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

D2.1 CROSSCON Open Specification - Draft

Document Identification			
Status	Final	**Due Date**	31/10/2023
Version	1.0	**Submission Date**	31/10/2023

Related WP	WP2	**Document Reference**	D2.1
Related Deliverable(s)	D2.3	**Dissemination Level (*)**	PU
Lead Participant	TUD	**Lead Author**	Markus Miettinen Shaza Zeitouni
Contributors	UNITN, UMINHO, UWU, BEYOND	**Reviewers**	Lukas Petzi, UWU Gergely Eberhardt, SLAB

Keywords:
CROSSCON Trusted Execution Environment, CROSSCON Hypervisor, CROSSCON System on Chip

Document Information

List of Contributors	
Name	**Partner**
Markus Miettinen	TUD
Shaza Zeitouni	TUD
Marco Roveri	UniTN
Michele Grisafi	UniTN
Žiga Putrle	BEYOND
João Sousa	UMINHO
David Cerdeira	UMINHO
Sandro Pinto	UMINHO
Lukas Petzi	UWU

Document History			
Ver.	**Date**	**Change editors**	**Changes**
0.1	29/06/2023	M. Miettinen (TUD)	Initial Draft with Table of Contents
0.2	24/07/2023	M. Roveri (UniTN)	Initial complete draft of section 2.3. Initial draft of introductory section 3.
0.3	13/08/2023	David Cerdeira (UMINHO)	Initial draft of section 5.1.
0.3	15/08/2023	João Sousa (UMINHO)	Initial complete draft of section 4.1 (CROSSCON Stack Components).
0.4	18/08/2023	João Sousa (UMINHO)	Initial complete draft of section 5.2 (Instantiation Options for CROSSCON).
0.4	22/08/2023	M. Grisafi and M. Roveri (UniTN)	Finalized draft of introductory section 3, of sections 3.1 and 3.2.
0.5	25/08/2023	David Cerdeira (UMINHO)	Finalized draft of sections.1, 4.2, 5.1, 5.2.
0.6	31/08/2023	Žiga Putrle	Initial draft of section 6 (HW/SW Co-Design).
0.7	08/09/2023	D. Cerdeira and J. Sousa (UMINHO)	Add instantiation option (TEE Environment with FPGA assistance) to section 3.1.
0.8	14/09/2023	Žiga Putrle	Adding overview of TEE and REE in section Definition of key concept and terminology.
0.9	29/09/2023	Žiga Putrle	Updating the content of section 6 (HW/SW Co-Design).
0.10	03/10/2023	Marco Roveri	Fixed some missing references and addressed some comments on the initial specification of the properties.
0.11	22/10/2023	João Sousa	Check the fixes address comments in with D2.1_v0.1 and D2.1_v0.2; Revision in UMINHO sections;
1.0	31/10/2023	Hristo Koshutanski	QC and final version submitted.

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	ShazaZeitouni (TUD)	30/10/2023
Quality manager	Juan Alonso (ATOS)	31/10/2023
Project Coordinator	Hristo Koshutanski (ATOS)	31/10/2023

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
APU	Application Processing Unit
EC	European Commission
Dx.y	Deliverable number y belonging to WP x
HW	Hardware
ISA	Instruction Set Architecture
MCU	Micro-Controller Unit
MMU	Memory Management Unit
MPU	Memory Protection Unit
OS	Operating System
OTP	One Time Programmable Memory
PMP	Physical Memory Protection
RAM	Random-access Memory
REE	Rich Execution Environment
RoT	Root of Trust
RTOS	Real-Time Operating System
sPMP	S-Mode Physical Memory Protection, supervisor PMP
SMT	Satisfiability Modulo Theory
SW	Software
TEE	Trusted Execution Environment
VM	Virtual Machine
WP	Work Package
EL	Exception Level
SEL	Secure Exception Level
TA	Trusted Application

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 having very different levels of HW 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 HW platform to provide the best possible level of isolation for sensitive workloads. This document reviews the security assumptions upon which the CROSSCON approach builds and provides a high-level overview of the components CROSSCON is comprised of and possible instantiation alternatives depending on the capabilities that the underlying HW supports. By using this approach, CROSSCON is able to support a very large variety of different hardware platforms thereby enabling the use of the CROSSCON framework for different types of IoT devices. Apart from existing legacy HW platforms this document also explores the design of a CROSSCON-specific SoC design which aims at providing 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 first general picture of the overall specification of the CROSSCON architecture. The role of this document is to provide a draft description of the core components and instantiation options of CROSSCON and to introduce the formal modelling approach used. This document will be used as a basis for the technical work of the project and will be subsequently refined to a final open specification later in 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 D1.1 [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 extensions for accommodating domain-specific hardware in CROSSCON (WP4) D4.1 [4] will use 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 and implementation related to the CROSSCON stack will be detailed. Later, the draft specification will be updated to accommodate possible changes and insights gained during the project work and documented as a final version of the open specification in deliverable D2.3 [5].

1.3 Structure of the Document

This document is structured in seven major sections. After this introductory section, Sect. 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 Sect. 3, we describe the overall high-level architecture of CROSSCON and enumerate its different instantiation options based on the underlying hardware capabilities of the platforms on which CROSSCON is implemented. Sect. 4 then provides a detailed description of the individual architectural components comprising the CROSSCON architecture. In Sect. 5 we then provide the description of the interfaces of CROSSCON's components and describe in Sect. 6 the considerations regarding HW/SW co-design before concluding the document with Summary and Conclusions in Sect. 7.

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 minimalistic 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 for designing effective countermeasures. CROSSCON assumes a strong adversary that possesses considerable power to compromise system security. In particular, the adversary is able to inject malicious code into the kernel (EL1) 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. However, we may then reconsider side-channel attacks in future revisions of the formalization.

Denial-of-Service (DoS) Attacks are acknowledged 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's 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 behaviour across the device landscape. Requirements in this context serve as guiding principles that shape the architecture's ability to integrate seamlessly 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, encompassing features like cryptographic accelerators and security-oriented technologies like 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 become a cornerstone. These APIs serve as the lingua franca that enables different devices to communicate and interact seamlessly. In this context, the GlobalPlatform API [6] provides a solid foundation that can be extended and tailored to meet the specific needs of the IoT ecosystem. This approach promotes uniformity, reducing complexity and enhancing interoperability across the device landscape.

In summary, the architectural impact of requirements in 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

2.3.1.1 Trusted Execution Environment (TEE)

As part of the current CROSSCON specification, we adopt a similar view of a trusted execution environment (TEE) as used by the Global Platform - as described in the TEE System Architecture document [1]. We summarize how we use the term TEE in the following section.

A trusted execution environment (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.

We assume that a TEE is primarily used to provide a protected execution environment for trusted applications (TAs) and other system assets that are vital to the security of the system and are considered part of the RoT. We assume that the TAs inside of the TEE are isolated from each other by a trusted separation kernel or some other mechanism.

Because the assets inside of a TEE (trusted kernel, trusted applications, etc.) are trusted, we also require that they are functionally correct, meaning that they provide the expected behavior.

We view the TEE as a dual environment to the rich execution environment (REE) where the rest of the functionality that is less critical for the security of the system resides, for example, rich operating systems like Linux and FreeRTOS with other services related to the functionality of the system.

Although we do not put any restrictions on how the guarantees provided by TEE are enforced (via software, hardware, or a combination of both), it's typically presumed that these guarantees are enforced with the help of some underlying isolation mechanism provided by hardware, for example, TrustZone.

We acknowledge that there are also other definitions of a TEE used in the literature and across the industry; for example, as used by ReZone [7], Keystone [8], Komodo [9], and Sanctuary [10] in research community and by Intel SGX (enclave isolation), Hex Five's MultiZone and SiFive's WorldGuard (hardware isolated environments) and AMD SEV, Intel TDX, Armv8.4 Secure Hypervisor, Arm CCA, RISC-V CoVE (VM isolated environments) in the industry. We note the following differences:

- We assume that the assets inside of the TEE are trusted, where this is not required by some other definitions.
- We assume that there is only one TEE alongside the REE when TEE is available. Other definitions often do not make any assumptions regarding the number of TEEs available on the system. Therefore, a system could have several TEEs that are independent of each other in a sense that if one of them is compromised the other TEEs are not affected.

We plan to revise the definition of the TEE that we use in the near future so that it will reflect some of the most recent work related to TEEs and advances done as part of the CROSSCON project.

Regarding the formal description of a TEE, we rely on the individual (formal) guarantees provided by the TEE on case-by-case basis and describe them when used in the formalization effort. It is important to note, that some of the guarantees might also very depending on the underlying implementation and hardware support.

Note that CROSSCON stack also allows creation of execution environments with strong security guarantees in the REE, called virtual machines (VMs), by using the CROSSCON Hypervisor and classical HW mechanism used for virtualisation (privilege levels and MMU/MPU, etc.).

See Sect. 4.1.3 for further explanation.

2.3.1.2 Rich Execution Environment (REE)

As part of the current CROSSCON specification, we view rich execution environment (REE) as a complex and rich environment for general computation tasks including operating systems, like Linux and FreeRTOS, and applications running on top.

We often view REE as a dual environment to TEE where TEE contains only parts of the system that are critical to the security of the system and are often considered as part of the RoT and REE contains parts of the system that are used for more general computation.

Note that, CROSSCON stack also allows creation of execution environments with strong security guarantees in the REE, called virtual machines (VMs), by using the CROSSCON Hypervisor and classical HW mechanism used for virtualisation (privilege levels and MMU/MPU, etc.).

See Sect. 4.1.4 for further explanation.

2.3.2 General Objectives

Here, we report the general objectives. Specific objects for the different specializations are reported in the respective section later on.

1. CROSSCON will support the presence of a TEE. If it is present, we may decide to use it.
2. CROSSCON will guarantee the integrity and confidentiality of data while supporting services operation;
3. CROSSCON will support more than one OS executing on the same device, depending on the characteristics of the device and the ISA itself.
4. CROSSCON will support the presence of an FPGA. If it is present, we may decide to use it.

2.3.3 General Assumptions

Here we report the general assumptions. Specific assumptions for the different specializations are reported in the respective section later on.

1. CROSSCON assume a TEE is primarily used to provide a protected execution environment for trusted applications (TAs)
2. Because the assets inside of a TEE (trusted kernel, trusted applications, etc.) are trusted, CROSSCON assume that they are functionally correct; meaning that they provide the expected behaviour.
3. 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.
4. Software running in a TEE is trusted and can, therefore, coexist with the REE components.
5. We assume 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.

2.3.4 General Constraints

Here, we report the general constraints. Specific constraints for the different specializations are reported in the respective section later on.

1. We support, for the time being, RISC-V, some versions of the Arm ISA, and some versions of the MSP430 ISA.
2. On bare metal systems, we offer a limited set of features compatible with the underlying architecture.

2.4 Approach to Formal Verification

As far as the formal verification is concerned, we analysed different specifications available in the literature: Sel4, CHERIoT, and GWV, to identify the different definitions of isolations adopted in each of them and then decide the one to adopt within our project. In the following, we briefly summarize each of them, starting from Sel4, CHERIoT, and finally GWV.

Sel4 security properties

Sel4 is a general-purpose operating system microkernel whose specification has been subject to machine-checked formal verification [11], and its implementation has proven to be functionally correct according to the specification. Sel4, among other things, guarantees the three classical security properties: confidentiality, integrity, and availability.

- *Confidentiality*: Sel4 will not allow an entity to read data without having been explicitly given read-access to the data.
- *Integrity*: Sel4 will not allow an entity to modify data without having been explicitly given write-access to the data.
- *Availability*: Sel4 will not allow an entity to prevent another entity's authorised use of resources.

We notice that, from a thorough reading of the documentation, the proofs for guaranteeing the above security properties do not consider time. Thus, the above security properties are static and not associated with time. Moreover, all the proofs leverage on the following assumptions [12]:

- The hardware operates as intended, meaning we assume it functions correctly by adhering to its specifications. In practical terms, this assumption entails that the hardware has not been tampered with and is operating within its designated conditions.
- the theorem prover is correct, i.e., we can trust the solver used to prove the properties, which is bug free, and the proof rules used to prove the properties are correct.

There are also other assumptions, and we refer the reader to [12] for additional details.

CHERIoT security properties

CHERIoT (Capability Hardware Extension to RISC-V for Internet of Things) [13] is an ISA and software model built on top of CHERI [14] [1]and RISC-V to provide spatial memory safety, deterministic use-after-free protection, and lightweight compartmentalization exposed directly to the C/C++ language model. CHERIoT can run existing embedded software components on a clean-slate Real-Time Operating System (RTOS) that scales up to large numbers of isolated (yet securely communicating) compartments. Thus, CHERIoT considers notions of *memory safety*. In their settings, a system is said to be *memory safe* if its references to memory are:

- *Unforgeable*: A reference to memory (in particular, the authority to access memory) can be constructed only from other references.
- *Monotonic*: A constructed reference will have no more authority than its progenitor reference(s) (and may have less).
- *Spatially Safe*: References to memory authorize access to a set of memory locations determined when the reference is constructed.
- *Temporally Safe*: References to a region of memory will not remain usable across reuse of memory for a different allocation.

CHERIoT leverages the notion of compartment: a collection of code, data, and capabilities that serves as an invocable security context. Given these concepts, it defines two global security properties:

- No compartment should be able to access another compartment's data, except where explicitly shared.
- No thread should be able to access another thread's data, except where explicitly shared.

We remark that, although CHERI was also applied to other architectures, for example, MIPS and Arm, CHERIoT is tight to RISC-V architectures only, and extension to other families is not trivial. We remark that being CHERIoT based on RISC-V, the proposed approach is tight to the RISC-V processors only, and extension to other families is not trivial. We remark that, CHERIoT targets embedded devices that do not support virtualization of the memory in the classical sense; thus, CHERIoT devices do not support classical operating systems (e.g., Linux), while CHERI, differently from CHERIoT, considers virtualization of the memory. In the case of the CROSSCON stack, we aim at covering both cases (with classical memory virtualization and without).

GWV security properties

Greves, Wilding and Vanfleet (GWV) [15] proposed a security policy that defines a model of a separation kernel, which enforces partitioning between applications running on a single processor system. To this extent, GWV defines three basic separation axioms:

- *Non-exfiltration* indicates that an executing partition will not influence memory segments outside of its permitted set of segments.
- *Non-infiltration* indicates that the executing partition can only use information from its permitted set of segments to affect its execution behaviour.
- *Non-mediation* indicates that when a partition executes, the effect on a segment does not depend on anything other than the segment's original value and the values of the current partition.

GWV security properties are very general and different from the other two considered previous cases they encompass time (execution).

The CROSSCON properties

For the CROSSCON approach, we will adopt the Sel4 classical security properties: confidentiality, integrity and availability, and in the formalization, we will provide formal counterparts of the following natural language security properties.

[1] The RISC-V open source project https://riscv.org/

- *Confidentiality:* CROSSCON will not allow the reading of data without having been explicitly given read access to the data.
- *Integrity: CROSSCON will not allow the modification of data without having been explicitly given write access to the data.*
- *Availability:* CROSSCON will not prevent authorised use of resources. This is similar to the case of Sel4, meaning that the different CROSSCON elements will eventually get the resource needed if they are authorized to get it.

This choice is motivated by the fact that these properties are not too generic as the one in the GWV model, and not too specific to RISC-V as the one of the CHERIoT. Similarly, to the Sel4 case, we will base our formalization on the following assumptions.

- Unless strictly necessary we keep the time out of the formal specification, thus considering static properties.
- We will assume the HW to behave correctly, but we can prove that some HW components developed (if any) in the project behave correctly.
- To start with, we assume no side-channel attacks. We may then reconsider this in future revisions of the formalization [16].
- We assume the verification engine is correct (I.e., it does not allow to conclude false results).

