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

D3.3 CROSSCON Open Security Stack Documentation - Final

Document Identification			
Status	Final	**Due Date**	31/08/2025
Version	1.0	**Submission Date**	29/08/2025

Related WP	WP3	**Document Reference**	D3.3
Related Deliverable(s)	D1.4, D1.5, D1.6, D2.1, D2.3, D3.1, D3.2, D3.4, D4.3, and D5.4	**Dissemination Level(*)**	PU
Lead Participant	UWU	**Lead Author**	Melanie Götz (UWU)
Contributors	UMINHO, UNITN, UWU	**Reviewers**	UMINHO, SLAB

Keywords
TEE Isolation and Abstraction, CROSSCON Hypervisor, CROSSCON New Trusted Services, CROSSCON TEE Toolchain, CROSSCON Bare-metal TEE.

This document is issued within the frame and for the purpose of the CROSSCON project. This project has received funding from the European Union's Horizon Europe Programme under Grant Agreement No.101070537. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

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

Document Information

List of contributors	
Name	**Partner**
Melanie Götz	UWU
Tymoteusz Burak	UWU
Fabian Schmitt	UWU
Hamid Dashtbani	UWU
Alberto Tacchella	UNITN
Michele Grisafi	UNITN
Carlo Ramponi	UNITN
Nikhilesh Singh	TUD
João Sousa	UMINHO
David Cerdeira	UMINHO
Luís Cunha	UMINHO
Daniel Oliveira	UMINHO
Manuel Rodriguez	UMINHO
Tiago Gomes	UMINHO
Sandro Pinto	UMINHO
Gergely Eberhardt	SLAB
Hristo Koshutanski	ATOS

Document history			
Ver.	**Date**	**Change editors**	**Changes**
0.1	19/11/2024	Christoph Sendner (UWU)	First draft - Initial Contributions;
0.2	30/05/2025	João Sousa, David Cerdeira, Luís Cunha, Daniel Oliveira, Manuel Rodrigues, Tiago Gomes, Sandro Pinto (UMINHO)	Update Chapter 2.
0.3	30/06/2025	João Sousa, David Cerdeira, Luís Cunha, Daniel Oliveira, Manuel Rodrigues, Tiago Gomes, Sandro Pinto (UMINHO)	Updated Chapter 3 to improve the previous D3.1 content. Specifically, updating and finalizing TEE isolation and Abstraction sections.
0.4	28/07/2025	Melanie Götz, Tymoteusz Burak, Fabian Schmitt, Hamid Dashtbani (UWU), Alberto Tacchella, Michele Grisafi, Carlo Ramponi (UNITN), Gergely Eberhardt (SLAB)	Revise and expand sections 3.3.1-3.3.5 with the new results.
0.5	31/07/2025	João Sousa, David Cerdeira, Luís Cunha, Daniel Oliveira, Manuel Rodrigues, Tiago Gomes, Sandro Pinto (UMINHO)	Updated Chapter 3 to improve the previous D3.1 content. Specifically, updating and finalizing CROSSCON Hypervisor sections.
0.6	06/08/2025	Melanie Götz, Tymoteusz Burak, Fabian Schmitt, Hamid Dashtbani (UWU)	Update Executive Summary, Introduction, Conclusions and ensure consistent formatting.
0.7	7/08/2025	Nikhilesh Singh (TUD)	Moved the TUD-relevant updated sections from the working copy document to the current document.
0.8	10/08/2025	Hristo Koshutanski (ATOS)	Updated Section 3.3.5 with the new results of network anomaly detection.
0.9	25/08/2025	Melanie Götz, Tymoteusz Burak, Fabian Schmitt, Hamid Dashtbani (UWU), Alberto Tacchella (UNITN)	Address reviewers' comments.
0.91	28/08/2025	Juan Alonso (Atos)	Quality Assessment.
1.0	29/08/2025	Hristo Koshutanski (Atos)	Final version submitted.

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Melanie Götz (UWU)	27/08/2025
Quality Manager	Juan Alonso (ATOS)	28/08/2025
Project Coordinator	Hristo Koshutanski (ATOS)	29/08/2025

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
ABI	Application Binary Interface
ACP	Accelerator Coherency Port
ACU	Auxiliary Control Unit
AES	Advanced Encryption Standard
AIA	Advanced Interrupt Architecture
AIRCR	Application Interrupt and Reset Control Register
API	Application Programming Interface
APLIC	Advanced PLIC
APU	Application Processing Unit
ASLR	Address Space Layout Randomization
BP	Blog Posts
CA	Client Application
CBOR	Concise Binary Object Representation
CCA	Confidential Computing Architecture
CFG	Control Flow Graph
CI/CD	Continuous Integration / Continuous Development
COSE	CBOR Object Signing and Encryption
CoT	Chain of Trust
COTS	Commercial Off-The-Shelf
CoVE	Confidential VM Extension
CPC	Cooperating Proof Calculus
CPU	Central Processing Unit
CRP	Challenge-Response Pairs
CSI	Channel State Information
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerabilities Scoring System
CWE	Common Weakness Enumeration
DAST	Dynamic Application Security Testing
DMA	Direct Memory Access
DRM	Digital Rights Management
DRTM	Dynamic RTM
DT	Device Tree
EL	Exception Level
ePMP	PMP-enhancement extension
FF-A	Firmware Framework for Arm
FOSE	Firmware Object Signing and Encryption
FOTA	Firmware OTA
FPGA	Field-Programmable Gate Array
FSA	Finite State Automaton
FuSa	Functional Safety
GIC	General Interrupt Controller
GP	GlobalPlatform

GPOS	General Purpose OS
GPT	Granule Page Tables
GPU	Graphic Processing Unit
HAL	Hardware Abstraction Layer
HS-mode	Hypervisor-extended Supervisor mode
I/O	Input/Output
IaaS	Infrastructure as a Service
IAST	Interactive Application Security Testing
IDAU	Implementation-Defined Attribution Unit
IETF	Internet Engineering Task Force
IMSIC	Incoming Message-Signaled Interrupt Controller
IOMMU	IO Memory Management Unit
IOMPU	IO Memory Protection Unit
IOPMP	IO Physical Memory Protection
IoT	Internet of Things
IOVA	IO Virtual Addresses
IP	Intellectual Property
IPA	Intermediate Physical Addresses
IPC	Inter-Partition Communication
IPI	Inter-Processor Interrupt
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
JOSE	JSON Object Signing and Encryption
LLC	Last-Level Cache
M-mode	Machine mode
MAC	Message Authentication Code
MCS	Mixed-Criticality System
MCU	Microcontroller Unit
ML	Machine Learning
MMIO	Memory-Mapped I/O
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSI	Message-Signaled Interrupt
MTE	Memory Tagging Extension
mTLS	mutual TLS
MTT	Memory Tracking Tables
NSC	Non-Secure Callable
NVD	National Vulnerability Database
NVIC	Nested Vectored Interrupt Controller
OS	Operating Systems
OTA	Over-the-Air
OTF	On-The-Fly Encryption/Decryption
OTP	One-Time-Programmable
PAC	Peripheral Access Controller
pCPU	Physical CPU
PLIC	Platform Local Interrupt Controller
PMP	Physical Memory Protection
PMU	Performance Monitor Unit
PPC	Peripheral Protection Controller

PSCI	Power State Coordination Interface
PSP	AMD Secure Processor
pTA	pseudo Trusted Application
PUF	Physical Unclonable Function
QoS	Quality of Service
RDC	Resource Domain Controller
REE	Rich Execution Environment
RFF	Radio Frequency Fingerprinting
RFFI	RFF Identification
RMM	Realm Manager Monitor
RNG	Random Number Generator
ROP	Return Oriented Programming
RoT	Root of Trust
RSA	Rivest-Shamir-Adleman
RTM	Root-of-Trust Measurement
RTOS	Real-Time OS
RTU	Real-Time Processing Unit
S-mode	Supervisor mode
SAST	Static Application Security Testing
SAU	Security Attribution Unit
SB	Security Bulletin
SBI	Supervisor Binary Interface
SBOM	Software Bill Of Materials
SCA	Software Composition Analysis
SDEI	Software-Delegated Exception Interface
sdGPOS	Software-Defined GPOS
sdRTOS	Software-Defined RTOS
sdTEE	Software-Defined TEE
SEAM	SEcure-Arbitration Mode
SEV	Secure Encrypted Virtualization
SEV-ES	Secure Encrypted Virtualization-Encrypted State
SEV-SNP	Secure Encrypted Virtualization-Secure Nested Paging
SGX	Software Guard Extensions
SIEM	Security Information and Event Management
SLoC	Source Lines of Code
SMC	Secure Monitor Call
SMCCC	Secure Monitor Call Calling Convention
SmMTT	RISC-V Supervisor Domains Access Protection
SMMU	System Memory-Management Unit
SMP	Symmetric Multiprocessing
SMT	Satisfiability Modulo Theory
SoC	System on Chip
SOT	System Orchestrator Tool
SP	Scientific Publications
SPH	Static Partitioning Hypervisor
sPMP	supervisor Physical Memory Protection
SRTM	Static RTM
SUIT	Software Updates for Internet of Things
TA	Trusted Application

TCB	Trusted Computing Base
TD	Trusted Domains
TDX	Trust Domain Extensions
TEE	Trusted Execution Environment
TF	Trusted Firmware
TinyML	Tiny Machine Learning
TLB	Translation Lookaside Buffer
TOCTOU	Time-Of-Check-to-Time-Of-Use
TPM	Trusted Platform Module
tRTS	trusted Runtime System
TSM	Trusted Security Manager
TZASC	TrustZone Address Space Controller
U-mode	User mode
UC	Use Case
uRTS	untrusted Runtime System
UUID	Universally Unique Identifier
vCPU	Virtual CPU
VE	Virtualization Extensions
vFPGA	virtual FPGA
VM	Virtual Machine
VMM	VM Monitor
VS-mode	Virtual Supervisor mode
VU-mode	Virtual User mode
WP	Work Package
XMPU	Xilinx Memory Protection Units
XPPU	Xilinx Peripheral Protection Units

Executive Summary

The Internet of Things (IoT) ecosystem is inherently heterogeneous, encompassing a wide range of devices, from low-power Microcontroller Unit (MCU) with minimal security capabilities to high-performance, multi-core Application Processing Unit (APU) equipped with reconfigurable hardware. A typical IoT system spans multiple layers, including hardware, firmware, and the Operating Systems (OS), each adding complexity and expanding the attack surface. This diversity introduces significant security challenges, such as the lack of interoperability and isolation between different Trusted Execution Environment (TEE) implementations, the difficulty of delivering dynamic and per-Virtual Machine (VM) services without inflating the Trusted Computing Base (TCB), the absence of multi-factor authentication trusted services utilizing Physical Unclonable Function (PUF) or Wi-Fi environmental fingerprinting, the challenges in secure firmware updates and cross-compilation, and the limited security capabilities of bare-metal devices.

This deliverable summarizes the final results of WP3, which focused on the development of the CROSSCON Open Security Stack. It presents the stack's architecture, core components, and the progress achieved toward building a flexible, portable, and secure solution tailored to the heterogeneous IoT landscape. The CROSSCON Open Security Stack is designed to address the aforementioned challenges by enabling robust isolation mechanisms, supporting dynamic workloads, and ensuring portability across diverse edge platforms.

The work includes: the decomposition of trusted services from the trusted kernel to improve TEE abstraction and isolation; the design of the CROSSCON Hypervisor, which is based on static partitioning but has been extended with dynamic mechanisms to support per-VM isolation and dynamic TEE creation the development of novel trusted services, such as PUF based and context-aware authentication mechanisms; the creation of the CROSSCON Toolchain, which facilitates secure compilation and firmware updates; and the implementation of the CROSSCON Bare-Metal TEE, aimed at delivering TEE-level security guarantees to resource-constrained devices.

These efforts are grounded in the specifications and requirements defined in WP1 and WP2. The results presented here—including our initial analysis, architectural design, and planned next steps—lay a solid foundation for the integrated CROSSCON stack to be delivered in WP5.

1 Introduction

1.1 Purpose of the Document

This document selects target platforms for CROSSCON stack development, presents an initial version of the CROSSCON open security stack, and outlines the research and development progress of its components. It includes a comprehensive literature review on TEE technologies and their primary vulnerabilities, and proposes TEE isolation mechanisms to decompose trusted services from trusted kernels. Moreover, it explores virtualization technologies, specifically static partitioning hypervisors, and their potential to increase isolation and security assurances at both the architectural and microarchitectural levels. It discusses the features and implementation of the CROSSCON Hypervisor in detail. This document also emphasizes the significance of creating innovative trusted services, complementary to existing ones, and describes such novel trusted services and their functionalities, architectures, and implementation. Furthermore, it examines the current update mechanisms utilized in IoT and conducts a literature review on secure compilation proposals, and establishes the CROSSCON Toolchain. Finally, it discusses the existing solutions for implementing TEEs on low-end devices and details the design of the proposed Baremetal TEEs.

