ for Arm® Platforms*. [online] Available at: https://www.renesas.com/ja/document/oth/hex-five-multizone-security-faq-arm-platforms?srsltid=AfmBOor5UIl4MtvmCAuIgQvUClwOeMx4xshafSWjGaDnrMGxpMfFpQON. 2020.

[12] WorldGuard Task Group. *WorldGuard Specification*. 2023.

[13] David Kaplan. ``Protecting VM register state with SEV-ES''. In: *White paper* (2017), p. 13.

[14] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. ``Intel tdx demystified: A top-down approach''. In: *ACM Computing Surveys* 56.9 (2024), pp. 1–33.

[15] Arm. *Learn the architecture - TrustZone for AArch64*. [online] Available at: https://developer.arm.com/documentation/102418/0102/TrustZone-in-the-processor/Secure-virtualization. 2020.

[16] Arm. *Introducing Arm Confidential Compute Architecture*. [online] Available at: https://developer.arm.com/documentation/den0125/0300/Overview. 2023.

[17] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. ``CoVE: Towards Confidential Computing on RISC-V Platforms''. In: Association for Computing Machinery, 2023.

[18] Arm. *Trusted Firmware-A (TF-A)*. URL: https://www.trustedfirmware.org/projects/tf-a/.

[19] RISC-V International. *RISC-V Open Source Supervisor Binary Interface (OpenSBI)*. URL: https://github.com/riscv-software-src/opensbi.

[20] Sandro Pinto and Nuno Santos. ``Demystifying arm trustzone: A comprehensive survey''. In: *ACM computing surveys (CSUR)* 51.6 (2019), pp. 1–36.

[21] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. ``The ANDIX research OS — ARM TrustZone meets industrial control systems security''. In: *International Conference on Industrial Informatics (INDIN)*. 2015.

[22] Sandeep Gupta. ``An edge-computing based Industrial Gateway for Industry 4.0 using ARM TrustZone technology''. In: *Journal of Industrial Information Integration* (2023).

[23] Qinyu Zhu, Quan Chen, Yichen Liu, Zahid Akhtar, and Kamran Siddique. ``SMC Calling Convention (SMCCC)''. In: *Security and Communication Networks* 2023.1 (2023), p. 7369634.

[24] Samuel Pereira, João Sousa, Sandro Pinto, José Martins, and David Cerdeira. ``Bao-enclave: Virtualization-based enclaves for arm''. In: *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*. IEEE. 2022, pp. 1–6.

[25] Arm. *Introducing Arm Confidential Compute Architecture*. [online]. URL: %5Curl%7Bhttps://developer.arm.com/documentation/den0028/latest%7D.

[26] RISC-V International. *RISC-V SBI specification*. [online] Available at: https://github.com/riscv-non-isa/riscv-sbi-doc.

[27] Arm. *Arm Power State Coordination Interface Platform Design Document*. [online] Available at: https://developer.arm.com/documentation/den0022/e.

[28] Arm. *Arm Generic Interrupt Controller Architecture Specification GIC A version 3 and version 4*. [online] Available at: https://developer.arm.com/documentation/ihi0069/h/.

[29] Arm. *GIC*. URL: %5Curl%7Bhttps://developer.arm.com/Architectures/Generic%20Interrupt%20Controller%7D.

[30] RISC-V International. *RISC-V Platform-Level Interrupt Controller Specification*. [online] Available at: https://github.com/riscv/riscv-plic-spec.

[31] RISC-V International. *RISC-V Advanced Interrupt Architecture (AIA)*. [online] Available at: https://github.com/riscv/riscv-aia.

[32] Arm. *SMC Calling Convention (SMCCC)*. URL: https://developer.arm.com/documentation/den0028/latest.

[33] GlobalPlatform. *TEE Client API Specification 1.0*. [Online]. Available at: https://globalplatform.org/specs-library/tee-client-api-specification/.

[34] CROSSCON Team. *Deliverable D2.4: CROSSCON formal specification - Final Version*. 2025.

[35] ARM. *AMBA 3 AHB-Lite Protocol Specification (Version A)*. 2006. URL: https://developer.arm.com/documentation/ihi0033/a/?lang=en (visited on 2024).

[36] ARM. *AMBA 3 APB Protocol Specification (Version B)*. 2004. URL: https://developer.arm.com/documentation/ihi0024/b/?lang=en (visited on 2024).

[37] *PSA Certified Crypto API*. 1.2. Issue Number: 1. ARM. Mar. 2024.

[38] SiFive. *SiFive WorldGuard Technical Paper*. Tech. rep. SiFive, 2021.

[39] ARM. *ARM Security Technology, Building a Secure System using TrustZone Technology*. Tech. rep. ARM, 2009.