For the formalization we will follow the architecture decomposition of the CROSSCON stack leveraging on adaptation of existing verification techniques, e.g., model checking [16], theorem proving, e.g., Satisfiability Modulo Theory (SMT) [17] and compositional reasoning [18].

3 High-level Architecture

In the project proposal, we considered the following high-level architecture for the CROSSCON stack. In Figure 1. The CROSSCON open security stack considered in the proposalthe green components (corresponding to the new ones) 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.

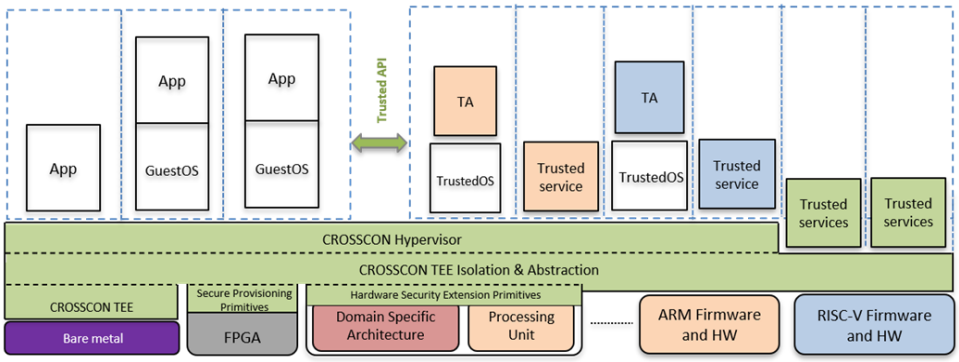

Figure 1. The CROSSCON open security stack considered in the proposal

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

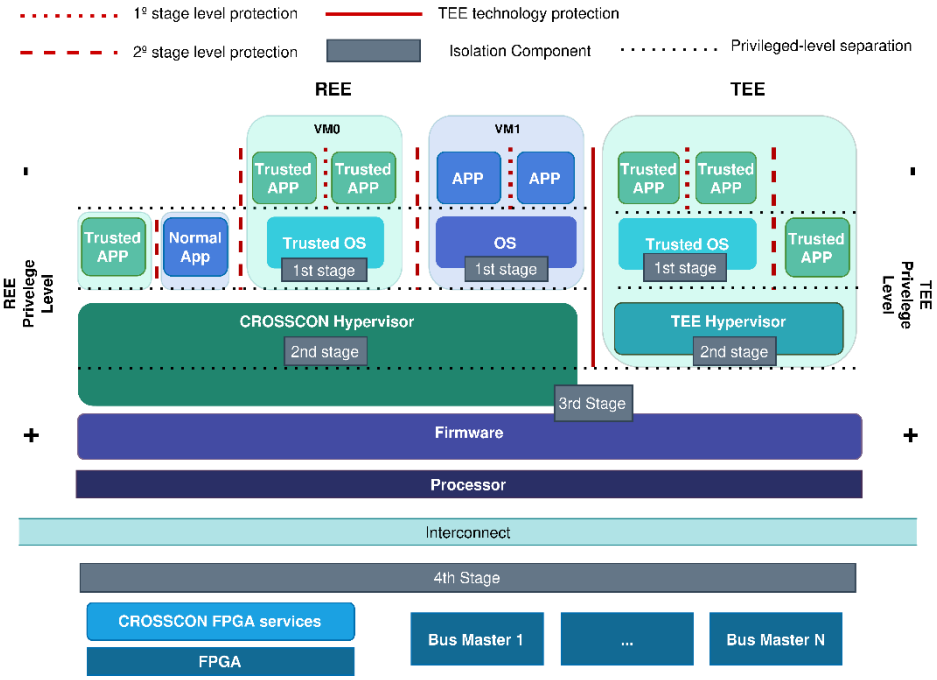

Figure 2. The refined CROSSCON stack

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 ISA that varies depending on the CPU architecture families (E.g., RISC-V, Arm with all their variants) considered, each equipped with possibly different capabilities (E.g., the presence of a TEE, privilege levels). On top of the ISA, we consider the possible presence of firmware (that 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, thus being 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 and an untrusted application since both enjoy the same level of protection, being completely isolated. Furthermore, VMs can either consist of a single application (normal or trusted application) or more complex software, comprising both an OS (or trusted OSes) and its applications (trusted applications) (see Figure 2). In the first case, the VM is executed in one of two privilege levels above the hypervisor. In the second case, the VM leverages the bottom two privilege levels, thus allowing the VM OS to manage the resources for its applications. In this case, the OS will establish a first stage isolation between the various applications: each will be executed on a virtual set of isolated resources. The VM OS will then handle the virtualization, translate addresses, and allocate the rest of the resources accordingly. Notably, this level of virtualization is a further abstraction than the one established by the hypervisor (the only software with full access to the physical resources).

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 will be bootstrapped on demand by the hypervisor and then destroyed when no longer needed. All communication between different VMs must go through the hypervisor, which securely handles the communication.

If the underlying architecture is equipped with advanced isolation primitives (e.g., TrustZone), the CROSSCON Hypervisor will leverage them to create a separate TEE. The TEE will operate on three different privilege levels, allowing different trusted OSes to co-exist under the supervision of a TEE hypervisor. Similar to the CROSSCON Hypervisor, the TEE hypervisor will handle the 2nd stage isolation in the TEE, offering a virtual view of the system resources to the various Secure OSs. As in the first case, a Trusted Application (TA) can either be self-contained or be comprised in an OS. In this case, they will be offered another virtual view of the system resources (1st stage). Notably, in this scenario, the CROSSCON Hypervisor will also require running partially on the firmware level to supervise the communication between the REE and the TEE.

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.

Figure 2 is a high-level representation of the CROSSCON architecture on a device fully equipped with the latest and most advanced security hardware. However, CROSSCON strives to achieve a similar level of protection (depending on the hardware capabilities) on less powerful architectures, on which only a subset of the above features is implemented. In the following sections, we will oversee how CROSSCON tackles real scenarios. Before going into the architecture's details, we discuss the objectives, assumptions and constraints our work builds on.

Objectives:

- CROSSCON aims to support different instantiation options while covering different privileged levels, and ensuring the security between components of the system stack;
- CROSSCON will support different combinations of privileged levels, including the use of fewer privileged levels than the outlined in Figure 2, depending on the class of device and ISA itself.

Assumptions:

- There are three REE privilege levels: EL2 (hypervisor level), EL1 (OS Kernel level), and EL0 (application level).

- There are three TEE privilege levels: SEL2 (secure hypervisor level), SEL1 (secure OS kernel level), and SEL0 (secure application level).
- There is a fourth level, the EL3 (machine level), where, in some ISAs, the firmware is executed.
- CROSSCON assumes different ISA's privileged levels. In most cases, they follow the hierarchy: EL3 > SEL2 > SEL1 > SEL0 > EL2 > EL1 > EL0.
- Depending on the ISA and the class of the device, the CROSSCON hypervisor will operate on the suitable privileged level.
- The CROSSCON stack assumes four stages of isolation;
- In instantiations that include TEE or virtualization, the correct configuration of the 4th stage isolation is assumed, thus assuring that other bus masters cannot attack undue system components.
- CROSSON will assume that CROSSCON Hypervisor will provide 3rd stage isolation only when TrustZone-M is present. Otherwise, it will provide 2nd stage isolation.
- CROSSCON assumes that there is only one TEE alongside the REE when TEE is available.

Constraints:

- The provisioning of Trusted VMs is only available when the CROSSCON Hypervisor component is present.

3.1 Architecture Description

The CROSSCON system stack could comprise several combinations of the components mentioned in Sect. 4, which could lead to different instantiation options. A single environment in the CROSSCON system stack could go from a simple application running in an MCU platform encompassing just one privilege level to a complex system stack operating in an APU platform encompassing multiple trusted environments around four privilege levels. Table 1 demonstrates the possible instantiation options that CROSSCON will consider.

Table 1: CROSSCON's possible instantiation options

	Virtualization	Trusted VM	TEE	FPGA
TEE- and Virtualization -less environment	-	n/a	-	-
TEE-less environment with Virtualization	x	-	-	-
TEE-less environment with Virtualization and Trusted VMs	x	x	-	-
Virtualization-less environment with TEE	-	n/a	x	-
Environment with TEE and Virtualization	x	-	x	-
Environment with TEE, Virtualization, and Trusted VMs	x	x	x	-
Environment with Virtualization and Trusted VMs with FPGA assistance	x	x	-	x

Every instantiation option hinges on the presence of the two most privileged components to support virtualization and TEE technologies, either together or independently. In instantiations that include TEE or virtualization, the correct configuration of the 4th stage isolation is assumed, thus assuring that other bus masters cannot attack undue system components.

3.1.1 TEE- and Virtualization-less Environment

Numerous hardware-assisted security architectures have emerged as potential solutions to counter the escalating cyberattacks targeting internet-connected devices. Thus, a significant challenge arises from the incompatibility of these proposals with a considerable portion of the currently deployed embedded devices, particularly those constrained by resources (e.g., low-end devices). These devices are primarily engineered for attributes like low cost, small size, and low power consumption. Since

MCUs often do not have basic hardware-based memory safety features such as MPU, it leaves them more vulnerable to malware infestation attacks.

The TEE- and Hypervisor-less environment instantiation option within CROSSCON is tailored to address the security requirements of resource-constrained low-end devices. The goal is to provide a security solution that isolates applications running on these devices. The visual representation presented in Figure 3 depicts an environment with minimal components – comprising the applications and the CROSSCON bare metal TEE – without any intervening privileged level between them.

In this scenario, CROSSCON adopts a software-based methodology to ensure isolation between normal applications and trusted applications. This approach entails the PISTIS implementation [19], marking the inception of a comprehensive software-based trusted computing architecture. PISTIS extends vital features, including memory isolation, remote attestation, and secure code update services, all fortified by robust security assurances.

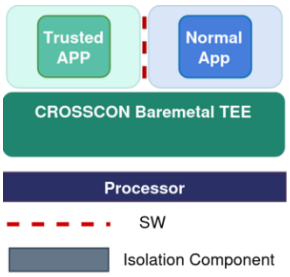

Figure 3. A TEE- and virtualization-less environment

3.1.2 TEE-less Environment with Virtualization

In a TEE-less setup with virtualization, the CROSSCON system stack layout aligns with Figure 4 by inserting the hypervisor component, thus achieving consolidation and integration of various systems onto the same hardware platform. In this CROSSCON instantiation option, the hypervisor component sits above the firmware, managing guests and attributing physical resources to them.

To isolate guest resources, the hypervisor uses isolation components categorized in the second isolation stage, depending on the processor type. In APU processors, the hypervisor will leverage virtual memory, while MCU families will leverage hardware security primitives such as 2nd stage MPU or PMP. Regarding the isolation in the above layers, i.e., user-level applications, the guest kernel component isolates them with isolation components categorized in the first isolation stage, e.g., Linux OS leverages MMU to isolate processes.

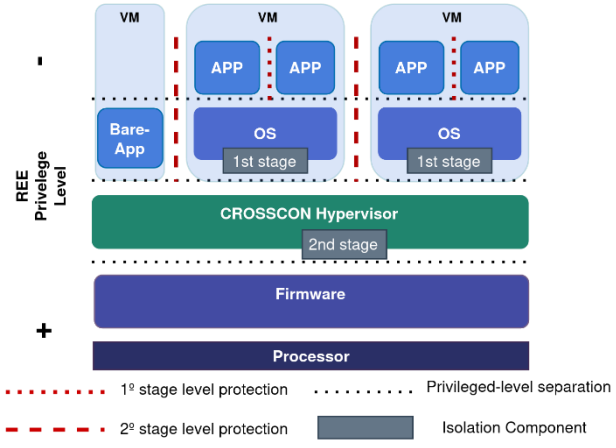

Figure 4. A TEE-less environment with virtualization

3.1.3 A TEE-less Environment with Virtualization and Trusted VMs

This instantiation option is not too different from the previous one. As shown in Figure 5, it differs from the previous by bringing the concept of trusted applications (TAs) to the normal world. In this instantiation option, the CROSSCON Hypervisor enables the maintenance of isolated lightweight VMs that protect security-sensitive applications from being compromised by malicious parties. Figure 5 illustrates the coexistence of regular guests (VMs with blue background) and trusted guests (VMs with green background), both operating atop the CROSSCON Hypervisor. Both normal and trusted VMs distinguish between each other through some implementation details at hypervisor level.

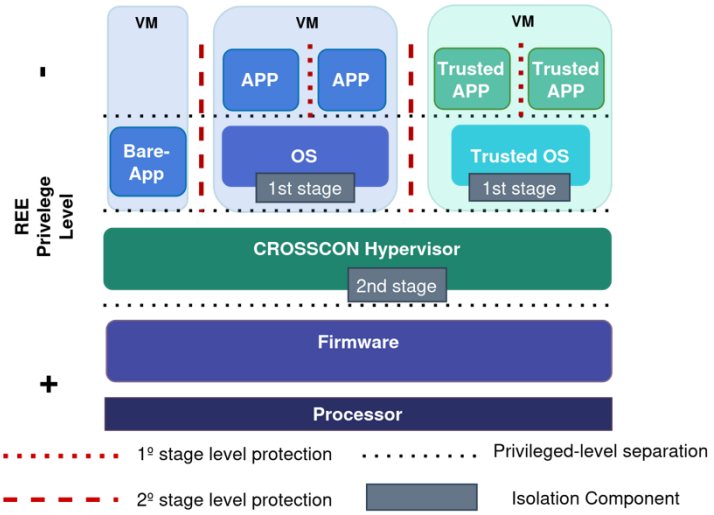

Figure 5. A TEE-less environment with virtualization and TEE VMs

3.1.4 A Virtualization-less Environment with TEE

In contrast to the virtualization-enabled instantiations, this CROSSCON instantiation option focuses on a setup without a hypervisor while integrating existing TEE components. As previously highlighted, a TEE plays a crucial role in ensuring the integrity and confidentiality of sensitive data. Different architectures incorporate distinct iterations of TEE technology, each uniquely suited to their characteristics. Within the realm of Arm processors, TEE implementation exhibits variability. For instance, in APUs like Cortex-A, a dual-world TEE is established by setting a CPU bit, and the transition between worlds is managed through the Secure Monitor. Conversely, MCUs such as Cortex-M adopt a distinct approach characterized by its low-cost and low-performance nature. This design prioritizes efficiency within resource-constrained environments with only two privileged levels (privileged and non-privileged). It eschews the inclusion of a separate privilege level for monitor software, as shown in Figure 6.

When deciding on this instantiation option and including TEE technologies, it's important to note that the placement of components will look different depending on the architecture and the way they are implemented, for example, whether in Arm Cortex-A or Cortex-M. CROSSCON aims to address this heterogeneity by accommodating various TEE implementations across a broad spectrum of platforms, not just for Arm but also for RISC-V and potentially other architectures.