1.2 Relation to Other Project Work

Building on deliverables D2.3 and D1.5, this document presents the research and development results of the CROSSCON stack components initially proposed in D3.1. Particularly relevant are the draft specifications laid out in document D2.3 and the device classification and requirements established in deliverable D1.5. Deliverable D3.4 is closely related to this document as it provides the source code repositories where the current source code of the work done in WP3 can be found, along with documentation for its use. Furthermore, the components presented in this document will be subject to validation tests which are specified in deliverable D1.6. Results of these tests will contribute to the D5.6. This will entail providing the foundational building blocks of the CROSSCON Stack to WP5, which will integrate them into a functional prototype.

1.3 Structure of the Document

This document focuses on different aspects of the CROSSCON Open Security Stack. It begins with the platform selection, Section 2, detailing the security capabilities of the selected platforms for CROSSCON and how they map to each partner and Use Case (UC). Following this, the document presents the research and development results of the CROSSCON security stack components. Specifically, it addresses TEE isolation and abstraction, Section 3.1, CROSSCON Hypervisor features and design, Section 3.2, novel CROSSCON trusted services, Section 3.3, CROSSCON TEE Toolchain, Section 3.4, and the development of CROSSCON Bare-Metal TEE, Section 3.5. The document concludes by summarizing key findings in Section 4.

1.3.1 Changes w.r.t. D3.1

This document extends the draft presented in D3.1. Section 3.1.3 expands the original discussion of TEE vulnerabilities into a more comprehensive TEE analysis, providing an extended assessment of threat vectors and mitigation strategies. Section 3.1.5 has been significantly elaborated to include a detailed

description of the design and implementation of the proposed TEE solution, clarifying the architectural choices and integration aspects. A new section, Section 3.2.6, introduces CROSSCON Multiple VM Monitor (VMM) Support, outlining its motivation, design, and implications for secure virtualization. Section 3.3.1 has been extended to present mechanisms of novel PUF authentication schemes, as well as their relation to the GlobalPlatform (GP) model, providing a stronger basis for evaluating trust anchors. Section 3.3.2 elaborates the context-based authentication mechanisms with a detailed description of the verifier, enrolment, and authentication phases, as well as the implementation details of the trusted service integrated into the CROSSCON Hypervisor. A new section on Remote Attestation Section 3.3.3 has been added, introducing the fundamental concepts of attestation, the prover–verifier interaction model, and its critical role in ensuring integrity and trustworthiness across distributed and heterogeneous environments. In addition, a dedicated section on the CROSSCON TEE Toolchain 3.4 has been included, describing its role in supporting secure update mechanisms and its integration with DevSecOps platforms as well as a section on the CROSSCON Bare-Metal TEE 3.5, covering requirements, platform evaluation, and implementation, including both Memory Protection Unit (MPU) and non-MPU variants. Information on the Perimeted Guard discussion has been moved to D4.3.

2 Platform Selection

CROSSCON aims to develop a new, open, flexible, and highly portable IoT security stack that can operate on various edge devices and multiple hardware platforms. The following section presents the methodology used for selecting target platforms for CROSSCON development.

The platform selection uses several key outputs from previously submitted CROSSCON deliverables, including the Device Classification from D1.5 and the CROSSCON instantiation options from D2.3. All platform-related information from these deliverables was considered in the platform selection process.

Summary of CROSSCON Device Classification (D1.5): To classify IoT devices in terms of security, deliverable D1.5 proposes a device classification process based on the provided security capabilities/services and security guarantees.

Class 0 (NO SECURITY): Class 0 devices are the most resource-constrained, usually low-cost, and offer ultra-low-power operation. Because of their limited resources, they are not adequate to perform critical functions and do not provide any hardware security guarantee. Additionally, these devices rely entirely on software-based security, which makes them more vulnerable to attacks.

Class 1 (BASIC SECURITY): Class 1 devices are resource constrained, but feature basic security capabilities (e.g., MPU) and a small number of privilege levels (typically two). Despite featuring a more secure stack than Class 0 devices, these devices are still vulnerable to several attacks because of their reduced security capabilities.

Class 2 (STRONG SECURITY): Class 2 devices contain some integrated or discrete security hardware resources (e.g., secure storage, on-time programmable memories) and architectures with additional privileged levels that enable some form of isolated execution. Due to their capabilities, they can provide stronger security guarantees than Class 1 devices.

Class 3 (EXTENDED SECURITY): Class 3 devices provide the most security capabilities and guarantees. They typically include advanced security hardware resources (e.g., Random Number Generator (RNG), PUF, HW-based intrusion detection) and their architectures support for multiple privileged-levels, including hypervisor and TEE support.

Within these classes, it is useful to distinguish between the performance levels of the device. Here we describe two main performance classes MCU and APU. Nonetheless, our considerations extend across a wide range of platforms, encompassing devices positioned at various points along this spectrum.

APU: An APU typically features a powerful Central Processing Unit (CPU) with multiple cores, targeting general-purpose tasks. They typically support General Purpose OS (GPOS), such as Linux, by featuring memory virtualization hardware (e.g., Memory Management Unit (MMU)), and often include virtualization extensions to support the execution of a hypervisor.

MCU: MCUs commonly feature low power features when compared to APUs. An MCU typically includes a processing unit, memory and some peripherals on the same chip. They excel in meeting embedded applications' real-time and low-power requirements and feature several integrated peripherals for diverse functionalities. They may support real-time OS, e.g., FreeRTOS, lacking memory virtualization hardware capabilities required by feature-rich OSes.

Summary of CROSSCON Design Specifications (D2.3): The CROSSCON stack is designed to support multiple architectures and vendors across a wide range of implementation scenarios. These scenarios are

referred to as CROSSCON instantiation options, each representing a different configuration of the CROSSCON stack. Based on the functionalities provided by CROSSCON components (e.g., CROSSCON Hypervisor, CROSSCON Bare-metal TEE) and the security features available on each platform (e.g., virtualization extensions, TEE, secure boot, RNG, PUF), six instantiation options were defined. The complete list is provided in D2.3. Supported architectures include ArmV7-A, ArmV8-A (prior to v8.4), RISC-V for APUs, and TI MSP, AVR, ArmV6-M, ArmV7-M, ArmV8-M, ArmV8-R, and RISC-V for MCUs.

To bind selected platforms with appropriate instantiation options, we evaluated the security capabilities of each platform alongside the requirements of CROSSCON components (e.g., CROSSCON Hypervisor, CROSSCON Baremetal TEE, new Trusted Application (TA)). The functionality of each CROSSCON component depends on the hardware features available on the target platform. For instance, platforms equipped with hardware virtualization support, such as a second-stage MMU, can support the CROSSCON Hypervisor. In contrast, platforms with only two privilege levels and basic security primitives rely on the CROSSCON Baremetal TEE to enforce isolation. Based on these considerations, CROSSCON defines the following instantiation options:

SW-only isolation environment (i): This scenario addresses the security requirements of resource-constrained low-end devices with no hardware resource protection. In this instantiation option, CROSSCON adopts software-based techniques to ensure isolation between normal applications and TAs.

Basic memory isolation environment (ii): This scenario addresses the security requirements of resource-constrained devices with basic memory isolation technologies. In this instantiation option, CROSSCON leverages basic security primitives, e.g., MPU, to ensure isolation between normal applications and TAs.

TEE-less environment with virtualization (iii): A platform can have dedicated hardware that facilitates hypervisor implementations but lacks dedicated TEE hardware. In this CROSSCON instantiation option, the hypervisor might run above the platform's firmware, if present, managing guests and attributing physical resources to them. In platforms equipped with an APU, the hypervisor leverages virtual memory, while in MCU-enabled devices, it relies on hardware security primitives like 2nd stage MPU.

Virtualization-less environment with TEE (iv): This CROSSCON instantiation option addresses platforms with TEE support but without virtualization extensions. In this CROSSCON instantiation option, the hypervisor component operates in privileged secure world side, managing guests running in non-secure world part. Since most of APU platforms include virtualization extensions, this scenario is exclusive to the MCU platforms.

Environment with TEE and Virtualization (v): This CROSSCON instantiation option combines the flexibility of the hypervisor with TEE security guarantees. It includes all isolation capabilities provided by the CROSSCON architecture. In this configuration, the CROSSCON Hypervisor executes at a dedicated privileged level on the non-secure side, while certain TEE hardware components can assist the hypervisor in specific critical services. Additionally, with support from other platform security features, this option also features mechanisms used by CROSSCON to enable the creation and isolation of multiple zones within the secure world.

Environment with Virtualization, TEE, and Field-Programmable Gate Array (FPGA) (vi): A platform may include accelerators deployed in the FPGA fabric. This instantiation option demonstrates CROSSCON's awareness of the interface with these components and the associated security implications of FPGA-enabled devices. In this configuration, the CROSSCON Hypervisor runs at a dedicated, privileged level on the non-secure side, allowing guest systems to request FPGA services. These services are also available to components running in the secure world.

In addition to the CROSSCON instantiation options, some contexts may require executing TEE components outside of the hardware TEE environment. This applies in cases where the platform lacks dedicated hardware for supporting multiple trusted components. In such cases, isolation can be achieved using

virtualization or hardware features designed for TEEs. Section 3.2.5.2 provides further details on this approach, introducing a novel feature of the CROSSCON Hypervisor: the per-VM TEE.

2.1 Platform Analysis and Selection

The CROSSCON stack is specified in a generic and vendor-independent manner. However, its design must consider the specific hardware interactions of the target platforms with development of software components. Therefore, representative platforms were selected to guide the development of the software stack components.

Based on the CROSSCON device classification and the available instantiation options, we selected a set of platforms to cover at least one platform architecture, primarily RISC-V or Arm, and one representative for each CROSSCON instantiation type. The selected platforms include: MSP430F5529LP, NUCLEO-G0B1RE, NUCLEO-H743ZI2, ESP32-C3-AWS-ExpressLink-DevKit, NXP LPC55S6x, Arty-100T (BA5x), Beagle-V, Raspberry Pi 4B, BeagleBone AI-64, ZCU 102, and Genesys 2.

Security Capabilities. Table 1 summarizes the security features of these platforms. The features include support for TEE technologies, memory and Input/Output (I/O) isolation, cryptographic accelerators, On-The-Fly Encryption/Decryption (OTF) capabilities, RNG, PUF, Universally Unique Identifier (UUID), Secure Elements, Secure Boot, and One-Time-Programmable (OTP) memory. Additionally, each platform is categorized by its target application domain, such as APU, MCU, or platforms with FPGA support.

Table 1: Features of platforms selected for CROSSCON.

	Name	TEE	IO-Isol.	Core-Isol.	Crypt. Accel.	OTF	RNG	PUF	UUID	Sec. Elem.	Sec. Boot	OTP
MCU	MSP430F5529LP	-	-	-	-	-	-	-	-	-	-	-
	NUCLEO-H743ZI2	-	-	MPU	-	-	✓	-	-	✓	-	-
	NUCLEO-G0B1RE (STM32G0B1RE)	-	-	MPU	-	-	-	-	-	-	✓	✓
	ESP32-C3-AWS-ExpressLink-DevKit	-	-	PMP	✓	✓	✓	-	-	-	✓	✓
	NXP LPC55S6x (LPC55S6x)	TF-M	✓	MPU	✓	✓	✓	✓	✓	✓	✓	✓
	Arty-100T (BA51H)	PMP|SPMP	✓	PMP|SPMP	-	-	-	-	-	-	-	-
APU	Beagle-V	PMP	-	-	-	-	✓	-	-	-	-	✓
	Raspberry PI 4B	-	-	MMU(2nstage)	-	-	✓	-	-	-	✓	✓
	Beaglebone AI-64	TZ-A	✓	MMU(2nstage)	✓	-	✓	-	-	-	✓	✓
FPGA	Genesys 2 (CVA6)	PMP|SPMP	✓	PMP|SPMP	✓	-	-	-	-	-	-	✓
	ZCU 102	TZ-A	✓	MMU(2nstage)	✓	-	-	✓	-	✓	✓	✓

With a clear understanding of the security capabilities of each platform, we can classify them according to their designated Class (0 to 3) and the corresponding CROSSCON instantiation option.