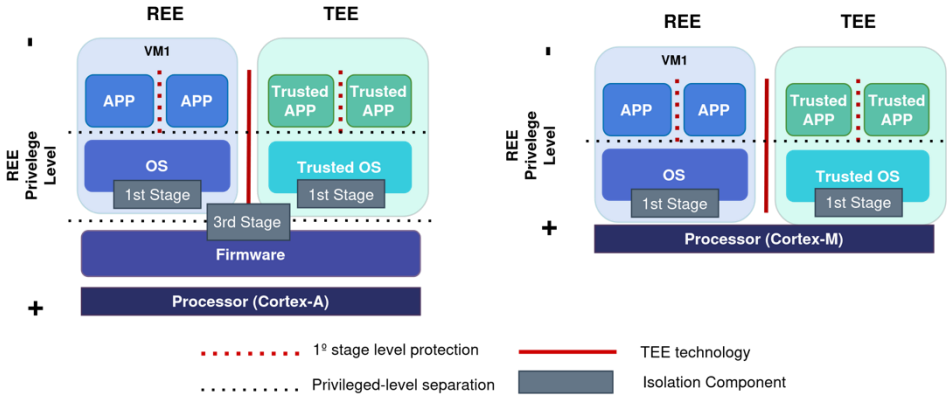

Figure 6. A virtualization-less with TEE component

3.1.5 Environment with TEE and Virtualization

This CROSSCON instantiation option introduces the combination of hypervisor flexibility combined with TEE trustworthiness, as depicted in Figure 7. In contrast to prior instantiations, this model encompasses all isolation stages, spanning the entire system stack with all types of components, including TEE technologies (include 3rd isolation stage components) alongside a hypervisor (include 2nd isolation stage components), supporting multiple guests at the kernel level, with its user-level applications (1st isolation stage components). Similar to the previous instantiation, CROSSCON considers widely adopted and resource-capable implementations, such as Arm's Cortex-M and Cortex-A processors.

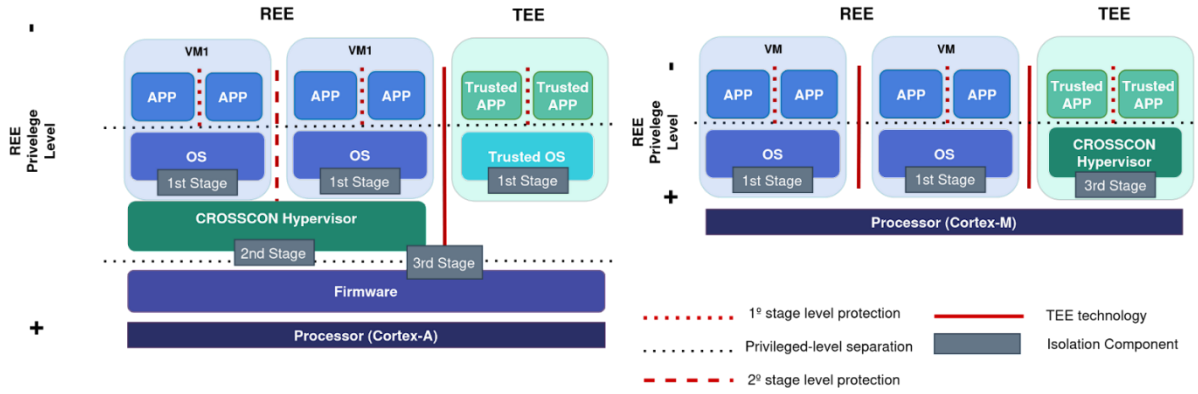

Figure 7. Environment with TEE with virtualization

3.1.6 Environment with TEE, Virtualization, and Trusted VMs

To ensure the inclusion of trusted applications within the non-confidential zones, this instantiation option within CROSSCON augments the capabilities of the hypervisor component, even in the presence of TEE technologies within the system. The goal is the same as the instantiation option that covers trusted VMs, Sect. 3.1.3, i.e., enabling the maintenance of isolated lightweight VMs that protect security-sensitive applications from being compromised by malicious parties. Figure 8 shows two possible combinations in two different families, i.e., in an APU (e.g., Arm Cortex-A) and MCU (e.g., Arm Cortex-M) families, covering trusted VMs managed by the hypervisor.

The ability to make the hypervisor manage trusted applications in non-secure worlds, rather than being overseen by the platform manufacturer, i.e., by the TEE technologies, makes the CROSSCON adaptable to more combinations of technologies and envision all kinds of possibilities that a computer system can have to guarantee security. Additionally, it allows developers to move complex functionality from the TEE, reducing its TCB.

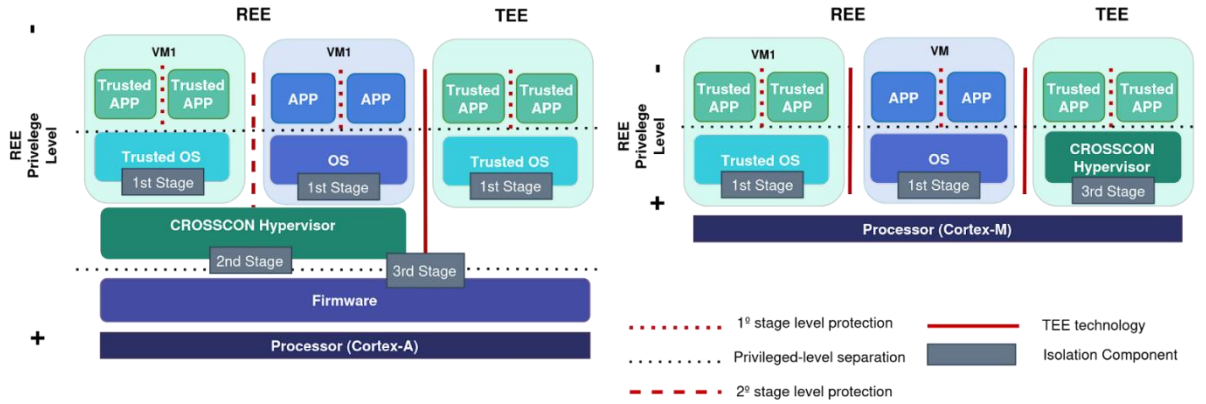

Figure 8. Environment with TEE, virtualization, and TEE VMs

3.1.7 Environment with Virtualization and Trusted VMs with FPGA Assistance

To not preclude the reconfigurable hardware presence, this CROSSCON instantiation option broadens the possibilities for system composition, motivated by a potential increase in adoption of reconfigurable platforms in IoT use cases. Through this instantiation option, CROSSCON demonstrates an awareness of the interface with these components and carefully considers the security implications.

Some TEE technologies already can handle the secure integration of FPGA. TrustZone, for instance, enables the redistribution of memory regions as well as the redistribution of peripheral configuration registers between normal and secure worlds, ensuring that only authorized secure masters can access memory regions marked as secure.

However, to support scenarios where we aim to create additional secure environments, we leverage the IOMMU, SMMU (System Memory Management Unit) equivalent to an ARM terminology. IOMMU serves to achieve memory and peripheral isolation, limiting the address space accessible to I/O devices and thereby safeguarding the OS against, more specifically, bus master attacks, which could originate from the FPGA. For example, the ZU+ MPSoC features the Xilinx Peripheral Protection Units (XPPU) and Xilinx Memory Protection Units (XMPUs). The XMPU facilitates the dynamic redistribution of memory regions among various masters, while the XPPU redistributes peripheral configuration registers among different masters. In the context of CROSSCON, the IOMMU, and IOMPU enable the CROSSCON Hypervisor to prevent arbitrary memory and peripheral accesses from the platform's bus masters. These assets ensure security, enabling the seamless allocation and reallocation of resources across the established environments and enhancing overall flexibility within the system.

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 Sect. 3. In the following, each of these stack components is described in more detail. We also provide in Sect. 4.2 a description of the new CROSSCON trusted services providing higher-level security functionalities to applications utilising the CROSSCON architecture.

4.1 CROSSCON Stack Component

This section provides a comprehensive description of each potential architecture component within the system stack, including isolation components. System stack components are presented in a bottom-up order, starting from the Firmware component, which may exist or not depending on the CPU architecture and the needs of the above layers of the stack. Proceeding further, the Hypervisor component for virtual environment's support among several high-end and low-end platforms. At the same level of consideration, the TEE components for establishing trusted zones, supporting trusted services operations and ensuring data confidentiality and integrity. Subsequently, the REE components for non-critical operations support. Lastly, the confidential computing technology components as potential supplementary elements of CROSSCON architecture at the designing time. Before going into the CROSSCON Stack Component's details, we describe the objectives, assumptions, and constraints our work builds on.

Objectives:

- CROSSCON aims to cover all possible components in the system stack.
- CROSSCON aims the utilisation of isolation mechanisms to protect system bus accesses, going from CPU accesses, DMA engines and auxiliary processing elements.
- Due to the nature of microcontrollers, CROSSCON aims to replace virtual memory protection mechanisms (e.g., MMU) with physical memory protection mechanisms (e.g., MPU).

Assumptions:

- CROSSCON assumes that each stack component operates in the proper exception level that was designed for.
- CROSSCON assumes that HW components of the device behave correctly.
- CROSSCON assumes that the TAs inside of the TEE are isolated from each other by a trusted separation kernel or some other mechanism.
- CROSSCON Stack assumes that communications between worlds are trusted, either with their reliance on a standard between the trusted OS and the hypervisor.

Constraints:

- Some CROSSCON components could depend on recent ISAs with few/no support (e.g., TEE Hypervisor, Arm Realms).

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, system-on-chip (SoC) devices, or other electronic devices. The firmware acts as an intermediary between the hardware and the operating system or 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) [20] is the reference firmware implementation for a myriad of devices. This firmware implementation serves as a universal reference implementation for the Trusted Board Boot Requirements (TBBR). TF-A offers a standardized approach encompassing hardware initialization, memory setup, secure boot implementation, and secure

firmware updates, and provides the Secure Monitor for secure switching between the secure world (TEE) and the normal world (REE).

When delving into RISC-V technology, the prevailing choice for open firmware implementation is OpenSBI [21]. OpenSBI provides features like processor initialization, memory configuration, and the integration of 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 CROSSCON Hypervisor

Community efforts have underscored the potential of hypervisor components in overseeing trusted components, even extending their integration into the REE environments. Some studies have demonstrated that it is possible to leverage hypervisors to provide lightweight virtual machines in the normal world and allow developers to relocate complex functionality from the secure world, reducing the system TCB [22, 10].

In the new CROSSCON abstract model, Figure 2, situated immediately above the firmware lies the hypervisor component. The hypervisor's ultimate objective is to create and support distinct and isolated VMs, ensuring they run as if they were operating independently on separate hardware. The hypervisor operates within a dedicated layer, with higher privileges than the OS, safeguarding hardware resources and leveraging various isolation mechanisms.

For effective resource isolation, including shared architectural resources like last-level caches, interconnects, and memory controllers, the hypervisor may adapt its security features according to the underlying hardware. Its design varies based on the specific ISA it operates on (see Sect. 3). For example, in APU processors, the hypervisor leverages virtualization extensions, including 2nd stage translation and other virtualization techniques, to establish isolation between virtual machines. In contrast, in simple MCU processors, the hypervisor employs components that restrict access to physical memory for each guest without the need for hardware mechanisms.

CROSSCON intends to provide a hypervisor that allows hardware resources to not be shared across VMs and provide a set of built-in mechanisms (e.g., cache coloring) to guarantee strong isolation not only at the architectural but also micro-architectural level. The idea is to complement a TEE architecture with a thin static partitioning hypervisor layer to achieve enhanced isolation and security guarantees through a micro-kernel-like design. The hypervisor will ensure the correct enforcement of the access control policies to guarantee that VMs can securely execute security-sensitive workloads, for example, running a Trusted OS and TAs. Addressing the lack of flexibility to dynamically create and manage new VMs and services on static partitioning hypervisors is the objective of task T2 of WP3.

4.1.3 Trusted Execution Environment (TEE)

Trusted Execution Environments (TEE) are security solutions designed to safeguard application integrity and confidentiality. They utilize dedicated hardware to run security-sensitive applications in isolated domains, distinct from the host platform's OS.

The firmware component has a crucial role in setting up the necessary hardware features that enable the separation of the TEE and the REE. Firmware ensures that these worlds are separated in memory and that access controls are in place to prevent unauthorized access from one world to the other.

A prominent TEE technology addressed by CROSSCON is ARM TrustZone, widely adopted in mobile, industrial, server, and low-end devices. ARM TrustZone introduces a dual-world paradigm: the "normal world", a Rich Execution Environment (REE), and the "secure world", the TEE. In the normal world, the hypervisor, OSes, and numerous applications operate, providing an ideal for general computing tasks. Conversely, the secure world appears as an isolated processing environment where applications execute securely, independently, and detached from the normal world.

TEEs have become increasingly important to ensure the confidentiality and integrity of computations, especially in the context of IoT systems, where diverse devices coexist. Because of this, achieving a

security baseline across the entire IoT system is a significant challenge. Different devices use varying hardware, each implementing often proprietary instances of security features (e.g., Intel SGX, Trustonic TEE, Qualcomm TEE), making it difficult for their security services to interoperate seamlessly. To address these challenges, the CROSSCON abstract model, Figure 2, seeks to overcome TEE deficiencies and ensure the interoperability of TEE implementations across different ISAs. Additionally, by bridging the gaps between various TEEs, CROSSCON aims to provide additional isolation capabilities by enabling the decomposition of trusted services from being hosted by a single TEE kernel, through isolation components enumerated in Sect. 4.1.6. Specific design details about TEE isolation will be further described in Deliverable D3.1.

The CROSSCON abstract model addresses the inclusion of TEE components, such as trusted application (Trusted App), trusted kernel (Trusted OS), and in some architectures (e.g., RISC-V), it addresses scenarios with TEE hypervisors (e.g., CoVE mention in Sect. 4.1.5).

Trusted application components, as the name suggests, 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.

The trusted kernel operates with a supervisor, i.e., OS level privileged level within the Secure World, and it plays a vital role in the Trusted Execution Environment (TEE). It provides fundamental OS primitives for scheduling and managing Trusted Applications (TAs), implements device drivers for trusted peripherals access, handles cross-world requests securely, and incorporates shared libraries and TEE primitives, such as remote attestation, trusted I/O, and secure storage. These components ensure a robust and secure TEE environment for critical operations and data protection.

Overall, CROSSCON intends to address a wide range of TEE implementations, covering as many implementations of TEE components, i.e., trusted kernels and trusted applications as possible. The abstract model in Figure 2 illustrates that CROSSCON will mainly focus on Arm and RISC-V technologies considering low- and high-end embedded devices.

4.1.4 Virtual Machines (VM)

As previously mentioned, the REE is a complex and rich environment for general computing tasks encompassing all well-known system components depicted in the CROSSCON abstract model, i.e., Operating Systems (OSes) and their applications (Apps). In CROSSCON, these workloads run in VMs when the CROSSCON Hypervisor is used. In the absence of a hypervisor, they are "stand-alone" elements and run without the hypervisor access control checks. These elements include bare-metal applications and OSes.

The OS runs at a supervisor-privilege level in either a guest or stand-alone manner. It manages tasks such as deploying drivers for seamless hardware communication, thread management, network, and inter-process communication (IPC).

Without the intervention of OSes, bare-metal applications operate at the supervisor- or user-privilege level (depending on ISA), providing simple services either in a stand-alone or guest manner.

CROSSCON considers all possible scenarios involving OSes operating alongside bare-metal applications. This approach ensures a comprehensive integration of different system configurations, which will be further discussed in Sect. 5.3.

4.1.5 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 virtual machines from potential attacks launched via higher-privileged components like operating systems and hypervisors. CoVE builds on the concept of a TEE Virtual Machine (TVM), which mirrors the approach taken by Realm, where non-TEE elements (hypervisor components) continue to manage confidential components. This management is performed by the TEE security monitor, a component located at the hypervisor privileged level of the TEE side, acting as an intermediary between the hypervisor and TVMs. Importantly, it is endowed with the capability to manage, execute, and destroy TVMs, 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 operating systems and hypervisors. It's important to note, however, that these technologies are relatively new, and as a result, current commercial off-the-shelf (COTS) compatibility is lacking.

Considering their vital relevance to the project's objectives and the projected timeline, the CROSSCON abstract model thoughtfully reserves space for their integration. This strategic decision acknowledges their potential impact and sets the stage for their eventual application as the technology landscape evolves. A deep investigation will be considered for the revised version later in the project.

4.1.6 Isolation Components

Hypervisor and TEE technologies have showcased their significance in establishing distinct isolated environments across various ISAs. The scope of these technologies is tightly coupled to the architecture they engage with, utilizing diverse isolation components to establish secure barriers between them.

The CROSSCON stack assumes four stages of isolation, as depicted in Figure 2. Each isolation stage contains mechanisms to create isolation between components at lower privileged levels. This tiered approach involves applying the 1st isolation stage to establish separation between applications at the user-privileged level. The second isolation stage is to isolate multiple kernel instances running at the supervisor-privileged levels. The third isolation stage is to classify isolation components that facilitate TEE provisioning. Lastly, the fourth isolation stage operates at the interconnect layer, isolating access from various master sources (e.g., DMA). This approach of categorizing each stage of isolation component accommodates both MCU and APU platforms running different versions (e.g., Armv6, Armv7, Armv8).

The CROSSCON hypervisor aims to ensure the secure coexistence and operation of VMs, with isolation strategies tailored to the specific characteristics of MCU and APU core families. Starting with MCU families, they lack virtual memory translation mechanisms, i.e., guests run with direct access to physical memory. Hence, to ensure isolation between guest components in MCU families, hypervisors leverage hardware security primitives to restrict each guest component's access to physical memory. CROSSCON recognizes that Arm and RISC-V MCUs employ as isolation components the Memory Protection Unit (MPU), 2nd stage MPU, Physical Memory Protection Unit (PMP), and Supervisor Physical Memory Protection Unit (sPMP). Conversely, in APU families, where memory translation is present, hypervisors redistribute and virtualize physical memory to each guest that maintains. For Arm and RISC-V APUs, mechanisms like MMU and 2nd stage MMU facilitate this functionality.

Notwithstanding, each of the aforementioned components plays an isolation role at the kernel and application levels, including both the 1st and 2nd isolation stages of isolation components.

TEEs ensure the integrity and confidentiality of data while supporting security services operations in isolated zones. Different isolation mechanisms serve APU and MCU core families. CROSSCON abstract model shows that Arm employs TrustZone for both APU and MCU families (note that the instantiation of TrustZone in both these cases is significantly different), whereas RISC-V utilizes PMP as an isolation component. These components encompass the isolation between trusted and non-trusted environments, i.e., the third stage of isolation components. CROSSCON will support the presence of a TEE. If it is present, we may decide to use it.

CPUs are the central element of a system; however, other components that access system resources coexist to offer varying degrees of functionality. From simple DMA engines to auxiliary processing elements, e.g., GPUs or FPGAs, there is a wide range of bus masters (their name is due to being able to initiate transactions on the system bus) that can represent security threats. To protect against them, TEE technology extends to the system bus and bus master themselves to be able to distinguish between accesses from trusted and untrusted bus masters. On virtualization-enabled systems, IOMMUs can be used to a similar effect. With IOMMUs, it is possible to restrict bus master access to system resources, for example, to restrict that a DMA device is only able to access a specific VM memory region.

Following is a description of each isolation component that CROSSCON addresses. It's important to note that this document is subject to change, so additional isolation components may be included as more architectures, designs, or implementations are explored.

1. **Memory Management Unit (MMU)** - MMU is a hardware component found in complex computer systems (in APUs) that plays a crucial role in translating virtual memory addresses used by low-privileged software into physical memory addresses. Typically, it performs at the kernel level (1st stage of isolation component), allowing programs to operate without needing to know the actual physical location of data. This isolation component enhances memory protection by enabling each process to have its own isolated memory space while sharing the physical memory resources.
2. **Memory Protection Unit (MPU)** - MPU is a hardware component found in some MCU architectures that provides a level of memory protection and access control. It helps ensure the security of a system by enforcing restrictions on how different parts of memory can be accessed by software. The MPU defines specific memory regions with distinct access permissions, preventing unauthorized or unintended memory accesses. Unlike an MMU, which primarily focuses on virtual-to-physical memory address translation, an MPU focuses on controlling access rights and ensuring the isolation of memory regions, enhancing system security.
3. **MMU and MPU (2nd stage)** - An advanced evolution of memory management, both the MPU 2nd stage and MMU 2nd stage are isolation components that extend the capabilities of their respective base technologies at the hypervisor level (2nd stage of isolation components). These enhanced stages add an extra layer of enhancing memory access control and translation, improving security and efficiency in a system.
4. **Physical Memory Protection (PMP)** - The PMP in RISC-V is a hardware isolation component to control memory physical accesses. PMP offers a targeted approach to regulating access to physical memory addresses without requiring memory translation, making it suitable for both APU and MCU platforms. PMP can be compared to the MPU (2nd stage), which also focuses on refining memory access permissions in a hypervisor context. In the context of the CROSSCON project, PMP is considered part of the isolation components of the 2nd stage, i.e., regulated by the hypervisor privileged level. Additionally, PMP's versatility extends to the 3rd stage of isolation components classification, aligning with the objectives of the TEE, i.e., streamlining access control while maintaining the integrity and confidentiality of distinct workloads through configurations at the firmware level. This dual-purpose application showcases the adaptability of PMP and its capability to address both the hypervisor and TEE requirements. This versatility

underscores PMP's suitability for diverse computing environments, making it an asset for the CROSSCON project's overarching objectives of enhanced security, integrity, and isolation.

5. **S-Mode Physical Memory Protection (sPMP)** - The sPMP unit is a PMP-based isolation component specifically meant for isolating user-level applications, i.e., the primary stage in CROSSCON isolation components classification. It is crucial to ensure that kernel components can restrict physical addresses that user-level components can access to support secure processing and isolate faults of user-level applications. Hence, the sPMP registers can grant permissions to the user-privileged level, which has none by default, and revoke permissions from the kernel-privileged level, which has full permissions by default. sPMP is always configurable by both firmware and kernel software.
6. **Arm TrustZone -** Arm TrustZone is a hardware-based security technology that creates isolated environments, dividing a system into secure and non-secure worlds. Its isolation ensures sensitive data protection and secure operations, safeguarding against unauthorized access and attacks.
7. **System MMU (SMMU) –** The SMMU, an IOMMU, enables the use of memory mapping, i.e., virtualization techniques, to ensure that data transfers between these devices and system memory occur securely. The key features are address translation and access control, preventing malicious devices from accessing unauthorized memory regions.
8. SW isolation aka PISTIS.

4.2 CROSSCON Trusted Services

In the sections that follow, we delve into the various services provided by CROSSCON. We begin by exploring context-based authentication, which leverages the device's environment to create unique fingerprints, thereby enhancing security, especially for second-factor authentication. Subsequently, we introduce Physical Unclonable Functions (PUFs) as a promising approach aligning with multi-factor authentication principles and offering 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 extend our exploration beyond the mentioned trusted services, shedding light on ongoing work in areas such as Machine Learning Model Protection and Behavioral Anomaly Detection on devices.

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 meta-data to create a fingerprint of the Wi-Fi landscape around a device. During the enrolment phase, a device captures this meta-data, 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 enrolment and authentication phases.

4.2.2 Exploring Physical Unclonable Function (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.

4.2.3 Secure FPGA Provisioning

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 provider must be protected from rogue workloads provisioned by malicious clients that can disrupt the execution of other clients running their workloads on the same FGPA accelerators or even physically damage the FPGA hardware of the service provider. To accommodate these requirements, the Secure FPGA provisioning service will provide two main sub-services: the FPGA RoT Setup service and the Secure FPGA Configuration/Bitstream Scanning service.

The FPGA RoT Setup service relies on two fundamental steps that are crucial in fostering an environment of trust and security. Firstly, the service emphasizes rigorous verification of the integrity and authenticity of the RoT's code or configuration. Advanced cryptographic techniques and hardware-backed security measures are employed to ensure that the RoT's code remains unaltered and free from tampering. Rigorous validation checks confirm that the RoT's code matches the expected configuration, providing a solid foundation for the entire FPGA configuration process and instilling confidence in users that their designs are processed by authentic firmware. Secondly, the service prioritizes the configuration of the RoT on the pertinent FPGA(s). At the end of these steps, the RoT runs in isolation on the FPGA, and its operations remain shielded from external interference and unauthorized access, offering robust protection for users' sensitive bitstreams. In this manner, the

FPGA RoT Setup service sets a high standard in FPGA security, delivering an unmatched level of assurance and trust for users' critical designs.

The Secure FPGA Configuration/Bitstream Scanning service is meticulously designed to ensure that hardware designs to be configured on the FPGA are free from malicious circuits while upholding the utmost privacy of the bitstream.

The primary objective of Secure FPGA Configuration/Bitstream Scanning is to guarantee the integrity and authenticity of the FPGA design before it is deployed on the hardware. FPGA designs can be complex and may involve multiple contributors, making them susceptible to malicious insertions. CROSSCON leverages advanced scanning and vetting techniques to diligently examine the FPGA bitstream and detect potential anomalies or malicious components that could compromise the system's security. During the scanning process, the service validates that the bitstream adheres to the intended design specifications, i.e., ensuring that it does not configure or overwrite neighboring FPGA resources that are not allocated for the current tenant and that no malicious circuits are included. Any discrepancies or unauthorized alterations are flagged and brought to the system administrator's attention for further investigation.

Once the scanning process is complete, a detailed report is provided to the system administrator, outlining the analysis results. The report includes information about the integrity of the FPGA design and any detected anomalies.

To ensure privacy and confidentiality, CROSSCON employs state-of-the-art cryptographic techniques to secure the bitstream. Bitstreams are encrypted using robust encryption algorithms, and the encryption keys are securely managed and never exposed, ensuring that only authorized parties can access the contents of the bitstream. The scanning is performed inside a trusted execution environment to protect the bitstream against unauthorized access.

4.2.4 Expanding Research Horizons: Beyond Listed Trusted Services

Our research goes beyond the listed trusted services. We're actively working on areas like Machine Learning Model Protection, Behavioral Anomaly Detection on devices, and Secure Code Updates. These aspects are currently in progress and will be included in future versions. Our commitment to evolving IoT security drives us to explore innovative avenues and ensure comprehensive protection for IoT devices and systems.

5 Specifications of Proposed Architecture

In this section, we delve into two critical aspects of the system. Firstly, we specify the interfaces, establishing how developers and runtime components interact with the CROSSCON Stack. Secondly, we delve into the various instantiation options, all of which are grounded in a generic architectural framework. The goal is to provide a comprehensive understanding of the system interfaces and, thus, elucidate how users can leverage its capabilities to meet their specific needs.

5.1 CROSSCON Hypervisor Interface

This section specifies the CROSSCON Hypervisor interface. The CROSSCON Hypervisor will be based on the Bao static partitioning hypervisor. Bao is a security and safety-oriented, lightweight bare-metal hypervisor prioritizes fault containment and real-time performance. Bao's implementation is streamlined, consisting of a minimal layer of privileged software that utilizes ISA virtualization support to create a static partitioning hypervisor architecture. In Bao, resources like CPU, memory, and I/O are allocated and partitioned when virtual machines (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 decision results in a smaller Trusted Computing Base (TCB), enhancing security. Additionally, Bao facilitates 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 detailed in a file, and developers use this file to specify how they want the hypervisor to behave.

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. Since the configuration file directly influences how the hypervisor configures the system, it serves as the primary way for developers to interact with the hypervisor. To ensure system security, developers need to have a good understanding of how to set up this configuration.

In runtime, 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.

5.2 Configuration File Structure

The configuration file is a C source file that is compiled by the hypervisor's build system. The configuration parameters are defined through a C struct whose fields are filled by the developer configuring the system. The configuration file defines the following structure:

```
#include <config.h>

struct config config {
        ...
}
```

In practice, this structure is compiled by the C compiler, and the resulting binary will typically contain the VM images and their configuration and hypervisor global configuration. The order of the configuration fields will not affect the correct operation of Bao.

5.2.1 Load Options for VM Images

It is necessary to inform Bao on how guest VM images are loaded. Two main options are available: i) include the images directly in the config file, or ii) have the images loaded into memory independently.

To include the VM images (e.g., .bin file) in the config binary itself, the C macro VM_IMAGE should be used. This macro takes two arguments: a name that will be used as a handle and a file system path to the VM image. Example usage:

```
VM_IMAGE(img1_name, "/path/to/vm1/binary.bin");
VM_IMAGE(img2_name, "/path/to/vm2/binary.bin");
```

5.2.2 Config Fields

5.2.2.1 Shared Memory

Shared memory is one or more memory regions that are shared between at least two VMs. The configuration of shared memory is done by populating the *shmem* struct:

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

Where:

[1] **size** - the size of the shared memory region. The size of the shared memory must always be multiples of 4KB (0x1000 in hexadecimal).

[2] **place_phys** - if set to True, it allows to specify the physical address of the memory region. The default value is False.

[3] **base** - base address of the shared memory. **Note:** Used only in MPU-based systems.

[4] **phys** - physical address of the shared memory. This field must be configured when place_phys is set to True. **Note:** Not used in MPU-based systems.

Shared memory regions are established by using an array:

```
struct config config = {
...
    size_t shmemlist_size;
    struct shmem *shmemlist;
...
}
```

Where:

[1] **shmemlist_size** - number of entries in the array of shared memory regions.

[2] **shmemlist** – an array of shared memory regions.

An ID is assigned to each shared memory, with the ID corresponding to the index in the shared memory array. This ID is used to associate a shared memory region to an IPC.

5.2.2.2 VM Configuration

To configure Bao to set up one or more VMs, *vm_config* entries should be added to the array of VMs (*vmlist*). The array of VMs must include at least one VM. The relevant config fields are the following:

```
struct config config = {
...
    size_t vmlist_size;
```

```
    struct vm_config vmlist[];
...
}
```

Where:

[1] **vmlist_size** - number of entries in the *vmlist* array.
[2] **vmlist** – an array of VMs.

It is possible to define various aspects of a VM's virtual environment, for example, memory, devices, and interrupts. This is done by setting the fields of the struct *vmconfig*:

```
struct vm_config {
    struct image;
    vaddr_t entry;
    cpumap_t cpu_affinity;
    colormap_t colors;
    struct vm_platform platform;
};
```

Where:

[1] **image** - configuration of guest image parameters. See below in Guest config.
[2] **entry** - entry point address in the VM's address space. This is the address that will be used by Bao to begin VM execution.
[3] **cpu_affinity** - bitmap that sets the preferred physical CPUs assigned to the VM. If this value of this field is mutually exclusive between VMs, the physical CPUs assigned to each VM follow the bitmap. Otherwise (in case of bit overlap or lack of affinity definition), the CPU assignment is defined by the hypervisor. By default, this field is set to 0x0, meaning no physical CPU is preferred by the VM.
[4] **colors** - bitmap selecting cache colors for the L2 shared cache. See below in Coloring. By default, this field is set to 0x0, meaning all cache colors will be used.
[5] **platform** - set various aspects of the virtual platform that is presented to the VM; see below in VM Platform Description.

5.2.2.3 Guest Config

The VM image loading and installation information is set through the following struct:

```
struct image {
    vaddr_t base_addr;
    paddr_t load_addr;
    size_t size;
    bool separately_loaded;
    bool in place;
};
```

Where:

[1] **base_addr** - image's address in VM's address space.
[2] **load_addr** - image's address in the hypervisor address space. This field should be defined using the macro VM_IMAGE_OFFSET(img) if the image the VM_IMAGE macro was previously used. Otherwise, it should be set to the address the bootloader used to load the VM images.
[3] **size** - the size of the image. This field should be defined using the macro VM_IMAGE_SIZE(img) if the image the VM_IMAGE macro was previously used. Otherwise, it should be set to the size of the VM images.
[4] **separately_loaded** - informs the hypervisor that the VM image was loaded separately by a bootloader. This field is set to False by default.
[5] **in place** - if True Bao will use the image in place and won't copy it to another memory location. This field is set as False by default.