Class 0 Platforms: The *MSP430F5529LP* MCU platform lacks any hardware security primitives, relying entirely on software for hardware security guarantees. No APU platforms apply to this category.

Class 1 Platforms: Among MCU platforms, we select the *NUCLEO-G0B1RE (STM32G0B1RE)*, featuring MPU, secure boot, and OTF encryption/decryption and the *NUCLEO-H743ZI2*, equipped with MPU, RNG, and PUF. On the APU side, we select the Beagle-V, which includes Physical Memory Protection (PMP), RNG, and OTP functionalities.

Class 2 Platforms: The MCU segment features the *ESP32-C3-AWS-ExpressLink-Devkit*, offering capabilities such as PMP, RNG, cryptographic accelerators, secure boot, and OTP. For APU devices, the *Raspberry PI 4B* offers MMU (2nd stage), RNG, cryptographic accelerators, secure boot, and OTP, while the *Beaglebone AI-64* supports TEE via TrustZone-A, MMU (2nd stage), cryptographic accelerators, RNG, secure boot, and OTP.

Class 3 Platforms: The *NXP LPC55S6x* encompasses all identified security features. Platforms featuring advanced security support and an FPGA include the ZCU-102, Arty-100T, and Genesys2. Particularly the ZCU-102, which features an APU with Trustzone-A, MMU (2nd stage), cryptographic accelerators, PUF,

secure element, secure boot, and OTP.

Each platform matches the corresponding instantiation option, taking into account all relevant CROSSCON components (e.g., CROSSCON Hypervisor, CROSSCON Baremetal TEE, TA) and the platform's security capabilities. Table 2 summarizes the mapping between the selected platforms and device class and instantiation options. Additionally, it details the respective architectures, their availability to project partners, and how they map to the UCs.

Option 1 Platforms: The *MSP430F5529LP* was specifically chosen to meet the instantiation option ***i*** because of its Class 0 classification.

Option 2 Platforms: The platforms selected for instantiation option ***ii*** include the *NUCLEO-G0B1RE*, *NUCLEO-H743ZI2*, *ESP32-C3-AWS-ExpressLink-DevKit*, and the *Beagle-V*. Although the *Beagle-V* is based on a RISC-V architecture with a three-level privilege model (M, S, and U modes), and supports memory protection through Physical Memory Protection (PMP) and MMU, it is still considered limited in terms of comprehensive security features.

Option 3 Platforms: For instantiation option ***iii***, the chosen platforms are the *Raspberry Pi 4B* and the *Beaglebone AI-64*.

Option 4 Platforms: The *NXP LPC55S6* is the only board to support this instantiation option.

Option 5 Platforms: This instantiation option features the *Beaglebone AI-64* and the *Genesys2*. The *Genesys2* is inserted into this instantiation when running the CVA6 RISC-V core with virtualization support (H extension) and an IO Memory Management Unit (IOMMU).

Option 6 Platforms: This instantiation option requires the support for FPGA capabilities,TEEsupport, and virtualization support. Therefore, the platforms meeting these criteria are the *Arty-100T*, the *ZCU102*, and the *Genesys2*. The *Arty-100T* is categorized under this option when running the BA5x RISC-V core present in the CROSSCON SoC, while the *Genesys2* is included when running the CVA6 RISC-V core with virtualization support (H extension) and an IOMMU.

With a wide array of selected platforms targeting different security features and different performance

Table 2: Platform selection with the respective mapping to the class of the device, architecture, partner, instantiation option and to the UC.

	Name	Class	Arch	Partner	Insta. Option	UC
MCU	MSP430F5529LP	0	MSP430	UNITN	*(i)*	
	NUCLEO-G0B1RE	1	Armv6-M	UNITN	*(ii)*	
	NUCLEO-H743ZI2	1	Armv7-M	UNITN / UM	*(ii)*	
	ESP32-C3-AWS -ExpressLink-DevKit	2	RISC-V	UNITN / UM	*(ii)*	
	NXP LPC55S6x	3	Armv8-M	UWU / BIOT / UM / 3MDEB	*(iv)*	UC1
	Arty-100T (BA5x)	3	RISC-V	SLAB / UM / BEYOND	*(vi)*	
APU	Beagle-V	1	RISC-V	UM	*(ii)*	
	Raspberry PI 4B	2	Armv8-A	3MDEB / SLAB / CY / UWU / BIOT / UM	*(iii)*	UC1 / UC2 / UC3 / UC4
	Beaglebone AI-64	2	Armv8-A	UM	*(iii) and (v)*	
	Genesys2 (CVA6-H w/ SPMP for Hyp)	3	RISC-V	UM	*(v), (vi)*	
FPGA	ZCU 102	3	FPGA	TUD / UM	*(vi)*	UC5
	Genesys 2	3	FPGA	BEYOND	*(vi)*	

levels, UC provider partners selected the platforms that best match their UC: The *NXP LPC55S6* was selected for UC1; the *Raspberry Pi 4B* is selected by multiple UC provider partners, being featured in UC1, UC2, UC3, and UC4; lastly, the *ZCU102* is selected for UC5.

3 Research Results

This chapter presents the research and innovation activities undertaken for the design and development of the CROSSCON stack. The chapter begins with TEE isolation and abstraction, covering the analysis of TEE models, known vulnerabilities, isolation mechanisms, and proposed abstraction approaches. This is followed by the design of the CROSSCON Hypervisor, which examines virtualization and microarchitectural isolation techniques, evaluates hypervisor features, and selects the platform to serve as the foundation for CROSSCON Hypervisor development. The progress on two key features of the CROSSCON Hypervisor is reported: dynamic virtual machine creation and per-VM TEE service support. The next part focuses on the development of novel trusted services, including PUF-based authentication, remote attestation, environmental fingerprinting, FPGA-related services, behavioral analysis services, and control flow integrity mechanisms. This is followed by the CROSSCON TEE Toolchain, which reviews existing IoT update mechanisms and standards, identifies integration requirements with DevSecOps platforms, and provides a literature review on secure compilation. The chapter also outlines the design of the CROSSCON Secure Update process. Finally, the chapter presents the CROSSCON Bare-Metal TEE, detailing requirements, platform analysis, state-of-the-art approaches, and implementation. Both MPU-based and non-MPU variants are examined to address the security needs of resource-constrained devices.

3.1 TEE Isolation and Abstraction

This section examines CROSSCON integration with TEEs, focusing on two fundamental challenges: insufficient TEE isolation and the lack of TEE abstraction. The first issue stems from persistent vulnerabilities in TEE implementations. Between 2013 and mid-2018, over 200 TEE-related bugs were reported [1]. We update this analysis, and uncover that while TEE security has evolved over the past five years, many newer MCU TEE deployments still repeat earlier mistakes, failing to adopt critical security improvements. The second issue arises from the significant heterogeneity across TEE technologies. Due to the competitive TEE market (involving different TEE vendors), TEE technologies tend to be developed independently with proprietary features, resulting in heterogeneous TEE programming models. This fragmentation leads to incompatible programming models, which hinders software reuse, particularly for legacy applications.

We begin by contextualizing common TEE technologies from various vendors and classifying them according to shared design principles and aligned security goals. This is followed by an analysis of representative TEE models. Building on previous work analyzing vulnerabilities on TEE implementations using TZ-A technology [1], we (i) conduct a comprehensive reassessment of TrustZone-assisted TEEs on Cortex-A devices and (ii) analyze TrustZone-assisted TEEs on Cortex-M MCU over the 2019–2024 period. The findings support the development of additional isolation mechanisms to decompose a single TEE system stack into multiple, separate TEE system stacks. We then demonstrate how this decomposition is applied to TrustZone TEEs across APU, MCU, and Real-Time Processing Unit (RTU) platforms. Prior studies provide empirical evidence that this approach mitigates the impact of vulnerabilities inherent in monolithic TEE architectures [2, 3]. To conclude, we address the issue of TEE abstraction by proposing a solution to improve interoperability of trusted services across heterogeneous TEEs.

3.1.1 TEEs and TEE Technologies

By definition, a TEE is a secure environment within a computing device, typically implemented either with the main processor or in a separate chip. TEEs are widely used across a range of platforms, from mobile devices [1] to servers [4], and are designed to protect sensitive operations from unauthorized ac-

cess or modification. TEEs operate alongside a Rich Execution Environment (REE). The REE typically runs a general-purpose OS that utilizes the platform's hardware to provide standard functionalities, while the TEE leverages dedicated hardware resources or mechanisms to ensure isolated execution. This design ensures strong separation from components running in the REE. Common TEE use cases include mobile banking, Digital Rights Management (DRM), and secure key management. Figure 1 shows a system architecture incorporating a TEE, including dedicated hardware components such as trusted cores, trusted RAM, and trusted ROM, as well as trusted regions within shared memory and storage.

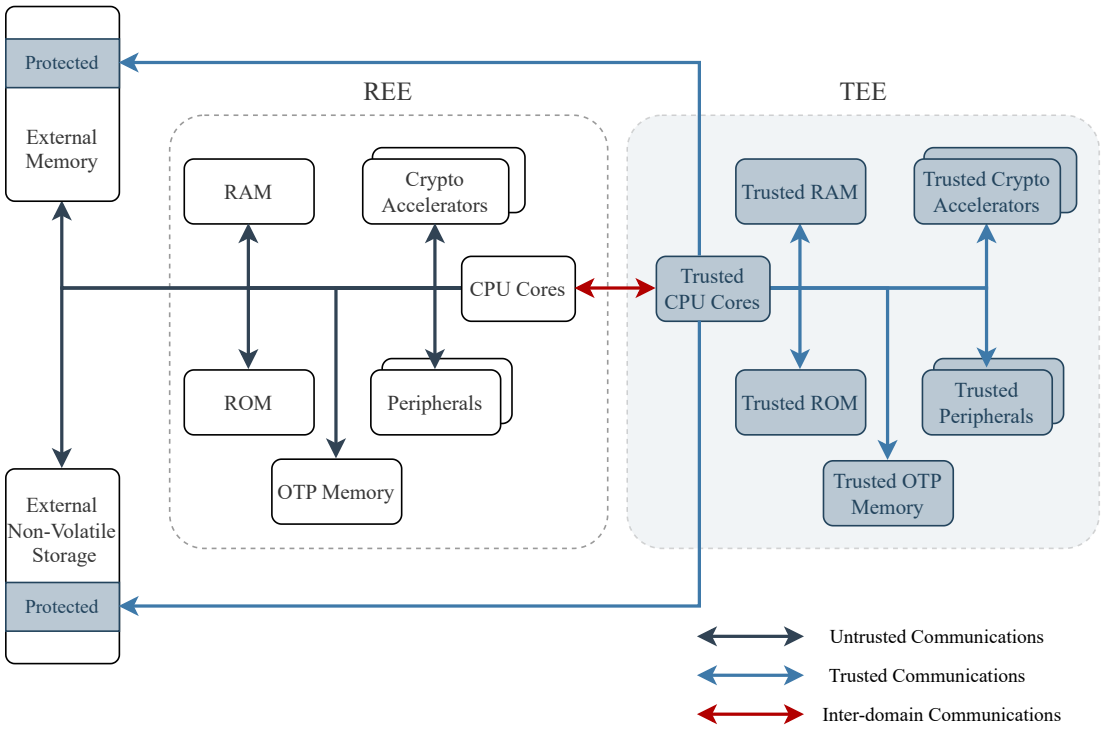


Figure 1: General representation of TEE components and their interactions.

According to the confidential computing consortium [5], TEE is an environment that provides a level of assurance of:

- **Data Integrity:** preventing unauthorized entities from altering data when data is being processed;
- **Data Confidentiality:** unauthorized entities cannot view data while it is in use within the TEE;
- **Code Integrity:** The code in the TEE cannot be replaced or modified by unauthorized entities.

Existing literature surveys various commercially available TEE solutions [1, 6], highlighting shared design decisions and aligning them with common security goals. Based on these insights, the following sections categorize TEE according to four key security properties: secure boot, run-time isolation, trusted I/O, and secure storage.

TEE Secure Boot

Secure boot ensures that the execution environment is correctly configured and that trusted components start in a known and verified state. This process establishes the correctness of the initial state of the TEE. To guarantee the authenticity and integrity of software executed on the device, secure boot relies on a Chain of Trust (CoT). The CoT begins with a Root of Trust (RoT), typically based on asymmetric cryptography. During manufacturing, the vendor generates a unique key pair. The private key is securely

retained by the vendor, while the public key is embedded into read-only memory on the System on Chip (SoC), such as OTP memory or fuses. This embedded public key serves as the hardware RoT for signature verification.