5.2.2.3.1 Coloring

In the context of Bao, cache coloring, also known as cache partitioning, is a technique used to partition the cache space among different VMs. This can ensure that each VM occupies distinct sections of the cache, reducing interference for time-critical guests.

The *colors* field value is truncated depending on the number of available colors calculated at runtime; thus, its effect is platform-dependent. By default, the cache coloring mechanism is not active.

5.2.2.3.2 VM Platform Description

The struct *vm_platform* describes the virtual platform available to the VM:

```
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;
};
```

Following is a description of each of these fields.

Number of vCPUs:

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

Memory Mapping:

[1] **region_num** [mandatory] - number of entries in the array of memory regions mapped to the VM.

[2] **regions** [mandatory] - an array of memory regions. Each memory region contemplates the following parameters:

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

Where:

[1] **base** - base virtual address of the memory region.

[2] **size** - the size of the memory region.

[3] **colors** - bitmap to enable cache coloring using specific cache colors. By default, no colors are used.

[4] **place_phys** – flag to indicate that a physical address to where the memory region will be mapped will be used. By default, this value is set to False.

[5] **phys** - physical address where the memory region should be mapped. Only meaningful if *place_phys* is set to True.

Inter-Partition Communication (IPC):

[1] **ipc_num** - number of IPCs assigned to the VM. By default, this field is set to 0.

[2] **ipcs** - an array of IPC. Each IPC contemplates the following parameters:

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

Where:

[1] **base** - virtual base address of the IPC memory region.
[2] **size** - the size of the IPC memory region. It should be smaller or equal to the shared memory region corresponding to the set shmem_id.
[3] **shmem_id** - ID of the shared memory associated with the IPC.
[4] **interrupt_num** - number of interrupts assigned to the IPC.
[5] **interrupts** - an array of interrupts associated with this IPC channel. Each interrupt is identified by an integer.

Devices:

VM devices are effectively a passthrough to let VMs directly access the platform's MMIO (memory-mapped IO).

[1] **dev_num** - number of devices, i.e., MMIO, regions.
[2] **devs** - an array of device regions. Each device region contemplates the following parameters:

```
struct vm_dev_region {
    paddr_t pa;
    vaddr_t va;
    size_t size;
    size_t interrupt_num;
    irqid_t *interrupts;
    streamid_t id;
};
```

Where:

[1] **pa** - physical address of the device region.
[2] **va** - virtual address of the device region.
[3] **size** - the size of the device region in bytes.
[4] **interrupt_num** - number of interrupts generated by the devices encompassed by the device region. By default, this value is 0x0.
[5] **interrupts** – an array of interrupt IDs generated by the device.
[6] **id** - id used by the system to match IOMMU / IOMPU filtering rules according to the VM's address space. By default, this value is set to 0x0. This value effect and meaning varies across platforms.

Memory Management:

[1] **mmu** - Configures Bao to use virtual memory using page tables. In MPU-based platforms (i.e., aarch64 Cortex-R), the hypervisor sets up the VM using an MPU by default. This field is only relevant in the aforementioned platforms. The default value of this field is False.

Architectural configurations:

[1] **arch** - definition of architecture-dependent configurations. Each supported architecture may require specific support based on the available architecture features. As such, each architecture defines this struct to include relevant configuration parameters.

ArmV8:

Armv8 is a 64-bit instruction set architecture (ISA) developed by Arm. This architecture is widely utilized in diverse computing devices due to its improved performance, energy efficiency, memory management, and security features. Armv8-specific features are configured using the following struct:

```
struct arch_vm_platform {
    struct vgic_dscrp gic;
    struct {
        ...
    } smmu;
};
```

Where:

[1] **vgic_dscrp** - configuration of the virtual Generic Interrupt Controller (GIC).
[2] **smmu** - configuration of the SMMU.

GIC:

To allow emulation of the interrupt controller, Bao requires that the GIC MMIO region be set up in a different way from the passthrough devices. Currently, support exists for Arm platforms that feature GICv2 [23] and GICv3 [24] versions of Arm's Generic Interrupt Controller (GIC).

```
struct vgic_dscrp {
    paddr_t gicd_addr;
    paddr_t gicc_addr;
    paddr_t gicr_addr;
    size_t interrupt_num;
};
```

Where:

[1] **gicd_addr** – address for the GIC distributor, applicable for both GICv2 and GICv3.
[2] **gicc_addr** – address for the GIC controller, applicable for both GICv2 and GICv3.
[3] **gicr_addr** – address for the GIC redistributor, applicable only for GICv3.
[4] **interrupts_num** – the number of interrupts the virtual GIC will present to the VM.

SMMU:

SMMU stands for System Memory Management Unit [25]. It is a hardware component commonly found in Arm-based systems that manages memory access and translation between different processing units, such as CPUs and peripherals. For Arm platforms, Bao supports the SMMUv2 [26]. Armv8-specific features are configured using the following struct:

```
struct {
    streamid_t global_mask;
    size_t group_num;
    struct smmu_group *groups;
} smmu;
```

Where:

[1] **global_mask** – a global mask that will match with stream IDs. See Arm's SMMUv2 specification for more details.
[2] **group_num** – number of SMMU groups.
[3] **groups** – an array of SMMU groups. Each SMMU group is defined by:

```
struct smmu_group {
    streamid_t mask;
    streamid_t id;
}* groups;
```

RISC-V:

RISC-V is an instruction set architecture (ISA) that stands out for its open nature, offering flexibility and customization opportunities. Developed with simplicity and modularity in mind, RISC-V has gained

traction across a variety of computing devices due to its potential for optimized performance, energy efficiency, memory management, and security features. Its open design encourages innovation, making it an attractive choice for a diverse range of applications and industries.

RISC-V-specific features are configured using the following struct:

```
struct arch_vm_platform {
    paddr_t plic_base;
};
```

Where:

1 **plic_base** – the base address of the virtual PLIC. PLIC is the RISC-V standard interrupt controller [27].