On power-up, the SoC uses the embedded public key to verify the digital signature of the software image. If the verification succeeds, it confirms that the software is authentic and has not been altered. The boot process then continues. Each stage in the boot sequence verifies the integrity and authenticity of the next stage before transferring control.

TEE Run-Time Isolation

TEE Run-Time Isolation protects critical resources, such as the CPU and memory, from potential threats during execution, ensuring the secure operation of TEE components and preventing unauthorized access to sensitive data. To achieve this, TEEs use partitioning strategies enforced by different isolation mechanisms. These strategies are typically categorized as spatial, temporal, or spatio-temporal partitioning. Isolation can be enforced using logical or cryptographic methods.

Spatial Partitioning. Spatial partitioning involves the separation of resources ensuring that the resources allocated to one component are isolated from and inaccessible to other system stack components. For example, in a system running multiple TAs, spatial partitioning ensures that resources allocated to one TA are inaccessible to another, enhancing the security and confidentiality of the TAs (see Figure 2:A).

Temporal Partitioning. Temporal partitioning is the time-based separation of execution. This allows multiple system stack components to share the same physical resources without interference, by assigning them access in distinct time slots. This approach enables the reuse of CPU and memory through controlled scheduling, ensuring that only one component can access the resources at any given time. For example, if multiple TAs running within the TEE in a single core, temporal partitioning ensures that the execution of one TA does not compromise the execution of another (see Figure 2:B).

Spatio-Temporal Partitioning. Spatio-temporal partitioning refers to a mix of both temporal and spatial partitioning, preventing interference not only across different spatial domains but also across different temporal domains. For example, if multiple TAs running within the TEE are assigned distinct memory regions (spatial isolation) while sharing a CPU core through time-sliced execution (temporal isolation), the system ensures that no TA can access another TA's memory or influence its execution, regardless of execution order or resource reuse (see Figure 2:C).

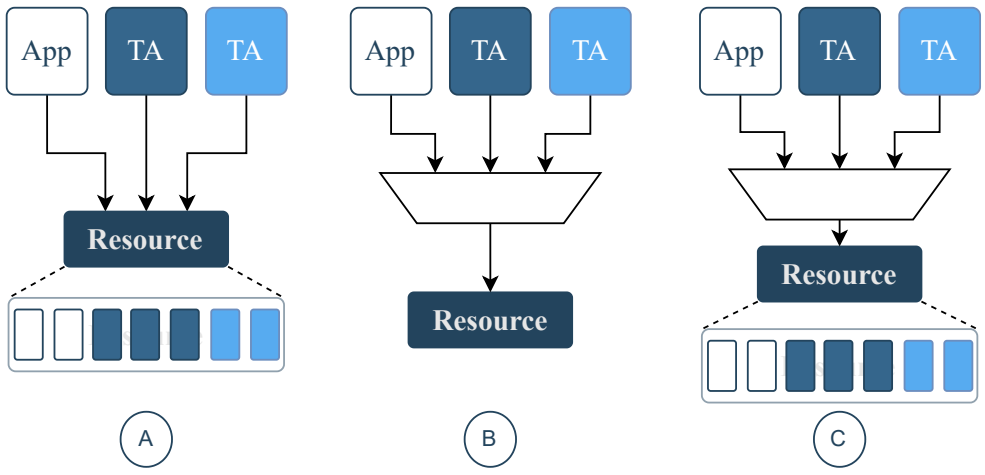


Figure 2: Resource partitioning strategies. A: Spatial Partitioning; B: Temporal Partitioning; C: Spatio-temporal partitioning.

Logical Isolation. Logical isolation refers to the use of system-level mechanisms to prevent malicious entities from accessing or intercepting protected data. Because it involves the handling of sensitive information, this isolation is typically enforced and monitored by a high-privileged component, such as firmware or a hypervisor, that has the authority to manage and reassign system resources. These privileged entities rely on security primitives to enforce strict access control policies and maintain the confidentiality and integrity of protected data (see Figure 3:A).

Cryptographic Isolation. Cryptographic isolation refers to the use of cryptography to only allow authorized entities to access/decrypt the correct content. Unlike logical isolation, where access to protected data is strictly restricted to authorized components (ensuring confidentiality through access control), cryptographic isolation allows unauthorized entities to observe the encrypted content, but prevents them from recovering the original data (ensuring confidentiality through encryption). Typically, TEEs include a cryptographic engine, which can be used to enforce this type of isolation (see Figure 3:B).

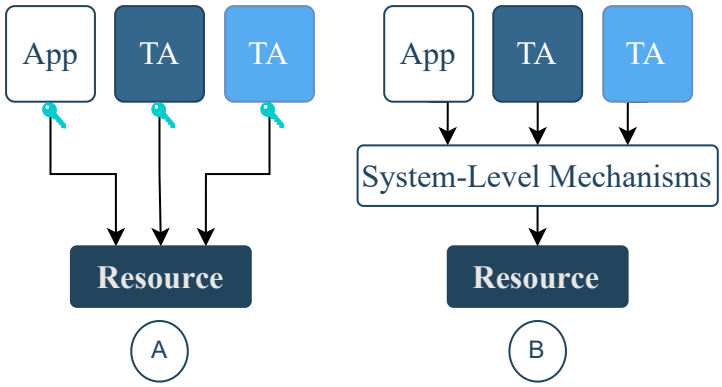


Figure 3: Enforcement mechanism for resource isolation. A: Logical Isolation; B: Cryptographic Isolation;

TEE Trusted I/O

Establishing Trusted I/O in TEEs is critical for safeguarding the integrity and confidentiality of TAs when utilizing peripherals. This involves ensuring secure communication channels between TAs and devices, as well as protecting TA data processed by devices. One approach for secure communication is using logical isolation, which secures Direct Memory Access (DMA) and Memory-Mapped I/O (MMIO) accesses. This can be achieved through access control filters or secure memory mappings, dynamically or statically configured during TEE runtime. Alternatively, a cryptography-based trusted path offers protection against fabric adversaries, involving cryptographic material. Some TEE implementations combine cryptography with logic isolation for robust MMIO access protection.

TEE Secure Storage

To protect security-critical data, TEEs require a secure storage mechanism. The most common approach is sealing and unsealing. Sealing refers to the encryption of data before it is stored persistently, while unsealing refers to its decryption when it is retrieved. This ensures that the data remains confidential and any unauthorized access or tampering is detected through cryptographic verification.

Encryption can be implemented in software or supported by dedicated hardware. Hardware-based encryption offers better performance and resistance to side-channel attacks but comes with higher cost. Encryption can follow either symmetric or asymmetric cryptographic methods. Symmetric encryption uses the same key for both encryption and decryption, while asymmetric encryption uses a pair of keys, one for encryption and another for decryption. Advanced Encryption Standard (AES) is the most widely

used symmetric encryption algorithm, while Rivest-Shamir-Adleman (RSA) is the most commonly used asymmetric algorithm [7].

3.1.2 TEE Models

Different TEE technologies from various vendors adopt distinct design choices and TEE models. This section examines hardware-based security technologies with a focus on solutions developed for Arm, Intel, AMD, and RISC-V architectures.

Arm TrustZone for Cortex-A. Arm TrustZone, introduced in 2004, is a security technology integrated into a wide range of Arm-based processors, that enables the establishment of a secure execution environment [8]. TrustZone aims to protect critical data and code by isolating it from potential threats, whether originating from the OS or hypervisor. In the TrustZone architecture, the system is divided into two distinct execution environments: the "Secure World" and the "Normal World" (or REE). The Secure World serves as an isolated domain, protected from unauthorized access, including high-privileged software components like the OS. Access controls are enforced by the CPU and system-level access controllers called TrustZone controllers. Additionally, TrustZone is often paired with secure boot mechanisms that verify the authenticity and integrity of firmware during system initialization this guarantees the execution of only verified firmware components.

Arm S.EL2/FF-A. Arm TrustZone technology, as implemented in Armv8.4 with the inclusion of the secure hypervisor, offers an advanced security and virtualization solution that extends the capabilities of TrustZone [9]. Armv8.4 extends Arm's security architecture to include a secure hypervisor mode (Secure Exception Level (EL)2 or Exception Level 2). This mode enables the execution of multiple VMs in the Secure World, each isolated from one another and from the Normal World. The secure hypervisor manages these VMs, providing secure isolation and control over their execution. The reference hypervisor is hafnium which does not support dynamic instantiation of VMs, creating them only during boot [10]. In essence, this iteration of TrustZone adds a privilege level to the secure world. The introduction of S-EL2 (Secure Exception Level 2) in Arm v8.4 architecture, lead to Firmware Framework for Arm (FF-A) [11]. When transitioning between exception levels (e.g., from EL1 to S-EL1), each level has a separate binary, requiring agreement on the Application Binary Interface (ABI). FF-A aims to provide a consistent ABI across different trusted VMs and hypervisors, making it easier to reuse certified Trusted Firmware (TF) and hypervisor configurations across various setups.

ArmCCA. Arm Confidential Computing Architecture (CCA) is designed to enhance the security of data and applications by providing isolated environments, known as Realms, where sensitive code and data can be processed and stored securely [12]. This architecture is particularly relevant in the context of cloud computing, edge computing, and IoT, where ensuring the confidentiality and integrity of data and applications is paramount. With the increasing amount of sensitive data being processed and the rising threats to data security, there's a growing need for more robust security solutions. Confidential computing addresses this need by protecting data in use, in addition to data at rest and in transit. The core concept of Arm CCA is "Realms": isolated environment capable of executing arbitrary code. Realms are designed to provide a high level of security for sensitive workloads. They operate separately from the normal OS environment, thereby protecting a wide range of software attacks. CCA features encryption to protect against memory attacks such as bus snooping or cold boot. It uses Granule Page Tables (GPT), a page table-like mechanism, instead of TrustZone Controllers for access control.

Intel Software Guard Extensions. Intel Software Guard Extensions (SGX), released in 2015, are specialized security instructions integrated into select Intel CPUs, enabling the creation of isolated memory regions known as "enclaves" within applications [13]. SGX is designed to protect enclaves from vulnerabilities originating in the OS or hypervisor, or otherwise malicious software, and hardware-level threats such as bus-snooping, or cold-boot attacks. Thus, SGX enclaves serve as secure compartments, protected from external inspection or access, even by high privileged software (e.g., the OS, or Hypervisor),

by CPU-imposed access controls to prevent unauthorized memory access. The integrity of enclaves' memory is guaranteed by a built-in memory encryption engine, ensuring on-the-fly memory encryption and decryption when data moves from CPU to memory or from memory to CPU, respectively. Encryption and decryption are performed with a key inaccessible to any software. Additionally, Intel SGX establishes remote attestation as a foundational security measure, enabling external entities to validate the secure execution of a software application within an enclave on an SGX-enabled platform.

Trust Domain Extensions. Intel Trust Domain Extensions (TDX), released in 2021, introduces hardware-based isolation features for VMs within designated Trusted Domains (TD) [14]. Similar to SGX, TDX's primary objective is to safeguard its isolated environment, specifically TD, against high-privileged software, notably the hypervisor, while also providing defense against hardware-level attacks like bus-snooping and cold-boot exploits. To address the protection against privileged software, TDX introduces a new execution mode, denoted as SEcure-Arbitration Mode (SEAM). SEAM mode hosts the execution of the TDX Module and associated VMs, ensuring their isolation from the broader system software. The TDX Module functions as a secondary, lightweight hypervisor, primarily responsible for defining access control policies, while resource management remains the responsibility of the untrusted hypervisor. In contrast to SGX, where a unified key is utilized for all enclaves, TDX adopts a per-Trusted Domain key model, assigning a distinct encryption key to each trusted domain, thereby enhancing security granularity. Additionally, akin to SGX, TDX incorporates a remote attestation mechanism, enabling the validation of TDX protection to remote third parties.

AMD SEV, SEV-ES, SEV-SNP. AMD Secure Encrypted Virtualization (SEV), released in 2016, protects VMs from security risks in virtualized environments [15]. It ensures the confidentiality of VM data and code, isolates VMs from potentially compromised hypervisors and protects against threats posed by co-located VMs and physical attacks. SEV encrypts the memory of each VM, using one key per VM to isolate guests from the hypervisor. The keys are managed by the AMD Secure Processor (PSP). AMD has extended SEV with improved security features since its release. The first is AMD Secure Encrypted Virtualization-Encrypted State (SEV-ES), an extension of SEV that encrypts all CPU register contents when a VM stops running. This prevents the leakage of information in CPU registers to the hypervisor, and can even detect malicious modifications to a CPU register state. More recently, AMD developed Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP), another extension of SEV that adds memory integrity protection to help prevent malicious hypervisor-based attacks like data replay, and memory re-mapping.