5.2.3 Example

The following example delineates a config source file that creates two VMs for an Armv8 platform.

```
#include <config.h>

/**
 * Declare VM images using the VM_IMAGE macro, passing an identifier and the
 * path for the image.
 */
VM_IMAGE(vm1, "/path/to/vm1/binary.bin");
VM_IMAGE(vm2, "/path/to/vm2/binary.bin");

/**
 * 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 = {
        {
            .image = {
                .base_addr = 0x80000000,
                /**
                 * Use the  VM_IMAGE_OFFSET and VM_IMAGE_SIZE to initialize
                 * the image fields passing the identifier of the image.
                 */
                .load_addr = VM_IMAGE_OFFSET(vm1),
                .size = VM_IMAGE_SIZE(vm1)
            },

            .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
                    }
                }
            },
        },

        {
            .image = {
                .base_addr = 0x00000000,
                .load_addr = VM_IMAGE_OFFSET(vm2),
                .size = VM_IMAGE_SIZE(vm2)
            },

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

VM1 image is a file found in the filesystem directory "/path/to/vm1/binary.bin" and the VM2 image file is "/path/to/vm2/binary.bin." The images are included in the binary by using the VM_IMAGE macro. Next, the struct config is defined. One shared memory region is configured for the system, which will be used to allow communication between the two VMs. The shared memory region size is 0x1000, i.e., 4096 bytes. Following are the VM descriptions for the two VMs.

VM1 image base address in VM1's virtual address space is 0x80000000. The macros VM_IMAGE_OFFSET and VM_IMAGE_SIZE are used to easily obtain the load address and size of VM1's image. VM1 will begin execution at address 0x80000000. Physical CPUs 0 and 1 are preferred by VM1. VM1 will use half of the available cache colors after the VM1 platform is defined. The platform will feature two vCPUS (corresponding to two physical CPUs, as there is 1-to-1 mapping). There will be a single memory region starting from 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 macros VM_IMAGE_OFFSET and VM_IMAGE_SIZE are used to easily obtain the load address and size of VM2's image. VM2 will begin execution at address 0x00000000. Physical CPUs 2 and 3 are preferred by 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 access the GIC controller and GIC distributor (GICv2 interrupt controller) at 0x2C000000 and 0x2F000000, respectively.

5.2.4 Runtime Interface

The hypervisor runtime interface comprises the collection of interfaces that a VM uses to interact with the hypervisor during the VM's execution. There are two types of interfaces: trap and emulation and hypervisor calls.

5.2.4.1 Trap and Emulation

A trap occurs when a virtualized guest operating system executes a privileged instruction or accesses memory regions unmapped to the VM. 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 a mode operation that provides firmware-level functionality. 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) [28] in the case of Arm and the Supervisor Binary Interface (SBI) accessed through an "ECALL" in RISC-V [29].

1. Arm SMC emulation - Arm SMC (Secure Monitor Call) [28] 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 virtual machines 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) [30].
2. 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 [29].

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 operating system or the hypervisor, to manage the system's behavior and maintain security and isolation between different software components or virtual machines. Some system registers are accessible only in certain privilege levels, ensuring that critical settings are not easily altered by user-level applications or VMs. Accesses to system registers need to be decoded for the type of operation (read or write) and must be handled accordingly.

1. Arm System Register emulation - In Arm, the CROSSCON Hypervisor controls access to GICv3 system registers ICC_SGIR and ICC_SRE. For more information, see [10] [24].
2. 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 would be MMIO regions to implement virtualization functionality (e.g., sharing a device between guests). Similarly, to system register emulation, memory accesses need to be decoded for the type of operation (read or write).

1. Arm MMIO emulation - The hypervisor emulates the interrupt controller in Arm platforms. The emulation of Arm's Generic Interrupt Controller (GIC) [31] enables the hypervisor to manage and process interrupts 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.
2. 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) [27] enables the hypervisor to manage and process interrupts driving the behavior of the physical PLIC without VMs being able to interfere with each other's interrupts. Currently, there is support for the PLIC interrupt controller.

5.2.4.2 Hypercalls

The hypervisor call interface, henceforth the hypercall interface, requires VMs to be aware of the hypervisor. A VM will issue hypercalls using dedicated instructions in a similar way to system calls issued by an application to the OS. This allows VMs to directly request services to the hypervisor.

Hypercalls provide a controlled pathway for VMs to access hypervisor functionalities, ensuring efficient execution. In the CROSSCON Hypervisor, the available hypercall interface provides access to IPC communication, i.e., communication between VMs. The ABI for issuing hypercalls in Arm is defined in SMCCC [28], while in the RISC-V for issuing hypercalls is defined in Supervisor Binary Interface Specification [29].

Inter-Partition Communication - Inter-partition communication enables information exchange among distinct partitions or VMs. This communication mechanism allows VMs to interact and share data while maintaining strong isolation boundaries to ensure security. By establishing secure channels for IPC, virtualized environments can achieve efficient resource utilization and versatile collaboration scenarios. The IPC interface contemplates two arguments: the ID of the shared memory, defined in the configuration file, and the IPC event ID, which corresponds to the interrupt ID to inject into the IPC channel participants.

Table 2: Hypercall interface for IPC between VMs

Parameter	Description
SHMEM_ID	The ID of the shared memory
IPC_EVENT	Interrupt to inject the participants of this IPC channel

5.3 Instantiation Options for CROSSCON

In the following, we discuss the deployment of the generic CROSSCON architecture on the different platforms we are considering so far, from the most powerful and rich platforms to the limited ones, where, for instance, the CROSSCON isolation is achieved through software. We first discuss the APU class processors, APU - RISC-V (M/H/S/U), then the APU - ArmV7/V8-A (<v8.4), the MCU class processors, MCU - RISC-V (M/S/U), the MCU - Armv6/v7-M / AVR / MSP, the MCU - Arm-V8-M, and finally the MCU - ArmV8-R. However, before going into the instantiation options for CROSSCON Stack's details, we discuss the objectives, assumptions, and constraints our work builds on.

Objectives:

- CROSSCON stack aims to be as heterogeneous as possible, considering covering multiple architectures across multiple ISAs.

Assumptions:

- CROSSCON will assume that in Arm, CROSSCON uses Arm Trustzone for both APU and MCU whereas RISC-V utilizes PMP;
- In APU RISC-V instantiation models, CROSSCON assumes the use of the PMP for TEE provisioning;
- In APU Armv7-A instantiation models, CROSSCON assumes CROSSCON Hypervisor will only manage the REE side. Leaving Trusted OS as the most privileged components in TEE side;
- CROSSCON Stack assumes that RISC-V MCU Hypervisors will be executed alongside the firmware privileged level.
- In constrained architectures like AVR, MSP, Armv6-M, and Armv7-M with just one privileged level, CROSSCON assumes the use of SW techniques to protect applications, when no MPU isolation components are present.

Constraints:

- On bare metal systems, CROSSCON offers a limited set of features compatible with the underlying architecture.

5.3.1 APU - RISC-V (M/H/S/U)

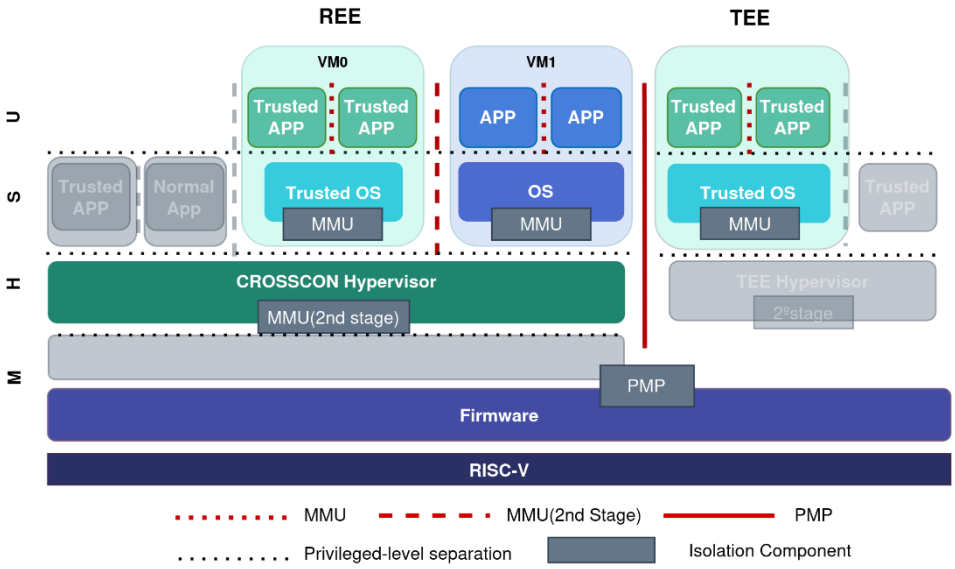

Figure 9. APU - RISC-V (M/H/S/U) architecture

The RISC-V APU architecture offers both a REE and a TEE via the use of the PMP module that ensures physical isolation between the two worlds. This module is managed by the firmware executing in machine (M) mode. On this architecture, the CROSSCON Hypervisor is executed solely in the Hypervisor (H) level, which is in charge of handling the 2-stage MMU. This allows the hypervisor to

isolate different complex VMs, each one running OSes (either trusted or untrusted) or some applications (either trusted or untrusted). The VM OS will also leverage the MMU, although the 1st stage only, to manage the resources of the various applications. Notably, the CROSSCON Hypervisor will handle either the communication between VMs or between worlds (REE and TEE).

5.3.2 APU - ArmV7/V8-A (<v8.4)

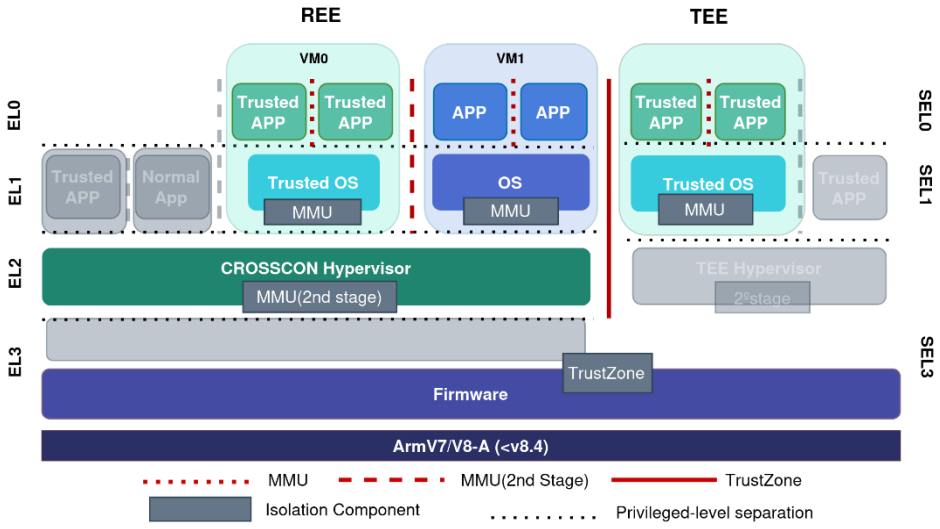

Figure 10. APU - ArmV7/V8-A (<v8.4) architecture

On Armv7-A and v8-A, the CROSSCON architecture is quite similar to the RISC-V APU. Comparing with them, (i) instead of using the PMP-like isolation mechanisms to establish a TEE, it leverages TrustZone technology, (ii) introducing secure privileged levels (Secure Exception Levels 0 and 1, for trusted OS and its trusted applications respectively). Trusted OS running in SEL1 will be in charge of handling the resources of the entire TEE without management of the CROSCCON Hypervisor. In this instantiation option we assume CROSSCON Hypervisor will only manage the REE with the ability of creating/managing isolated VMs (trusted or untrusted VMs), as well as all the communication with the rest of the system. Notably, the communication between worlds are handled by the firmware but relies on a standard defined between the Trusted OS and the hypervisor. Although the former can access and manage the hypervisor's resources, we assume that it is trusted.

5.3.3 MCU - RISC-V (M/S/U)

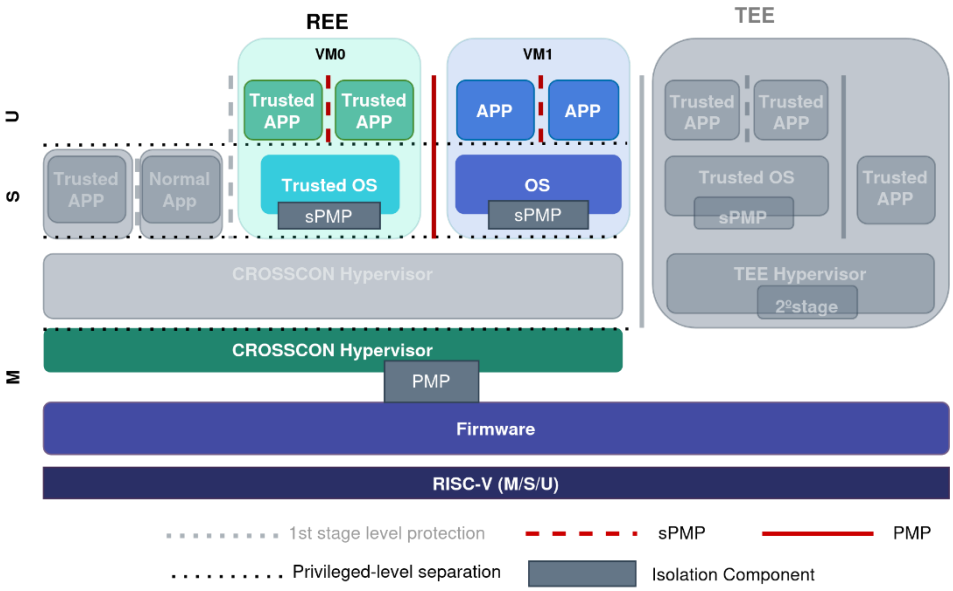

Figure 11. MCU - RISC-V (M/S/U) architecture

On the RISC-V MCUs there is no MMU, reason for which the hypervisor cannot simply handle the various VMs resources. However, these MCUs have both the PMP and the supervisor PMP (sPMP), which can be used as an alternative to the MMU to establish a 2nd stage and 1st stage isolation, respectively. In particular, the CROSSCON Hypervisor is executed alongside the firmware in Machine mode, thus having complete access to the resources of the system. It uses the PMP to assign different resources to complex VMs executing in S and U level. Each VM is equipped with an OS, either trusted or non-trusted, that manages the applications within the VM. To isolate the applications, rather than using the 1st stage MMU, the OS utilises the sPMP which, similarly to the PMP, can be used to restrict access to the system resources by the S privilege level. Notably, using the PMP as a mean to isolate the VM precludes the establishment of a TEE.

5.3.4 MCU - Arm-V8-M

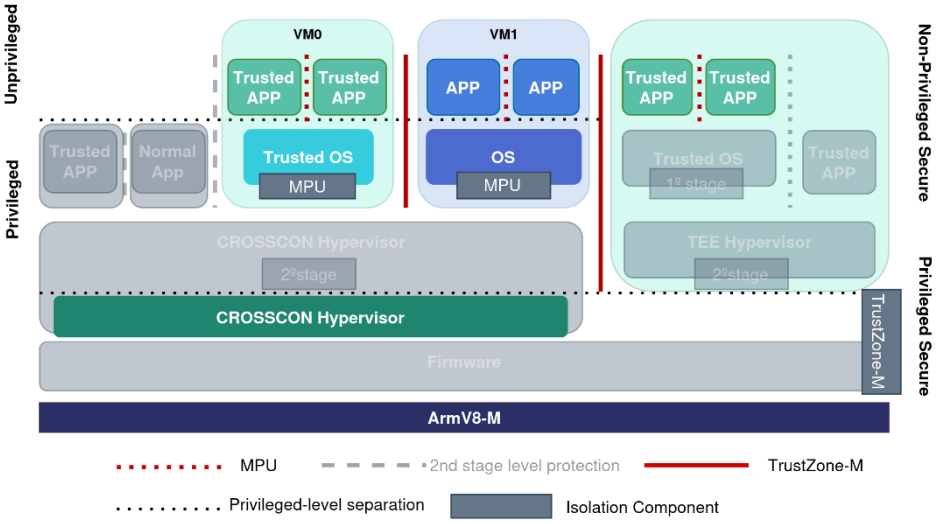

Figure 12. MCU - Arm-V8-M architecture

Contrarily to the Armv8-A, the Cortex-M family of devices relies on an architecture that lacks an MMU and only provides two privilege levels: privileged and unprivileged. However, this architecture is equipped both with an MPU and with TrustZone-M. In this constrained architecture, the CROSSCON Hypervisor is executed at the secure privileged level within TrustZone-M TEE, while the rest of the

software is executed at the unprivileged or secure unprivileged level. To enforce the isolation between the different VMs, the hypervisor leverages TrustZone-M technology, allowing the REE control over the MPU configuration to enforce the isolation between the applications within each VM. Notably, trusted applications can also be deployed on the non-privileged secure environment inside the TEE.

5.3.5 MCU - Armv6/v7-M / AVR / MSP

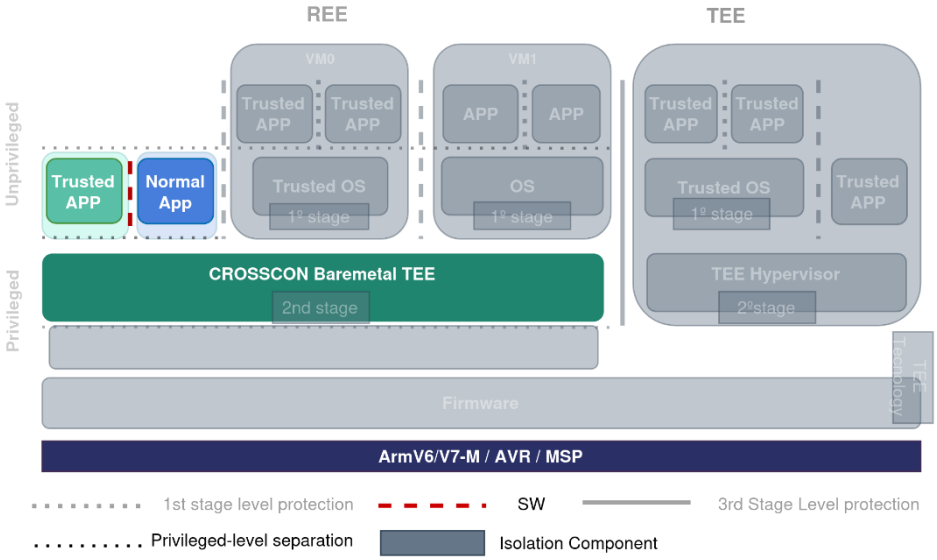

Figure 13. MCU - Armv6/v7-M / AVR / MSP architecture

The AVR, MSP and Armv6-M/v7-M architectures are particularly constrained by the lack of security features. On Armv6-M/v7-M we have only 2 execution privileges, while on MSP and AVR we have only one. This makes the creation of security features challenging. In these scenarios, we do not employ a proper hypervisor, but establish a bare metal TEE instead. This is an environment that supervises the execution of normal and trusted applications, isolating the two using mainly software techniques. These techniques, such as software instrumentation, supply for the lack of proper hardware and allow the TEE to establish a bare set of security primitives (e.g., memory isolation). Whenever the privileged level is available, the TEE is executed there and possibly it leverages a MPU to enforce isolation. If such hardware primitives are not provided, the TEE is executed alongside the rest of the software and isolated exclusively through software instrumentation.

5.3.6 MCU - ArmV8-R

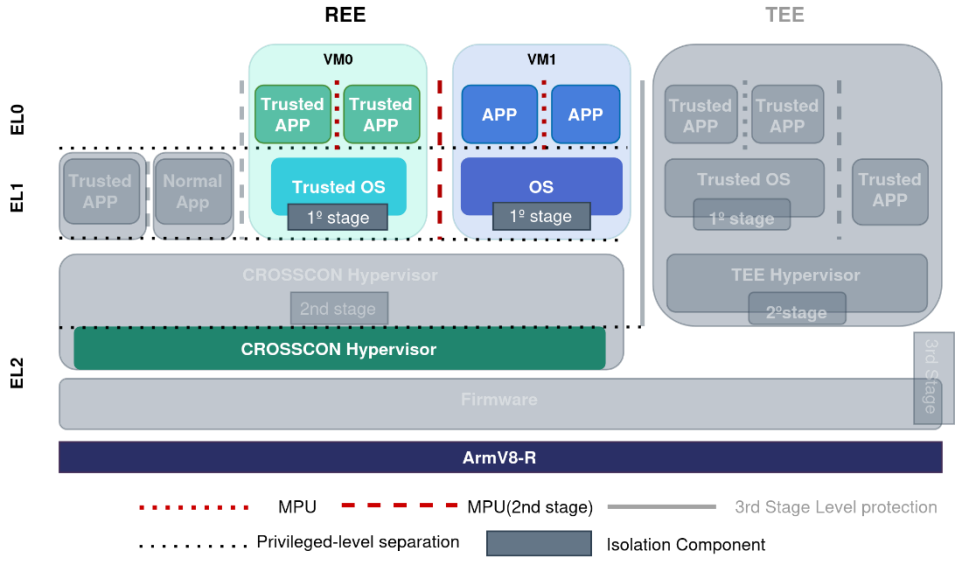

Figure 14. MCU - ArmV8-R architecture

The Armv8-R architecture offers a moderate number of security features, among which are a 2nd stage MMU and three privilege levels. In this context, the hypervisor is executed at the bottom level EL2 and uses the 2nd stage MMU to isolate the various VMs deployed in EL1 and EL0. Each VM had trusted or normal software comprising an OS and some applications. The OS can leverage the 1st stage MMU to isolate the single applications.

6 Hardware/Software Co-Design

Most modern-day IoT platforms and hardware (HW) architectures provide some HW primitives (MMU/MPU unit, random number generators, cryptographic accelerators, etc.) that can be used together with software to design IoT devices with security guarantees. Which security guarantees can be provided, what kind of impact the enforcement of those guarantees have on the performance of the system, and which use cases can be supported often depends on primitives provided by the HW. Therefore, having the right platform with the right HW architecture is important if one wants their IoT device to provide the strong security guarantees needed for a given use case and, at the same time, not sacrifice the device's performance.

In this section, we describe the CROSSCON system on chip (SoC) and Perimeter guard (PG) module that can be used to design a platform that, together with the CROSSCON stack, can provide strong security guarantees while not sacrificing performance.

6.1 High-level overview of CROSSCON SoC

The CROSSCON SoC is a design for a system on chip (SoC) with a RISC-V core that provides the necessary HW primitives that can be used to provide strong isolation of software and hardware that belongs to different security domains. Figure 15 shows the current high-level overview of the CROSSCON SoC that will be refined during the project.

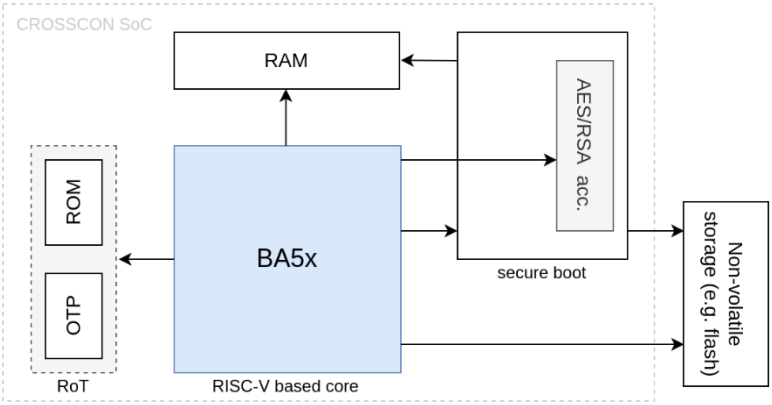

Figure 15: A high-level design of CROSSCON SoC

As shown by Figure 15, the main processing unit of the CROSSCON SoC is BA5x based core that has access to RAM, non-volatile storage (e.g., flash), secure boot sub-system, AES or/and RSA accelerator, and root of trust (RoT) that includes read-only memory (ROM) and one-time programmable memory (OTP). The BA5x is a line of RISC-V cores provided by Beyond Semiconductor. We currently intend to use the BA51 version of the core that supports RISV-V 32-bit ISA with machine, hypervisor, supervisor and user modes, atomic operations, 2-level MPU (PMP and sPMP), and some other extensions without an MMU unit.

The CROSSCON SoC can be used together with CROSSCON Hypervisor to provide isolation of the execution environment and other system resources (cryptographic accelerators, peripheral devices, etc.), where the CROSSCON Hypervisor is instantiated in a similar fashion as in MCU - RISC-V (M/S/U) setup, as described in the Sect. 5.3.3. The instantiation does not rely on a separate TEE and does not provide full memory virtualization, as an MMU unit is not available. By extending the (basic) CROSSCON SoC design with additional HW extensions (e.g., Perimeter guard), it can be used to build IoT devices that belong to class 1, 2, or 3 of the device classification introduced in section 2 of the Requirements Elicitation Initial Technical Specification document [2].

The system can be booted using the secure boot sub-system that loads and authenticates an encrypted program stored in non-volatile storage (e.g., flash memory). When the system is booted, BA5x runs

the first-stage code of the boot process stored in the ROM. The first-stage code configures the secure boot sub-system and provides it with the necessary secrets that are stored in the one-time programmable memory (OTP) and instructs the sub-system to decrypt, load, and authenticate the program stored in the non-volatile storage. BA5x then runs the program if the decryption and authentication are successful. The ROM and the OTP are a part of the RoT.

After the boot process is complete, the secure boot sub-system exposes the AES and/or RSA cryptographic accelerators to the program running on BA5x.

During the CROSSCON project, the CROSSCON SoC will be used as a base SoC for the development of further hardware extensions, for example, the perimeter guard module that we describe in the following chapter, which will provide additional HW primitives that can be used by the programs running on the SoC.

Note that the current high-level design of the CROSSCON SoC described in this chapter is expected to change during the project. The changes might include adding or removing parts of the SoC's functionality (e.g., secure boot sub-system) and using different versions of the BA5x core.

6.2 Perimeter Guard

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

6.2.1 Motivation

A separation kernel is a hardware and / or software mechanism that divides the system resources into several domains, often also called partitions [15] or worlds [32] [33], that are isolated between each other except for the information flows that are allowed by the separation kernel. A domain is composed of several resources (processing cores, memory, accelerators, peripheral devices, etc.) that are used to process and store data. A program can run in an execution environment inside of a domain where it has access to the resources of that domain and to the resources of other domains if allowed by the kernel. Which guarantees are provided by the isolation depend on the specific separation kernel, but the usual guarantees that are provided include confidentiality and integrity of the code and data inside of a domain for the assumed threat model. Examples of such separation kernels are Sel4 and Bao hypervisor.

Note that the difference between a separation kernel and a TEE is that the main goal of the separation kernel is to isolate the system resources. At the same time, TEE also provides other functionality/guarantees, for example, authenticity of the code and the ability to remotely attest to the execution environment [34].

The isolation between domains is achieved by precisely controlling which information is shared between domains, including the information sent to and received from different (on and off SoC) HW modules (cryptographic accelerators, graphic cards, peripheral devices, etc.). Because no guarantees are often provided about how the information flows inside a HW module, the module should be restricted to only one domain. However, this is often not the case because several domains require the functionality provided by the module. This leads to information leaking from one domain to another, resulting in domains being no longer isolated.

To share a HW module across domains and preserve the isolation, we need to ensure that there are no information flows going through the module that carry information from one domain to another that is not allowed. We can find several approaches to how information flows can be tracked and enforced in hardware across the literature [35] but most of the approaches are mainly concerned with how information flows can be enforced in central processing units, are generic, and require a lot of effort to apply to the existing HW modules, or they only cover modules with specific design features

(e.g., processing pipelines) [36]. Because of that, we see a need for a more general solution that can be easily applied to a wide range of HW modules.

In the rest of the chapter, we describe a Perimeter guard (PG) - 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 configuration. We describe several different behaviors of the perimeter guard that allow the module to be shared between domains while providing (explicit) information flow guarantees.

Although we think the PG can be applied to a broad range of HW modules, we mainly focus on how it can be applied to hardware accelerators, such as cryptographic accelerators.

6.2.2 Threat Model

Following is a threat model specific to the problem we are trying to solve with PG.

Setup: We assume that we already have a system with several domains that are separated in some way, for example, by a separation kernel, and that the separation mechanism guarantees non-interference [37] between two domains if they are not allowed to interact according to the policy.

Note a *policy* is a piece of information that describes how a system should behave; for example, it describes if information can be exchanged between two domains. A policy can be provided to a part of the system through its configuration.

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

Note we use the term *actor* to refer to any entity that can act in a domain, for example, programs, DMAs, peripheral devices, etc., which also includes HW modules connected to various interconnects that act as masters.

We assume the following threat model: The attacker has full control of one or several domains, excluding the privileged domain that can configure the PGs and contains the PG's policy. The attacker can execute arbitrary instructions, start new programs, influence existing programs, modify OS, control peripheral devices and HW modules, etc., in the domains it controls.

We **trust** the separation mechanism (e.g., separation kernel), the privileged domain configuring the PG, and the PG itself. We assume all the components we trust are functionally correct, have no bugs, and behave as expected.

In the following sections, we argue that the perimeter guard used together with the newly added module can be used to preserve non-interference and, at the same time, allow the module to be shared between domains while assuming the described threat mode, trust model, and the following assumptions:

- Physical attacks, such as fault injection and bus sniffing, are out of scope.

We consider physical attacks (e.g., fault injection and bus sniffing) as an orthogonal problem that can be mitigated by providing separate solutions. When a reasonable solution is available, we provide specific solutions for DoS and timing side-channel attacks.

We define *non-interference* as A domain X cannot interfere with a domain Y if no action performed by an actor in a domain X can influence the state of the domain Y.

From the definition, it follows that if we have a non-interference between two domains, then if we remove one domain, the other one should have the same behavior. The definition is aligned with the definition provided by Rushby [37].

From the guarantee of non-interference between domains, it follows that:

- code and data cannot be exfiltrated from a domain,
- code and data cannot be infiltrated into a domain, and
- code and data cannot be mediated between two domains.

We define exfiltration, infiltration, and mediation as:

- an actor A in domain X is not able to exfiltrate data from domain Y if it is not able to obtain any information about the state of Y if that is not allowed by the policy,
- an actor P in domain X is not able to infiltrate data into domain Y if it is not able to modify the state or Y if that is not allowed by the policy and
- an actor P in domain X cannot mediate information between domains Y and Z if it cannot instrument that information about the state of domain Y is obtained by domain Z if that is not allowed by the policy.

The definitions are in line with the definitions of exfiltration, infiltration, and mediation described in [15]. By showing that actors inside a domain cannot exfiltrate, infiltrate, or mediate information from, to, and between domains, we also show that the integrity and confidentiality of data in other domains are preserved. If an actor cannot exfiltrate or mediate the data, data in different domains will stay confidential. If a program cannot infiltrate or mediate the data, the integrity of the data in other domains will be preserved.

We define:

- integrity is the property that the data cannot be modified by any actor from other domains (directly or indirectly) if that is not allowed according to the policy and
- confidentiality is the property that the data cannot be read by any actor from other domains if that is not allowed according to the policy.

6.2.3 Perimeter Guard Architecture

A *perimeter guard* (PG) is a SoC 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.

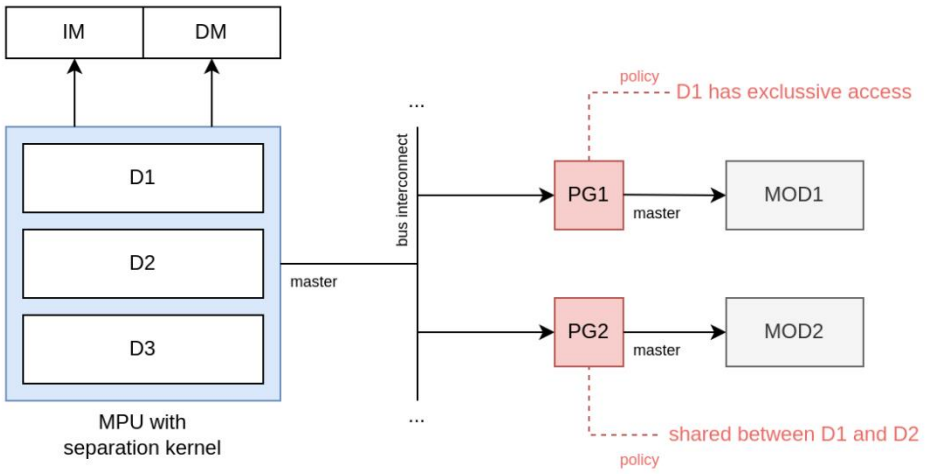

Figure 16: A SoC setup where a PG is placed in front of module 1 (MOD1) and module 2 (MOD2) to control the access to those modules

For example, consider the setup in Figure 16, where an MPU with three different domains - D1, D2, and D3 - is connected to module 1 (MOD1) through perimeter guard 1 (PG1) and to module 2 (MOD2) through perimeter guard 2 (PG2), where PG1 only allows D1 to access MOD1 and PG2 allows D1 and D2 to access MOD2. If a program from D1 sends a load request to MOD1, PG1 will allow the request to pass through. If a program from D2 tries to send a load request to MOD1, PG1 will reject the request because the policy only allows D1 to access the device. A program from D2 can send a request to MOD2 because the policy of PG2 allows MOD2 to be shared between D1 and D2. D3 does not have access to any device.

Note that, to simplify the discussion, we assume that we already have a microprocessor (MPU) with a separation kernel that divides the resources inside the MPU into different domains. We can look at the PG as a way to extend the separation kernel to include modules/devices external to the separation kernel. By doing this, we do not lose any generality of our approach because additional details

regarding the architecture of the MPU and how the separation kernel is implemented can always be filled in.

The PG controls access to the module by restricting which store and load requests can pass through to the module. We place the 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 the PG to have full control of who can access the module, the module should only be connected to the bus interconnect through the PG. No other direct connections to the bus interconnect are allowed.

A precise mechanism of how a PG rejects a request still needs to be determined. A viable option would be that the PG raises an error flag on the bus.

In order for the PG to decide which store and load requests can pass through, the master sending the request needs to provide the necessary information for the PG to make a decision. In the case of a setup in Figure 16, an MPU must provide a domain's ID with the request. In other cases, additional information could be considered in the decision. For example, PG could behave as an MPU that would restrict access to a specific address based on the domain's ID from where the request is coming from. The information used 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 when to allow access, only that the master can provide that information. How this information is provided to the PG still needs to be determined. Because the information needs to be provided upon each request, the bus communication protocol will likely need to be adjusted. We plan to explore how this can be done in our future work.

Note that the MPU is not the only master that can try to access a module. When we have a memory module, a direct memory access (DMA) module could also try to access the memory. Therefore, the DMA should also be able to provide the necessary information to the PG to make a decision, or the PG and its policy should treat the requests from a DMA differently.

The PG needs to be configured at the boot stage of the MPU by providing the access policy to the PG by the appropriate authority. A precise mechanism of how the PG can be configured still needs to be determined. For example, we could require that PG can only be configured by the programs running in the D1 (or some other privileged mode) by writing the access policy into a specific address range that is memory-mapped to the perimeter guard's internal configuration storage. This would allow only programs in D1 to configure the PGs and not programs running in other domains.

We consider several different modes of operation of a PG and analyze them according to the threat model. We identified the following ways of how information can be passed from one domain to another:

- through the (internal) state of the module, and
- through module availability (i.e., when and how long the device is used).

A malicious program could try to use the module's (internal) state to obtain information about the state or modify the state of another domain. For example, a malicious program could try to learn something about another domain from the state of a module after another domain uses the module. Alternatively, a malicious program could modify the module's state to influence the domain that will use the module after the modification. This general class of attacks can be avoided by precisely controlling the entire state of the module. For example, we could reset the module's state to a predefined state where no information about the previous domain can be obtained. We can reset the module every time a domain stops using the module, and another domain gets access to it. The reset guarantees that no information about the previous domain can be obtained from the internal state of the module from the current domain that has access to the module.

The reset approach can be generalized by allowing the PG to control the entire state of the module at specific points in the module's execution instead of just being able to reset the module's state. This requires the module to provide a mechanism that allows it to load and store the entire state of the module. Although not all the modules might be able to support such a mechanism, it can, if properly used, prevent other domains from obtaining any information about the previous domain that used the

module from the module's state and, at the same time, having other benefits like preserving the module's state between domain switching and better utilization of the module.

Another class of attacks is timing attacks, where the malicious program could use the information of when and for how long other domains use the module to learn something new about them. For example, if a program knows how long a module was used, it could learn which operation was performed by the module; or, if a program monitors the access patterns to a module, it also could learn something about the domain accessing that module. This class of attacks could be avoided by dividing the time that a module can be used into fixed slices where each domain could access the module on a specific period for a fixed amount of time. Although this approach avoids timing attacks, it can lead to underutilization of the module because a domain might not need or cannot use the module for an entire time slot.

We consider the following ways the PG could behave and call them *modes of operation*:

- exclusive access,
- time-sharing with reset, and
- time-sharing with context switching.

6.2.3.1 PG Operation Mode: Exclusive Access

The *exclusive access* mode is one of the most straightforward behaviors that PG can provide and can be used to provide exclusive access to a module according to a policy and information provided in each request. For example, as considered in a setup of Figure 16, only domain D1 can access MOD1, where access to the module is granted according to the domain's ID.

This mode of operation can be used to restrict access to a module only to specific domains for the run-time of the SoC. Suppose the access to a module is restricted to only one domain. In that case, the attacks related to the module's internal state and availability are not a problem because the module is not shared between domains. However, if the policy allows a module to be accessed from several domains, the programs running in those domains have unrestricted access to that module. Thus, the domains are no longer separated. Access to each module should only be restricted to a single domain to preserve the isolation between domains.