Confidential Virtual Machine Extensions. The RISC-V Confidential VM Extension (CoVE) represents RISC-V's response to the confidential computing UC [16]. Analogous to AMD SEV and Intel TDX, it allows for the execution of VMs shielded from an untrusted hypervisor, offering protection against hardware attacks when coupled with memory encryption, aligning with other confidential computing solutions. CoVE's Application Programming InterfaceApplication Programming Interfaces (APIs) are deliberately designed to accommodate multiple implementations, adaptable to diverse architectural constraints, thereby enabling versatile deployment strategies. A fundamental component of the CoVE architecture is the introduction of the Trusted Security Manager (TSM), operating in the Hypervisor-extended Supervisor mode (HS-mode). The TSM is similar to the TDX Module in that its main responsibility is establishing access control policies on trusted VMs. Notably, when coupled with the Memory Tracking Tables (MTT), CoVE provides fine-grained access control mechanisms similar to Arm's GPT. The MTT, resembling memory page tables, enhances access control by surpassing the limitations of PMP in terms of the number of regions, thereby enabling highly granular and adaptable access control policies.

PMP-based TEEs. The RISC-V architecture features the PMP, a mechanism designed for performing access control over system resources [17]. PMP can be used to establish multiple isolated execution environments, where each environment is restricted in the memory or peripherals it can access, protecting against untrusted software stacks or creating mutually distrusted execution environments in the same platform. PMP operates through control registers, which enable the specification of core access

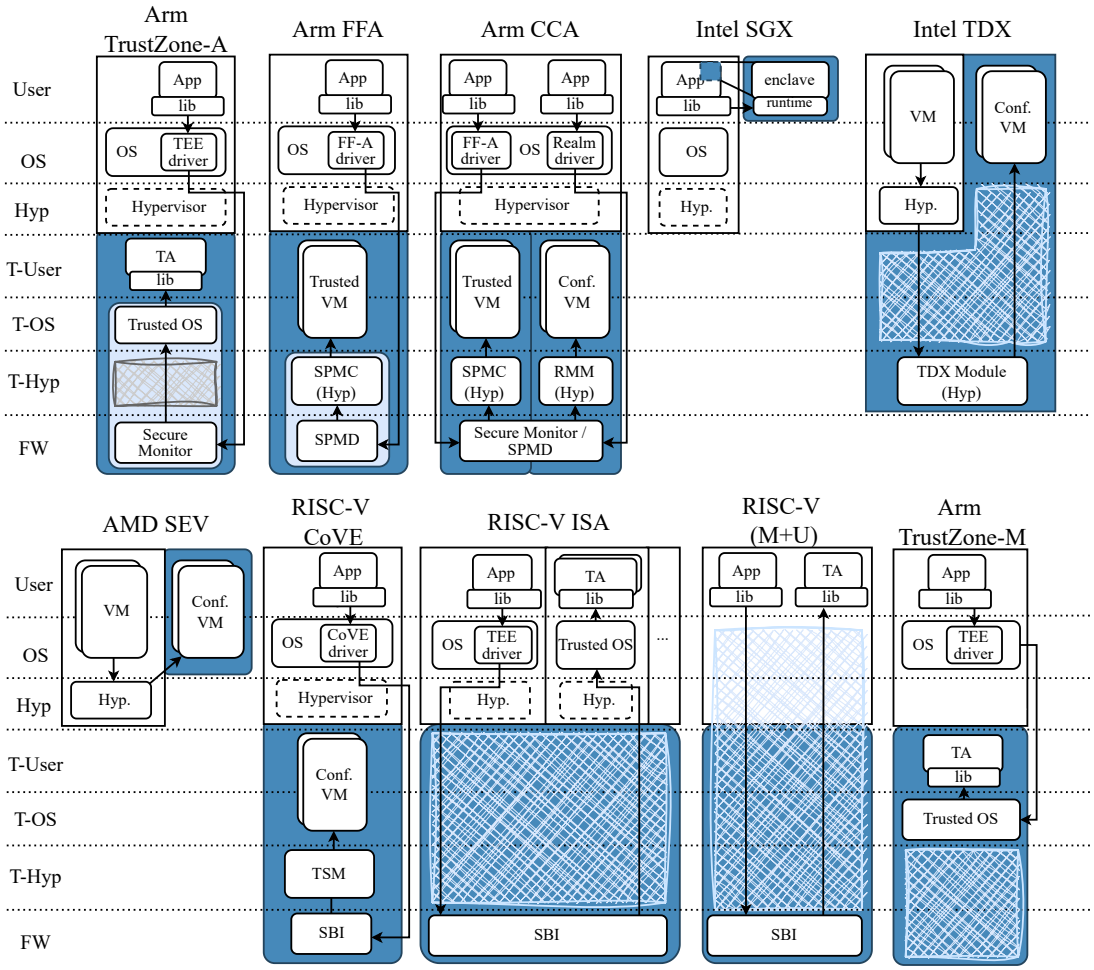


Figure 4: Representation of TEE models across multiple architectures. For APU-class devices: Arm TrustZone-A, FFA, and CCA; Intel SGX and Intel TDX; and RISC-V CoVE and RISC-V ISA extensions. For MCU-class devices: RISC-V(M+U) and Arm TrustZone-M.

permissions. While the number of entries is subject to implementation specifics, it will inevitably be limited compared to solutions based on virtual memory. The RISC-V architecture incorporates a layered privilege model, with the most privileged mode being Machine mode (M-mode). M-mode software is responsible for configuring the access permissions for each core, and if needed, it can dynamically reconfigure the access control policies. However, while PMP enforces access control at the CPU level, specifically per RISC-V hart, it does not extend its protection to other bus masters in the system. This limitation introduces potential security vulnerabilities, as peripheral devices with direct memory access capabilities may remain unregulated. To address this gap, the RISC-V community is actively standardizing the IO Physical Memory Protection (IOPMP) mechanism. IOPMP is designed to mediate and enforce memory access permissions for non-CPU masters, enabling fine-grained control over memory transactions originating from I/O devices.

Arm TrustZone for Cortex-M. Arm TrustZone for Cortex-M[18] (TrustZone-M) is integrated into Armv8-M architecture microcontrollers and offers two orthogonal security states: secure and non-secure. Unlike its TrustZone-A counterpart, TrustZone-M utilizes a memory map approach, i.e., the secure state of the processor is determined by whether the code runs from normal or secure memory. Memory is tagged with attributes that include secure, non-secure, and Non-Secure Callable (NSC), the latter allowing se-

cure entry points within a specific NSC region. In Armv8-M MCUs, memory space is partitioned through Attribution Units. The Security Attribution Unit (SAU) provides dynamic address partitioning, while the Implementation-Defined Attribution Unit (IDAU) provides static partitioning. Memory can also have privilege permissions defined by a TrustZone-aware MPU.

Figure 4 consolidates the TEE models analyzed in this section. It delineates their architectural privileged levels in ascending order, with greater privileges positioned at the bottom. On the top left side, TrustZone-A establishes communication between the OS in the normal world and the TEE in the secure world via a secure monitor. FF-A introduces a trusted hypervisor to oversee trusted VMs. CCAexpands TrustZone to incorporate the Realm world, housing the Realm Manager Monitor (RMM) responsible for creating confidential VMs. SGX safeguards enclaves tied to specific processes from interference by other system software. TDX and CoVE utilize a trusted hypervisor to create confidential VMs, leveraging resources provided by an untrusted hypervisor and managing context switching. SEV allows an untrusted hypervisor to manage confidential VMs, granting the option for full protection at startup or selective memory protection. The RISC-V Instruction Set Architecture (ISA) uses PMP to define security domains, with the Supervisor Binary Interface (SBI) handling context switching between them. In RISC-V microcontroller environments, which operate solely with Machine and User modes, the SBI also leverages PMP to regulate interactions between regular applications and TAs. In parallel, TrustZone-M on Arm MCUs facilitates communication between the normal and secure worlds via the NSC region, a dedicated area that redirects execution into the privileged secure state.

Arm TrustZone-assisted TEEs

Among existing TEE design models, Arm technologies hold a leading position in the IoT market. While TrustZone is also used in server [4] and industrial platforms [19], its widespread adoption in mobile devices [1] and, more recently, in low-power IoT systems makes it particularly relevant for securing embedded and resource-constrained devices [20]. For this reason, we focus primarily on Arm TrustZone technologies, specifically TrustZone-A and TrustZone-M, in our TEE isolation and abstraction implementations. Both architectures follow a dual-world model, consisting of the Secure World and the Normal World. However, they differ in key aspects, including (i) the definition of security states, (ii) exception level structures, (iii) mechanisms for world switching, and (iv) methods for cross-world communication. Figure 4 illustrates the TrustZone-based TEE architecture for Armv7/8-A (TrustZone-A) and Armv8-M (TrustZone-M) processors.

Security States. In Armv7/8-A, the processor's security state is defined by a Non-Secure bit configurable through system registers, which distinguishes whether the system is operating in the normal world or secure world. Conversely, Armv8-M does not use a system register bit for this purpose. Instead, it determines the security state based on the memory region being accessed: executing code from secure memory puts the processor in a secure state, while accessing non-secure memory regions puts it in a non-secure state.

Exception levels. The two architectures differ in their exception levels. In Armv7/8-A, the normal world stack includes EL0 (for user applications), EL1 (for the non-secure OS) and EL2 (for the hypervisor). The secure worlds counterpart includes secure-EL0 (for TAs) and secure-EL1 (for Trusted OS). TAs handle security-critical services like online banking [2] and full-disk encryption [21], while the Trusted OS manages TA scheduling and memory isolation. At the top of the privilege hierarchy is the EL3 , which hosts the secure monitor. Regarding TEE support, the secure monitor is responsible for securely switching between worlds. Armv8-M, in contrast, defines two privilege levels (privileged and unprivileged) within both secure and non-secure states. These privilege levels are independent of processor modes (Thread and Handler modes). Unprivileged levels typically run regular applications, while privileged levels host embedded OSes such as FreeRTOS [22] or Zephyr [23]. On the secure world side, unprivileged levels run TAs, whereas privileged levels execute integrated firmware or MCU-targeted Trusted OSes without separation into distinct ELs, as seen in Armv7/8-A.

World Switching. Both architectures support synchronous world entry, triggered by explicit requests from normal world, and asynchronous entry, driven by secure interrupts or exceptions. In TrustZone-A, synchronous entry into secure worlds is achieved via the Secure Monitor Call (SMC) instruction, which invokes the monitor at EL3 , the sole entry point into the secure world. In contrast, TrustZone-M (TZ-M) eliminates the need for a monitor mode or secure monitor software, significantly reducing world-switch latency and improving transition efficiency [8]. Instead of a centralized entry point, TZ-M introduces NSC regions that offer multiple secure world entry points. For transition between worlds, the following instructions are needed: (i) Secure Gateway (SG) for normal worlds to secure world transitions, (ii) BXNS for returning to normal from the secure world, and (iii) BLXNS for invoking normal functions from the secure world. Asynchronous world switching in both TrustZone-A and TrustZone-M occurs when normal execution is interrupted by secure events, such as IRQs configures as secure. To ensure logical isolation in these transitions, and to ensure that the data used in the trusted world components does not leak to any context of the untrusted world components, the context-switch involves three steps, i.e., (i) saving the current CPU context, (ii) purging the registers used for this transition, and (iii) restoring the next CPU context.

Cross-worlds Communications. In both architectures, communication between worlds is established through either shared memory or message passing. Shared memory allows both the Trusted OS and TAs in the secure worlds to access memory regions also visible to normal worlds, enabling direct data exchange. Trusted OS implementations, such as OP-TEE and mTower, rely heavily on this shared memory model for cross-world communication. Regarding message-passing, it involves transferring parameters via general-purpose registers during a world switch, typically invoked through the SMC instruction.

Interrupt Handling. In TrustZone-A (Armv7/8-A), interrupts are managed through the General Interrupt Controller (GIC), which distinguishes between secure and non-secure interrupt sources through configurable security registers (GICD_IGROUPR<n>), allowing secure interrupts to preempt non-secure ones. The system supports two interrupt models, FIQ and IRQ, with Arm recommending that IRQs be mapped to non-secure sources and FIQs reserved for secure ones. The monitor typically handles secure interrupts, and common scenarios include handling non-secure interrupts during secure execution or automatic world switches when secure interrupts occur during non-secure execution, with context saved to the non-secure stack. In contrast, TrustZone-M uses the Nested Vectored Interrupt Controller (NVIC), which enables fine-grained interrupt isolation at the hardware level without a secure monitor or hypervisor intervention. NVIC introduces Interrupt Target Non-Secure (NVIC_ITNS) registers, accessible only in the secure state, to assign interrupts as Secure or Non-Secure statically. Upon an interrupt, NVIC routes it based on its security attribution: if the security state matches the processor's current mode, standard M-profile handling occurs; if not, such as when a non-secure interrupt arrives during secure execution, the processor automatically saves secure context, clears sensitive registers, and transfers control to the non-secure handler. While secure and non-secure interrupts may share priorities in TZ-M, mechanisms like the PRIS (Priority Secure) bit in the Application Interrupt and Reset Control Register (AIRCR) allow secure interrupts to take precedence during critical operations.

3.1.3 TEE Analysis

As mentioned previously, hardware TEE solutions emerged to provide confidentiality and integrity of security-sensitive applications. Compared to traditional systems, they usually feature a smaller TCB and are thus expected to offer higher security. TEEs have become predominant across several areas, including mobile systems, industry, servers, and low-end devices. Furthermore, in the future, TEEs are expected to be integrated into trillions of IoT devices. However, numerous studies have demonstrated that existing commercial TEE implementations often fall short of their intended security guarantees.

Several studies have continually demonstrated vulnerabilities in TEE systems across many commercial

vendors [20, 24, 25]. One of the most comprehensive analysis cases is in [1], where Cerdeira et al., found 207 TEE bug reports on Arm-based devices assisted by TrustZone-A technology from 2013 until mid-2018. This analysis included commercial TEE implementations from vendors like Qualcomm, Trustonic, Huawei, NVIDIA, and Linaro. **Since that study, many additional vulnerabilities have been disclosed, highlighting the need to reassess the current security landscape. Furthermore, TrustZone-M TEEs for MCU devices have gained adoption, leading to multiple new implementations that also require dedicated analysis to determine whether the types of issues identified in previous analyses are also present in these new TEE implementations.**

To address this gap, we investigate two key questions: (i) how TEE security has evolved over the past five years and (ii) whether recent TrustZone-M deployments apply the lessons from the past. To this end, we (i) reassess the prior work by collecting and analyzing numerous vulnerabilities and limitations affecting TrustZone-assisted TEEs for Cortex-A devices and (ii) extend it to TrustZone-assisted TEEs for Cortex-M MCU devices.

Before delving into the analysis of TrustZone-assisted TEEs, we establish certain assumptions aligned with our project goals. First, our analysis focuses exclusively on vulnerabilities within the TEE software stack, not including any vulnerability present on the REE side. Second, we consider all software components executing in the normal world to be untrusted, and therefore potentially subject to complete adversarial control. We start from the principle that an attacker has the following goals: (i) hijacking trusted components (Trusted OSes or TAs) for privilege escalation or arbitrary code execution or (ii) accessing or corrupting sensitive information.

3.1.3.1 Methodology of Analysis

The process of analyzing TEE implementations entails several challenges:

- **Proprietary Nature and Limited Documentation:** Many TEE implementations are proprietary and lack comprehensive documentation, forcing researchers to rely on reverse engineering.
- **Architectural Heterogeneity:** The TEE ecosystem spans across multiple architectures (e.g., TrustZone-A and TrustZone-M), each with distinct design principles. Analyzing such diverse implementations demands in-depth knowledge of their architectural nuances.

To address these challenges, our analysis builds upon methodology from prior work, now extended to cover a broader set of TEE implementations, including MCU-based TEEs utilizing TrustZone-M. Our methodology consists of five phases: (i) selecting TEE implementations for analysis, (ii) describing data sources of bug reports, (iii) conducting temporal analyses of public disclosed vulnerabilities, (iv) classifying the severity of vulnerabilities, and (v) addressing validity concerns of our analyses.

Selected TEE Implementations

To identify the TEE implementations included in our analysis, we investigated the most prominent and commercially deployed TEE implementations from multiple vendors. Table 3 provides an overview of the selected vendors and their corresponding TEE implementations from ten major vendors (Qualcomm, Samsung, AMD, Google, TrustedFirmware, Trustonic, NVIDIA, Huawei, Tsinglink Cloud, and Ali Cloud), while categorizing them in terms of (i) target hardware (MCU or APU), (ii) availability, and (iii) GlobalPlatform compliance. This categorization helped to identify the wide variety of TZ-assisted TEEs across different vendors, each offering distinct TEE implementations tailored to specific platforms. For example, Qualcomm provides QSEE [26], which has been used in numerous Android smartphones, including those from Samsung, OnePlus, and Xiaomi. Samsung offers TEEGRIS [27], a proprietary and commercially deployed TEE for Samsung Exynos phones, alongside mTower [28], an open-source TEE for MCU devices that is still under active research and development. Google maintains Trusty [29], an open-source TEE for Android devices running on APU processors. TrustedFirmware (TF) project [30] maintains three open-

source solutions: OP-TEE [31], widely adopted for TrustZone development on APU devices, TF-A, and TF-M [32], reference implementations of secure firmware for APU and MCU devices. Trustonic maintains Kinibi and Kinibi-M, proprietary TEEs for APUs [33] and MCUs [34], respectively. NVIDIA supports two Trusted OSes: (i) the proprietary TZVault [35], tailored for safety-critical systems, and (ii) the open-source Trusted Little Kernel (TLK) [36], a solution based on Little Kernel (LK). AMD maintains the PSP[37], a co-processor that provides a secure execution environment by leveraging TrustZone technology and, hence, possesses a proprietary TEE kernel implementation running in APU devices. Huawei maintains proprietary TrustedCore[25] for APU devices. Finally, Tsinglink and Ali Cloud maintain closed-source IoT TEEs, the TinyTEE[38] and Link TEE Air, respectively. Nevertheless, in this selection phase, we also evaluated whether the TEE implementation adhered to GlobalPlatform standards [39]. GlobalPlatform compliance ensures compatibility with a wide range of TAs and devices, making them less prone to vulnerabilities caused by compatibility issues on non-standardized behavior [40]. From Table 3, eight of these implementations are GlobalPlatform-compliant.

Table 3: List of TEE implementations from different vendors, mapped with availability and GlobalPlatform compliance.

Vendor	TEE name	Class	Availability	GP-compliant
Qualcomm	QSEE	APU	Closed-source	✓
Samsung	TEEGRIS	APU	Closed-source	✓
	mTower	MCU	Open-source	✓
Google	Trusty	APU	Open-source	
TrustedFirmware	OP-TEE	APU	Open-source	✓
	TF-A	APU	Open-source	
	TF-M	MCU	Open-source	
Trustonic	Kinibi	APU	Closed-source	✓
	kinibi-M	MCU	Closed-source	
NVIDIA	TZVault	APU	Closed-source	✓
	TLK	APU	Open-source	
Huawei	TrustedCore	APU	Closed-source	✓
AMD	PSP TK	APU	Closed-source	
Tsinglink Cloud	TinyTEE	MCU	Closed-source	✓
Ali Cloud	Link TEE Air	MCU	Closed-source	

Data Sources

After selecting the TEE implementations for analysis, the next step involves gathering and organizing data from various public sources of security reports. We consulted multiple sources, including (i) Common Vulnerabilities and Exposures (CVE) databases such as the National Vulnerability Database (NVD)[41, 42], maintained by the MITRE Corporation; (ii) Security Bulletin (SB)[43, 44, 45, 46, 47]; (iii) Scientific Publications (SP), (iv) Blog Posts (BP), and (v) open-source codebases of TEE implementations. Table 4 consolidates bug reports from CVEs and SPs associated with each TEE vendor. Other sources, such as SBs, BPs, and open-source projects, are not explicitly represented in the table, as the CVE or SP entries typically capture the information they provide. However, because these sources provide additional context and other technical details not found in CVE or SP, they were essential to our analysis.

To consolidate all bug reports, we manually carried out keyword searches on CVE databases using terms related to TEE technologies, such as "TEE", “secure", "TZ", and specific names of commercial TEE implementations. SB from vendors like Samsung [48], NVIDIA [49], and Qualcomm [50] complemented the CVE database with additional information. SP from leading security venues (as shown in Table 4) and various BP supplemented the dataset with in-depth bug report analyses. Furthermore, open-source TEE projects, such as OP-TEE [31] and mTower [28], have enabled low-level inspection and validation of

Table 4: Reported vulnerabilities per vendor, organized by source: CVE databases and Scientific Publications.

Vendor	CVE	SP		Total
Huawei	-	40		302
Qualcomm	63		48	
Samsung	44	28		
TF	23	51		
Trustonic	5	-		
Google	21	-	-	21
AMD	3	-	-	3
NVIDIA	3	-	-	3
Ali Cloud	-	19	-	19
Tsinglink Cloud	-	13	-	13
Others	12	-	-	12
Total	**174**	**199**		**373**

specific issues. Based on the collected data, we analyzed TEE bug reports spanning from 2019 to 2024 to provide an updated perspective on the findings in [1]. While many of these bug reports were directly associated with a particular TEE implementation, others addressed broader vulnerabilities in the underlying TrustZone-M or TrustZone-A architecture, such as misconfigurations or side-channel attacks. Vulnerabilities not clearly assigned to a single TEE implementation were grouped under the "Others" category.

Table 4 presents a quantitative overview of 373 bug reports targeting various TEE vendors, separating them into two columns: 174 publicly disclosed vulnerabilities (the CVEs) and 199 bug reports collected from SP. Among SPs, several have already been submitted to the respective vendors and some have accepted to be published as a CVE. A large number of the bug reports from SP come from research that used fuzzing techniques, e.g., TEEzz [51], PartEmu [52], TEEFuzzer [53], and SyzTrust [54]. While TEEFuzzer and SyzTrust focused on quantifying vulnerabilities within specific TEE implementations, TEEzz and PartEmu extended their scope to multiple vendors. For instance, TEEzz uncovered 40 "crashes" across Qualcomm and Huawei TEEs, whereas PartEmu identified 48 previously unknown bugs across 194 Trusted Applications (TAs) from Qualcomm, Trustonic, Samsung, and Linaro. In addition to these studies, other works such as GlobalConfusion [40], uGlitch [55], BUSted [56], and others [57, 25, 58] have also contributed significantly to this dataset, enriching the landscape of known vulnerabilities across TZ-assisted TEEs.

3.1.3.2 CVE parameters for TEE Analyses

Among all sources of bug reports, CVEs required the most significant manual analysis effort. Each collected CVE was individually examined by analyzing parameters associated with the CVE entry format, which typically follows a consistent structure. Several parameters were particularly relevant for assessing TEE-related vulnerabilities. These include:

- The CVE description, which provides a concise summary of the vulnerability.
- The affected components, specifying the software or hardware impacted.
- The vendor, indicating the maintainer of the affected component.
- The CVE year, showing when the CVE was assigned.

- The Common Weakness Enumeration (CWE) classification, which describes the underlying type of software weakness (e.g., buffer overflow, improper validation).
- The Common Vulnerabilities Scoring System (CVSS) score, which quantifies the severity of the vulnerability based on the CVSS, ranging from 0 to 10.

Temporal Perspective of Disclosed Vulnerabilities

To analyze the evolution of publicly known vulnerabilities in TEEs, we collected CVEs and organized them according to the year of public disclosure and their relevance to either TrustZone-A and TrustZone-M architectures. Figure 5 presents this temporal distribution, spanning from 2019 to 2024.

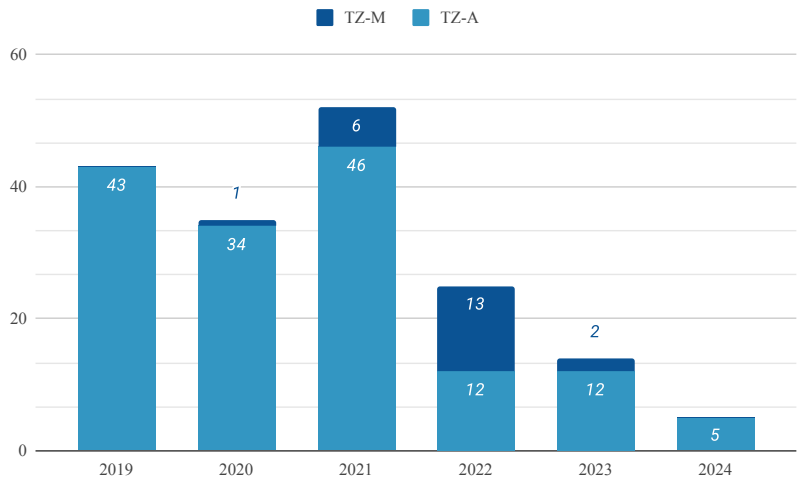


Figure 5: Temporal distribution of CVE reports from 2019 to 2024 in TEE leveraging TZ-A and TZ-M technologies.