Note that a module could be reassigned to a different domain if the appropriate authority reconfigures the PG. This can only be done when we are sure that no information will be leaked through the internal state of the module and through the time when the module was reassigned.

6.2.3.2 PG Operation Mode: Time-sharing with Reset

In the time-sharing with reset mode of operation, the PG restricts the access to a module to one domain at a time and allows the module to be accessed from another domain only after the module is reset to a known initial state. In order for a PG to be used in this mode of operation, the module needs to support a reset mechanism where the reset 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 behavior of the PG in the shared access with reset mode is described in the following diagram.

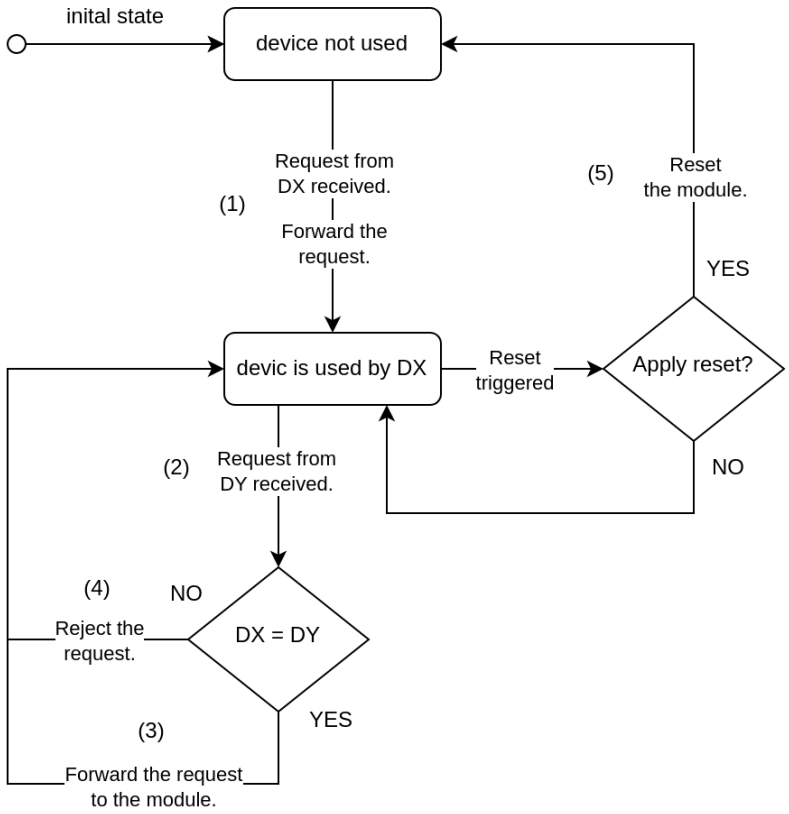

Figure 17: A simplified behavior diagram of a PG in a shared access with reset mode

At the beginning, the module is not used by any domain. When the first request is received (1) from a domain DX, the PG notes that DX currently uses the module and forwards the request to the module. When a new request (2) is received from the domain DY, the PG only forwards (3) the request to the module if the request was received from the same domain, i.e., DX = DY; otherwise, it rejects (4) the request. This way, DX can send several requests to the device while all the requests from other domains are rejected. In order for another domain to access the module, the module needs to be reset (5). After the reset, the module returns to the "device not used" state. There are several options for when to reset the module, each with its advantages and disadvantages:

Lock-use-reset-and-free mode: An actor from a domain sends a lock request to a PG of a module. The PG checks if the module is already locked. If it's not locked, it confirms the lock; otherwise, it rejects it. While a domain locks a device, only actors from that domain can access it. After the domain no longer needs the module, it releases the lock, and other domains can access the module. When a release-lock request is received, PG resets the module and then releases the lock. The problem with this approach is that any domain can perform a DoS attack by never releasing the lock, and the approach does not protect against timing attacks.

Reset-on-store mode: Each time a store request is made, the PG resets the module, marks the domain that sends the store request as the domain that uses the module, and forwards the store request to the module. Only the domain marked as the one that is using the module can perform load requests. Otherwise, the load requests are rejected.

This simple approach guarantees that no explicit information flow occurs between two domains. Still, it does not provide a way for other modules to check if a module is currently being used. This can be fixed by PG exposing this information to domains, for example, by allowing a flag to be read from a specific address that marks if a module is currently in use. This approach also has a problem of actors in other domains performing DoS attacks by constantly sending store requests, and it also does not protect against availability-based attacks.

All the suggested modes have problems with DoS attacks and timing attacks. We can address DoS attacks and timing attacks by introducing a fair scheduling mechanism and limiting the amount of time a specific domain can use the module.

6.2.3.3 PG Operation Mode: Time-sharing with Context-switching

Similar to time-sharing with reset mode, the time-sharing with context-switching mode only allows a single domain to access the module simultaneously. Still, instead of resetting the module, it stores the module's state until the domain uses the module for the next time. This requires the module to provide a store and load state mechanism that allows the PG to store the existing state and load the new state corresponding to the next domain that will use the module. The PG can use the load and store mechanism provided by the module to implement a "context switching" mechanism, which allows the module to be time-shared in a similar way as the execution time on the core is shared by the operating systems between different threads of execution. Such a mechanism allows the module, with the help of the PG, to effectively have a separate state for each domain.

When switching between domains, the PG performs the following steps:

1. stop accepting new (store/load) requests from the current domain,
2. store the current state of the module,
3. loads the state corresponding to the new domain and
4. start accepting requests from the new domain.

Requiring the module to provide the store and load mechanism for its entire state has several larger implications on the module design that can limit when such a mechanism can be provided.

First, it requires that the module defines points in its execution where the entire state of the module is accessible so that it can be stored and a new state loaded. This includes the states of all the stateful components of the module, which includes components explicitly intended to store state, for example, registers, and components that implicitly carry state across several cycles, such as combinational circuits. Identifying points where the entire state is accessible is vital so that the entire state is captured by storing and that no previous state is left on the module when loading a new state. For example, to avoid leaving a part of the state inside of a combinational circuit that performs its function over several clock cycles is to allow a state to be accessed at the beginning of the circuit's execution where input from other parts of the modules (e.g., registers) is taken.

Second, to avoid timing side-channel attacks, the points in the execution where the state can be stored and loaded need to be available on a fixed interval, allowing the state to be switched on fixed intervals so that a different domain can access the module. Figure 18 shows the execution time of a module where the state can be switched on n cycles.

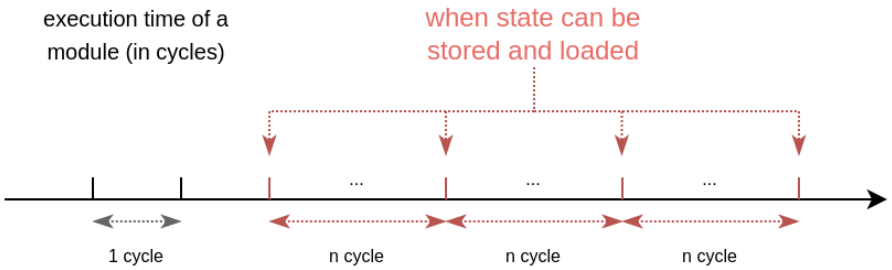

Figure 18: An execution time of a module in cycles that shows when the state of the module can be stored and loaded

If the above conditions can be met, we can use the PG to implement a round-robin scheduling mechanism to grant the module access at fixed intervals. By using the round-robin scheduling with fixed intervals, we prevent:

- DoS attacks because the module users have no control over when access to the modules is granted to them, and
- timing attacks because the user can only access the module on fixed periods for a fixed amount of time.

Note that access to the module is given to a domain regardless of whether the domain needs the access. Therefore, this approach can still underutilize the module equivalent to 1/k of the module's time if k domains share the module and one domain is not using it. Furthermore, designing a module to allow the use of the load and store mechanism could lead to a module with a worse performance than a module without the load and store mechanism.

Note that different scheduling systems can achieve various guarantees related to the timing side-channel attacks.

6.2.4 Integrating PG with the CROSSCON SoC

To use the module with the PG and preserve the isolation between domains, we need to integrate the module with the rest of the SoC in a way that does not undermine the existing isolation. How this can be done highly depends on the system's existing isolation mechanism and other guarantees we want to preserve. Following is a basic example of integrating a cryptographic accelerator (AES / RSA) and a PG with the CROSSCON SoC and the CROSSCON Hypervisor.

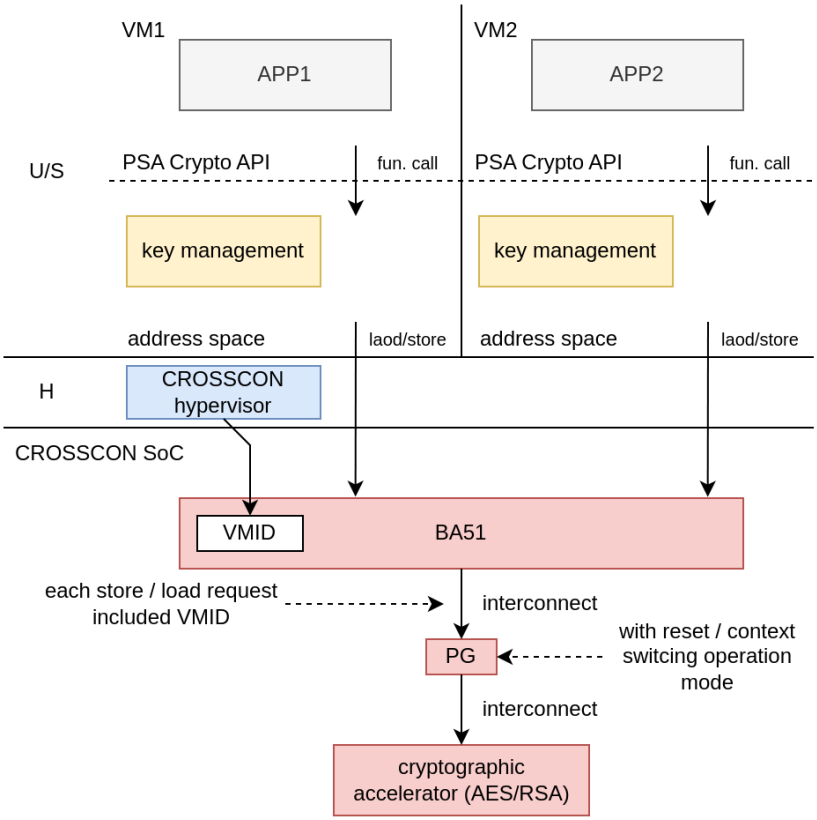

Figure 19: A basic example of how a cryptographic accelerator and PG can be integrated within the CROSSCON SoC and the CROSSCON Hypervisor

Figure 19 shows a basic setup where we use the CROSSCON SoC and the CROSSCON Hypervisor as a separation kernel to get isolated execution environments called virtual machines (VM). We assume that the CROSSCON Hypervisor is instantiated as in the MCU - RISC-V (M/S/U) setup, as described in Sect. 5.3.3. Using bus interconnect, we connect the cryptographic accelerator through the PG to the BA5x core. We assume that the PG is configured so that only VM1 and VM2 can access the module, and the accelerator can be time-shared between the VMs in reset or context-switching operation mode. We modify the BA51 core with additional instruction that allows the CROSSCON Hypervisor to

store a VM identifier (VMID) in the VM register of the BA51 core. The CROSSCON Hypervisor changes the VMID upon each context switch to identify the VM currently running on the core. The VMID is forwarded by the BA51 to the PG with each store and load request so that the PG knows from which VM the request is coming. This allows the PG to decide which request should be passed to the cryptographic accelerator according to the PG configuration. The accelerator's interface is memory-mapped to the address space of both VMs so that applications running inside of the VMs can interact with the accelerator. Note that the CROSSCON Hypervisor does not need to use the PMP to restrict access to the module; that is the responsibility of the PG. To simplify the use of the accelerator, the accelerator's functionality is exposed to the applications through a library that implements PSA Crypto API [38], drivers the accelerator, and provides other cryptographic primitives together with key management.

6.3 Using PG with Arbitrary SoC

Until now, we mostly considered setups where we have an MPU that supports a separation kernel with multiple domains and several external modules/devices. We can generalize this setup to an arbitrary SoC where we divide resources, including HW modules, into different domains by using PG and existing separation mechanisms.

The idea of dividing resources into different domains on the SoC level is not new and is already supported by some solutions. For example, SiFive's WorldGuard [32] and Arm's TrustZone [33] divide the SoC's resources into two or more worlds that are isolated from each other. The difference between WorldGuard, TrustZone, and our solution is that we primarily focus 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 worlds 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 a SoC with TrustZone support.

As far as we know, WorldGuard and TrustZone do not focus on a general problem of how to share a module between domains while preserving existing guarantees provided by a separation mechanism, although both solutions address this problem in some (less general) way specific for their setup.

7 Conclusions

In this document, we have presented the first draft 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 will be 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. During the course of the technical work in these work packages, potential required changes are communicated back to WP 2, which will lead to adaptations or realignment of parts of the architecture. The resulting finalized version of the architecture will then be documented in the final version of the CROSSCON open specification in the form of deliverable D2.3.

8 References

[1] CROSSCON WP1, "Deliverable D1.1: Use Cases Definition Initial Version," 2023.

[2] CROSSCON WP1, "Deliverable D1.2: Requirements Elicitation Initial Technical Specification," 2023.

[3] CROSSCON WP3, "Deliverable D3.1: CROSSCON Open Security Stack Documentation," 2023.

[4] CROSSCON WP4, "Deliverable D4.1: CROSSCON Extensions to Domain Specific Architectures Documentation," 2024.

[5] CROSSCON WP2, "Deliverable D2.3: CROSSCON Open Specification," 2025.

[6] GobalPlatform, "TEE Internal Core API 1.3.1," 2023. [Online]. Available: https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[7] D. Cerdeira, J. Martins, N. Santos and S. Pinto, "ReZone: Disarming TrustZone with TEE Privilege Reduction," in *USENIX Security Symposium*, 2022.

[8] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović and D. Song, "Keystone: An open framework for architecting trusted execution environments.," in *European Conference on Computer Systems (EuroSys)*, 2020.

[9] A. Ferraiuolo, A. Baumann, C. Hawblitzel and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Symposium on Operating Systems Principles (SOSP)*, 2017.

[10] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi and E. Stapf, "Sanctuary: ARMing TrustZone with User-space Enclaves," in *The Network and Distributed System Security Symposium (NDSS)*, 2019.

[11] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell and R. Kolanski, "Comprehensive Formal Verification of an OS Microkernel," *ACM Transactions on Computer Systems,* vol. 32, no. 1, 2014.

[12] "Sel4 - The Proof," [Online]. Available: https://sel4.systems/Info/FAQ/proof.pml.

[13] Saar Amar, Tony Chen, David Chisnall, et al., "CHERIoT: Rethinking security for low-cost embedded systems," Microsoft Technical Report MSR-TR-2023-6, 2023.

[14] R. N. M. Watson, P. G. Neumann, J. Woodruff, et al., "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture," University of Cambridge, Computer Laboratory, 2020.

[15] M. Wilding and D. Grave, "A Separation Kernel Formal Security Policy," in *International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.

[16] E. M. Clarke, O. Grumberg, D. Korening, D. Peled and H. Veith, Model checking, 2nd Edition, MIT Press, 2018.

[17] C. Barrett and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Model Checking*, vol. 336, Springer, 2018.

[18] A. Gupta, K. L. McMillan and Z. Fu, "Automated assumption generation for compositional verification," *Formal Methods in System Design,* vol. 32, no. 3, 2008.

[19] M. Grisafi, M. Ammar, M. Roveri and B. Crispo, "PISTIS: Trusted Computing Architecture for Low-end Embedded Systems," in *USENIX Security Symposium*, 2022.

[20] Arm, "Trusted Firmware-A (TF-A)," [Online]. Available: https://www.trustedfirmware.org/projects/tf-a/.

[21] RISC-V Internation, "RISC-V Open Source Supervisor Binary Interface (OpenSBI)," [Online]. Available: https://github.com/riscv-software-src/opensbi.

[22] S. Pereira, J. Sousa, P. Sandro, J. Martins and D. Cerdeira, "Bao-Enclave: Virtualization-based Enclaves for Arm," in *IEEE World Forum on Internet of Things (WF-IoT)*, 2022.

[23] Arm, "ARM Generic Interrupt Controller Architecture version 2.0 - Architecture Specification," [Online]. Available: https://developer.arm.com/documentation/ihi0048/latest/.

[24] Arm, "Arm Generic Interrupt Controller Architecture Specification GIC A version 3 and version 4," [Online]. Available: https://developer.arm.com/documentation/ihi0069/h/.

[25] Arm, "System MMU Support," [Online]. Available: https://developer.arm.com/Architectures/System%20MMU%20Support.

[26] Arm, "ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0," [Online]. Available: https://developer.arm.com/documentation/ihi0062/dc/?lang=en.

[27] RISC-V International, "RISC-V Platform-Level Interrupt Controller Specification," [Online]. Available: https://github.com/riscv/riscv-plic-spec.

[28] Arm, "SMC Calling Convention (SMCCC)," [Online]. Available: https://developer.arm.com/documentation/den0028/latest.

[29] RISC-V International, "RISC-V SBI specification," [Online]. Available: https://github.com/riscv-non-isa/riscv-sbi-doc.

[30] Arm, "Arm Power State Coordination Interface Platform Design Document," [Online]. Available: https://developer.arm.com/documentation/den0022/e.

[31] Arm, "GIC," [Online]. Available: https://developer.arm.com/Architectures/Generic%20Interrupt%20Controller.

[32] SiFive, "SiFive WorldGuard Technical Paper," 2021.

[33] Arm, "ARM Security Technology, Building a Secure System using TrustZone Technology, Issue C," 2009.

[34] M. Sabt, M. Achemlal and A. Bouabdallah, "Trusted Execution Environment: What It Is, and What It Is Not," in *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2015.

[35] W. Hu, A. Ardeshiricham and R. Kastner, "Hardware Information Flow Tracking," *ACM Computing Surveys,* vol. 54, no. 4, 2021.

[36] Z. Jiang, H. Jin, E. Suh and Z. Zhang, "Designing Secure Cryptographic Accelerators with Information Flow Enforcement: A Case Study on AES," in *Design Automation Conference*, 2019.

[37] J. Rushby, "Noninterference, Transitivity, and Channel-Control Security Policies," SRI International, Computer Science Laboratory., 1992.

[38] Arm, "PSA Crypto API 1.1.2," 2023. [Online]. Available: https://arm-software.github.io/psa-api/crypto/.

[39] M. Breskvar, "Systems and methods for data-driven secure and safe computing". Patent US11645425B2, 2020.