The timeline of CVE reports depicted in Figure 5 indicates that, despite the declining trend, the majority of vulnerabilities are linked to TZ-A TEEs, and only a few were reported to TZ-M TEEs. A plausible explanation for this pattern is the relatively recent development of TZ-M technology, combined with the poor adoption of Armv8-M-based MCUs, which contributes to the lower volume of research and, by extension, fewer publicly reported vulnerabilities. Nevertheless, the 199 bug reports uncovered in SP suggest that this trend may reverse in the near future. Specifically for TZ-M TEEs, SP identifies at least 60 vulnerabilities that remain undisclosed in public databases, i.e., 28 in Samsung's mTower, 19 in Ali Cloud's Link TEE Air, and 13 in Tsinglink Cloud's TinyTEE. These findings point to a likely increase in future CVE disclosures.

Severity Classification of Disclosed Vulnerabilities

Following the approach from our previous study, we manually reviewed and classified the CVE reports using the CVSS[59], which provides a numerical representation of the severity of each vulnerability.

Just as in the previous study, the severity classification is split across four severity levels, with the potential impact of a vulnerability depending on its level, i.e., critical (CVSS[9,10]), severe (CVSS[7,9[), medium (CVSS[5,7[), and low (CVSS[0,5[). Accordingly, Table 5 categorizes (i) the severity of vulnerabilities across the selected TEE vendors, (ii) the ratio between TZ-A and TZ-M vulnerabilities, and (iii) a comparison between the number of vulnerabilities per severity involved in this study with those reported six years ago [1].

Unlike in the past, where most vulnerabilities were classified as critical and severe, this time, *severe* and *medium* were the levels with the most vulnerabilities. Specifically, 75 vulnerabilities (42%) are labeled

Table 5: Severity classification of CVE in commercial TEE vendors, grouped by device class. Red arrows indicate worsening trends, while green arrows highlight improvements compared to [1].

Vendor	Critical	Severe	Medium	Low	Total (TZ-A TEE)	Total (TZ-M TEE)
Qualcomm	3	40	20	0	63	-
Samsung	22	18	2	2	31	13
Google	0	3	13	5	21	-
TF	7	9	3	4	15	8
Trustonic	2	0	3	0	5	0
NVIDIA	0	2	1	0	3	-
AMD	0	0	2	1	3	-
Others	1	3	2	5	11	1
Total	**35 (20%)**	**75 (42%)**	**47 (26%)**	**17 (10%)**	**152**	**22**
Total (in [1])	53 ↘	27 ↗	22 ↗	22 ↘	124 ↗	-

as *severe*, while 47 (26%) are categorized as *medium*. This indicated that the number of *critical* vulnerabilities have decreased (green arrow) from 53 to 35 and that the number of *severe* and *medium* have more than doubled-rising (red arrows), i.e., from 27 to 75 for *severe* vulnerabilities and from 22 to 47 for *medium* vulnerabilities. This overall growth is also evident in the total number of vulnerabilities, which increased from 124 to 174. Of these, 152 affect TZ-A TEEs, and 22 are associated with TZ-M TEEs. Samsung stands out with the highest number of *critical* vulnerabilities (22) and also leads in TZ-M-related vulnerabilities (13). On the other end of the spectrum, NVIDIA and AMD report the lowest number of vulnerabilities, with only three each. Although Qualcomm exhibits the highest total number of vulnerabilities (62), and Samsung leads in the *critical* category (21), these numbers should not be used to interpret them as the most insecure TEEs. The data represents only the number of vulnerabilities publicly disclosed and identified in our study, excluding any that remain unreported. Even so, these results underscore the persistent exposure of TZ-based TEEs to security vulnerabilities.

3.1.3.3 Threats to Validity

The entire collection, selection, and classification process of vulnerabilities on TZ-assisted systems was conducted manually, followed by a cross-check by two researchers independently to resolve discrepancies and reach a final agreement. Moreover, due to some impressions in the description of many vulnerabilities and the limited information on proprietary TEEs, the analyses could potentially have some imprecisions. Furthermore, there is a potential risk of over-representing certain vendors, particularly when a specific TEE implementation accounts with a high number of reported vulnerabilities. To mitigate this, conclusions were drawn with caution, aiming to preserve fairness and minimize bias arising from the irregular distribution of vulnerability data. Finally, this study focused primarily on disclosed vulnerabilities, which means the existence of undisclosed issues may further expose security weaknesses in the TZ-assisted systems of another vendors.

3.1.3.4 Classification of TEE Bug Reports

For a fair comparison and we derive classification from our previous work. As result, our analysis categorizes TEE issues into three main classes: (i) architectural, (ii) implementation, and (iii) hardware classes. Within each class, we contextualize each subclass by (i) pointing to specific bug reports, (ii) providing the status update of the TZ-A TEE study [1], and (iii) comparing it to the status of TZ-M TEEs. Note that, in case there are bug reports that do not fit any subclass, we had a new subclass for it. Table 6 lists all classes and subclasses involved in this analyses.

Table 6: Classification of TEE issues by class, subclass, and subsubclass.

Class	Subclass	Subsubclass
Architectural	TEE Attack Surface Issues	I01. SW drivers run in the TEE kernel space I02. Wide interfaces between TEE components I03. Large TEE TCB
	Isolation between NSW and SW	I04. Unrestricted TA Memory Mapping I05. Information leaks to NSW through debugging I06. Weak World Switching
	Memory Protection Issues	I07. Weak ASLR I08. Weak Stack Cookies I09. Excess of Memory Permissions
	Trust Bootstrapping Issues	I10. Lack of software-independent TEE integrity reporting I11. Supported TA revocation I12. TA Authentication Issues
Implementation	Validation Bugs	I13. Validation Bugs Between Worlds I14. Validation Bugs Between Trusted Components I15. Validation Bugs Within Trusted Components
	Functional Bugs	I16. Bugs in Memory Protection I17. Bugs in Security Mechanisms I18. Type Confusion Bugs I19. Unsanitized State
	Extrinsic Bugs	I20. Race Condition Bugs I21. Software Timing Side-Channels
Hardware	Architectural Implications Issues	I22. Attacks through Reconfigurable Hardware Components I23. Attacks through Energy Management Mechanisms
	Microarchitecture Side-Channel Issues	I24. Timing Side-Channel Attacks I25. Long-Term Data Remaining Attacks

3.1.3.5 Architectural Issues

Architectural issues encompass all problems originating from design flaws in TEE stacks. From the collection of disclosed vulnerabilities, we identified 27 CVEs classified under this category, accounting for 15% of the total CVEs. Table 6 presents the possible subclasses that fall under the architectural issue category. Next, we contextualize all architectural issues with their respective subclasses of issues, point to multiple related bug reports, compare them with the previous TEE analysis, and extend the comparison to TEE environments deployed for the MCU class of devices.

TEE Attack Surface Issues. This subclass classifies vulnerabilities that arise when the TEE attack surface becomes too large, potentially compromising the security of trusted components such as TAs, Trusted OSes, and Monitors. Based on the collected bug reports, three main causes were identified: (i) secure world drivers often have excessive privileges when executed at high privilege levels within the Trusted OS, (ii) TAs expose an excessive number of services to the Normal World, while Trusted OSes expose too many services to TAs, and (iii) secure world components tend to have large Trusted Computing Bases (TCBs). *Secure world drivers:* In the TZ-A TEEs, QSEE, TEEGRIS, and OP-TEE continue to employ monolithic kernel architectures, wherein drivers share the same privilege levels as the Trusted OS, thereby inheriting broad privileges. For TZ-M TEEs, the position of drivers in the system stack remains ambiguous, as neither Arm documentation nor current implementations provide clear guidance on assigning drivers to privileged or unprivileged levels. As such, for this analysis, we classify TZ-M architectures as monolithic when drivers and firmware (or Trusted OSes) reside at the same privilege level. *Large Interfaces:* Regarding the issue of large interfaces between trusted components, our findings indicate that the interface surface remains excessively large. This is primarily due to (i) the many unnecessary functionalities implemented within individual TAs and (ii) the number of critical services handled from Trusted OS requested by TAs. Real-world examples include vulnerabilities in Trusty (CVE-2021-34393), QSEE's TAs (CVE-2020-7958), TrustedCore [25], and mTower (CVE-2022-38155). *Large TCB:* Regarding TCB size, although we did not conduct a direct measurement, the existing literature consistently reports

similar concerns. For instance, the TrustedCore TCB was found to be approximately 16 times larger than previously reported in our previous study and the TF-M stands out with a huge codebase, exceeding 117K lines of code.

Isolation between Normal and Secure World. This subclass encompasses vulnerabilities that emerge when trusted components fail to maintain strict isolation between the secure and normal worlds. According to the analyzed bug reports, such design weaknesses manifest in three forms: (i) TAs map memory into the Normal World without proper access control, (ii) the presence of residual debugging information that attackers can exploit to extract sensitive data or bypass security mechanisms, and (iii) insecure world-switching mechanisms that allow the Normal World to interfere with or manipulate the Secure World. *Unrestricted TA Memory Mapping:* Among the analyzed bug reports, some TAs were able to map memory into Normal World unrestrictedly, allowing malicious entities in the normal world to manipulate TAs and gain privileges through a sequence of commands. For instance, a study on Kinibi [43] illustrated how attackers leveraged such mappings to escalate execution to S.EL1 and even S.EL3. Such memory mapping vulnerabilities were not found in TZ-M TEEs. *Information leaks to non-trusted world through debugging:* Regarding debugging-related issues, TAs failed to properly remove debug information, leaving behind sensitive data that could be exploited. A notable example was in Samsung's Widevine TA (CVE-2021-25476), which allowed attackers to bypass Address Space Layout Randomization (ASLR). Similarly, TF-M (CVE-2023-51712) contained a logging function vulnerability that exposed confidential information. *Weak World Switching:* Lastly, deficiencies in context-switching mechanisms of TF-M were also identified, specifically when handling banked stack registers. For example, CVE-2020-16273 describes a case where this led to a potential leak of sensitive data. No vulnerabilities related to weak context-switching mechanisms were found in TrustZone-A TEEs within our dataset.

Memory Protection Issues. This subclass covers vulnerabilities related to memory protection mechanisms that are either absent or poorly implemented in TZ-assisted TEE systems. Common defenses such as ASLR and stack cookies are intended to mitigate memory corruption attacks; however, design flaws or weak implementations often undermine their effectiveness. This category also includes vulnerabilities arising from excessive memory permissions, either due to improper restrictions or improper default permissions. ASLR randomizes the memory location of TAs, making it more difficult for attackers to locate and exploit them. Stack cookies aim to prevent stack-based buffer overflows by placing a known value (a “canary”) between the buffer and control data, thereby detecting attempts to tamper with return addresses. *Weak ASLR* Over the past six years, several TEE implementations, such as QSEE, TEEGRIS, Kinibi [33], TrustedCore, and OP-TEE, have adopted ASLR to prevent memory layout prediction. Nonetheless, weaknesses still existed in past years. In QSEE, a portion of the code was mapped to a fixed address (CVE-2020-3679), annulling the effect of ASLR. Both TEEGRIS [60] and TrustedCore [25] suffered from low entropy, offering only 32,768 and 256 possible memory locations, respectively, for installation. *Weak Stack Cookies:* Regarding stack protection, some platforms (e.g., TEEGRIS, Kinibi, TrustedCore, Trusty, and QSEE) have introduced stack cookies but with vulnerable implementations. For example, Trusty (CVE-2021-34375) and TrustedCore [25] employed static or non-randomized cookie values, which significantly weaken the protection and make bypassing feasible. *Excess of Memory Permissions:* Beyond these protection mechanisms, improper memory access permissions were encountered in TEE implementations over the last six years. In QSEE, inadequate isolation allowed TAs to overwrite memory regions belonging to other components (CVE-2020-11178). Similarly, Trusty misconfigured read-only memory regions with write access, allowing privileged TAs to tamper with core components (CVE-2021-34387). For TZ-M architectures, an optimization in the "small" profile of TF-M’s crypto service removed the encoding of the client ID-key ID pair during storage. This trade-off led to CVE-2021-40327, which allows Normal World components to retrieve Secure World keys.

Trust Bootstrapping Issues. This subclass encompasses architectural vulnerabilities related to the trust bootstrapping process in TrustZone-assisted systems. Trust bootstrapping typically goes from the RoT to a full CoT during the secure boot sequence (see Section 3.1.1). Secure boot ensures the integrity of local

software components, while remote attestation provides assurances about the integrity of those components to external verifiers. *Lack of software-independent TEE integrity reporting:* Current TZ-A TEE implementations lack dedicated hardware support for remote attestation, relying instead on software-based reporting mechanisms [61]. Consequently, the trust of the whole boot relies on the correctness of software components, which are commonly more vulnerable. *TA Authentication Issues:* As the boot process continues, TAs (typically stored in shared Normal World memory) are loaded into Secure World memory. This process typically involves decryption and authentication to verify the legitimacy of the TAs before execution. Nonetheless, some vulnerabilities reported over the past six years reveal weaknesses in this process. For instance, in Samsung's TEEGRIS architecture, the Android Debug Bridge (ADB) interface allowed unverified TAs, including potentially malicious ones, to be installed via a shared folder (CVE-2020-13834). *Supported TA revocation:* Another related process with trust bootstrapping pertains to the update and revocation mechanisms of TAs. While software updates aim to patch vulnerabilities, they may not be sufficient. Secure revocation ensures that deprecated or compromised TAs are not reused and that residual data is properly erased. Our findings show that despite some TEE vendors implementing revocation mechanisms, like Trustonic Kinibi, specific Samsung devices still allow revoked TAs to be loaded and executed in the secure world [43]. In contrast, TZ-M TEE systems exhibit a simpler and more static trust model. No related vulnerabilities have been identified to date. In TF-M, all TAs are statically defined at compile time and directly embedded into the firmware image, resulting in a monolithic architecture that avoids runtime loading and, consequently, many of the dynamic bootstrapping issues observed in TZ-A environments.

Overall, based on our analysis and using our last TEE study as a reference, two general conclusions can be drawn. First, the attack surface of TZ-assisted systems has continued to reveal vulnerabilities over the past six years, leading us to assume that TEE implementations remain notably large. Such bug reports include (i) the integration of kernel-space drivers in TZ-A TEEs, (ii) the monolithic design of both TZ-A/-M implementations, which bundle all components into a single execution environment, and (iii) the extensive interfaces exposed between trusted components and between worlds.

3.1.3.6 Implementation Issues

Implementation issues refer to bugs that arise from unintentional errors made by developers or from misinterpretations of system specifications. In our analysis, such issues account for 144 CVEs, representing approximately 83% of all identified vulnerabilities. Table 7 summarizes the main subclasses of implementation bugs observed across the CVE dataset, along with the corresponding number of vulnerabilities in each category. Notably, validation bugs represent the subclass with higher number of vulnerabilities involve, with a total of 67% of all implementation bugs, however, without an individual distribution. This is because, while some of these bugs are well-documented, the majority are vaguely described, making it difficult to distinguish them accurately. We opted to group them as a whole to ensure a fairer classification.

Table 7: Number of vulnerabilities per implementation issue subcategory.

Class	Subclass	#Bugs	Percentage
Validation Bugs	Validation Bugs Between Worlds	97	67.13%
	Validation Bugs Between Trusted Components		
	Validation Bugs Within Trusted Components		
Functional Bugs	Bugs in Memory Protection	2	1.40%
	Bugs in Security Mechanisms	10	6.99%
	Type Confusion Bugs	16	11.19%
	Unsanitized State	10	6.99%
Extrinsic Bugs	Race Condition Bugs	8	5.59%
	Software Timing Side-Channels	1	0.70%

Validation Bugs: This section addresses implementation bugs resulting from the improper or missing

validation of input values before their use. Unlike previous work, which attributed each bug to a specific TEE component, we instead classify bugs based on the transitions where validation is expected, enabling a unified view across different architectures. This shift is necessary because architectural differences, such as the absence of a Monitor mode in TrustZone-M, prevent a consistent component-based categorization. We, therefore, classify bugs according to three transition categories: (i) between worlds (Normal and Secure), (ii) between trusted components (e.g., TEE and TAs), and (iii) within internal functionalities of trusted components. *Validation Bugs Between Worlds:* In the first case, validation failures commonly occur when data crosses the world boundary, often via message-passing (register-based arguments) or shared memory. Notable examples include Trusty's HDCP service, where buffer lengths were improperly validated across several commands (CVE-2021-34376 through CVE-2021-34379), and OP-TEE (CVE-2019-25052), which allowed excessive memory freeing via improperly validated SMC parameters. Similar issues were identified in Qualcomm TAs (CVE-2023-21627), TF-M (CVE-2021-43619), and Samsung TEEGRIS (CVE-2021-25500). *Validation Bugs Between Trusted Components:* In the second category, OP-TEE exposed flaws when relying on TAs to validate parameters passed to internal memory functions; pseudo-TAs bypassed these checks, resulting in out-of-bounds access (CVE-2019-1010292). *Validation Bugs Within Trusted Components:* Finally, bugs in the third category emerged within the internal logic of trusted components, such as mTower, where TAs were allowed to access adjacent memory regions (CVE-2022-35858) or dereference null pointers due to insufficient checks (CVE-2022-36621, CVE-2022-36622). Trusty TAs also suffered from insecure deserialization logic, leading to buffer overflows (CVE-2021-34394, CVE-2021-34389).

Functional Bugs: This section addresses implementation bugs when the execution of an implemented program operates differently than expected. Unlike in the previous analysis, no peripheral configuration bugs were identified; however, we introduced and included type confusion and unsanitized state bugs. As a result, we assign bug reports into four groups: (i) memory protection bugs, which are related to failures in mechanisms that restrict memory access; (ii) security mechanism bugs, involving flaws in cryptographic or authentication operation; (iii) type confusion bugs, which arise when a program misinterprets the type of a value, pointer, or object; and (iv) unsanitized state bugs, which result from incomplete or repeated dynamic memory stages that cause execution to fail and returning to a clean state (we assume allocation, zeroing, usage, re-zeroing, and free as the dynamic memory stages). *Bugs in memory protection:* Examples of memory protection bugs include an access control flaw in Trusty TLK that allowed users with local privileges to access secure resources (CVE-2021-34395) and a vulnerability in cross-page address verification in OP-TEE (CVE-2019-1010293) that enabled unauthorized memory access. No equivalent vulnerabilities were found in the TZ-M environments. *Bugs in security mechanisms:* Regarding broken security mechanisms, OP-TEE allows cryptographic operations to be invoked out of order, bypassing initialization and causing a crash that could leak information (CVE-2019-25052); Qualcomm lacked authentication in its image verification process (CVE-2019-2338); and Samsung's TEEGRIS was vulnerable to brute-force attacks due to weak authentication (CVE-2020-13835). *Type Confusion Bugs:* Type confusion bugs were primarily caused by misinterpreted values from the normal world, such as in TEEGRIS, where chained commands allowed unauthorized memory access (CVE-2019-20584). As part of the type confusion category, recent research introduced GPCheck [40], an open-source static binary analysis tool designed to assist in reverse engineering and detecting GlobalPlatform (GP) type confusion vulnerabilities. Using GPCheck, researchers identified such vulnerabilities across multiple TEE implementations, including those from Qualcomm, Trustonic, Huawei, and Samsung (e.g., CVE-2021-1923, CVE-2024-20078). In the absence of robust validation mechanisms, all GlobalPlatform-compliant TEE implementations listed in Table 3 may be exposed to similar vulnerabilities. *Unsanitized State:* Unsanitized state bugs appeared in several cases: OP-TEE failed to fully clear TA memory after use, risking residual data exposure (CVE-2019-1010294); Qualcomm experienced a use-after-free due to improper pointer clearing (CVE-2019-2329); and in TF-M, premature memory cleanup during intermediate cryptographic steps led to information leaks and incomplete abort sequences (CVE-2021-32032). Additionally, OP-TEE suffered from a double-free vulnerability during key verification (CVE-2023-41325) caused by non-atomic memory management, which led to repeated freeing of already released regions in the

event of partial failure.

Extrinsic Bugs: This section focuses on implementation bugs resulting from external influences, such as timing or shared resource contention, which we categorize into two types: (i) race condition bugs and (ii) software side-channel issues. This classification replaces the earlier "concurrency issues" category from prior TZ-A analyses, broadening its scope to encompass race conditions in general. *Race Condition Bugs:* Race conditions occur when multiple components access shared resources without proper synchronization, either due to Time-Of-Check-to-Time-Of-Use (TOCTOU) flaws or incorrect management of synchronization mechanisms. TOCTOU vulnerabilities were identified in TEEGRIS, where a double-fetch issue in a TA enabled arbitrary code execution within the TEE (CVE-2019-20610). Similar bugs affected Qualcomm (CVE-2020-11298) and AMD TEEs (CVE-2021-46795), leading to memory corruption. Nevertheless, poor synchronization mechanisms involving misused or missing locks were found in Samsung TEEGRIS (CVE-2021-39647) and Qualcomm QSEE (CVE-2024-32899), causing memory corruption and even unintended reinitialization of the TEE. No equivalent vulnerabilities were reported in TZ-M environments. *Software timing side-channels:* Regarding software side channels, vulnerabilities were linked to the absence of time-constant functions, where execution time varies based on input values. For instance, in Qualcomm QSEE (CVE-2019-10483), attackers could craft inputs and measure execution times to infer sensitive data, such as passwords or cryptographic keys. A similar issue was observed in TZ-M environments through the "Lucky 13" attack (CVE-2020-16150), targeting the Mbed TLS library used in Trusted Firmware-M. This timing-based side-channel attack exploited time differences in how decryption errors were handled in CBC mode, enabling attackers to extract cryptographic keys and compromise encrypted communications progressively.

Overall, based on our analysis and using our last TEE study as a reference, three general conclusions can be drawn. First, in the past six years, validation bugs have been showcased as the most prevalent subclass of implementation issues in TZ-assisted TEEs, accounting for 67% of all implementation flaws. It is a fact that attackers often target transitions as the easiest way to enter a secure world, often by probing for input validation vulnerabilities. Second, type confusion bugs have emerged as a growing concern in TEE security, particularly in GlobalPlatform-compliant TZ-A TEEs [40]. One contributor to this analysis has demonstrated the presence of such vulnerabilities in GlobalPlatform-compliant TEEs on mobile devices, potentially affecting billions of users worldwide and extending those risks to other GlobalPlatform-based TrustZone implementations, including those on TZ-M platforms. Third, there is an ongoing lack of secure coding practices among TEE and TA developers. The continued presence of well-known issues, such as input sanitization flaws, underscores a lack of commitment to secure coding principles. Given the critical security role of TEEs, developers are expected to follow secure development guidelines when building these trusted components.

3.1.3.7 Hardware Issues

Hardware issues refer to flaws that arise from access to hardware components from other than the CPU, such as DMA controllers, peripherals, and accelerators. System-level security can be compromised if security primitives are not provided or correctly configured for these non-CPU masters. We assign these issues to the "Architectural Implication Issues" subclass. Notwithstanding, TEEs also rely on correctly implemented isolation mechanisms at the microarchitectural level, i.e., caches, TLB, and memory controllers could also compromise the integrity and confidentiality offered by a TrustZone-assisted environment. We assign these issues to the "Microarchitecture side-channel issues" subclass. In this study, hardware issues account for around 1% of the total bug reports.

Architectural Implications issues: In this section, we address all issues raised by two different attacks. Attacks originate from reconfigurable hardware components and those targeting energy management mechanisms. Typically, for TZ-assisted TEEs to protect against non-CPU bus master accesses, platform vendors implement filters such as Xilinx Memory Protection Units (XMPU) and Xilinx Peripheral Protection Units (XPPU). Issues related to their improper checking of accesses were placed in this subclass.

Given attacks through energy management mechanisms, they often involve fault injection attacks, commonly achieved through techniques such as frequency regulation or power management, to bypass security mechanisms and thereby leak sensitive information and gain unauthorized privileges. *Attacks through reconfigurable hardware components:* Examples of improper checks of such filters were found in both architectures. In TZ-A TEEs, researchers describe two issues in XMPU on ZU+ MPSoC, where non-CPU bus masters (e.g., accelerators running on FPGA) could bypass filter checks[62]. The first issue involved an improper interpretation of the Master ID by the XMPU, which failed to distinguish between accesses from the CPU and Accelerator Coherency Port (ACP). As a result, FPGA could use ACP to access CPU memory protected by XMPU. The second issue was related to a limitation in the checks done by XMPU, which only filtered accesses to DDR, leaving other memory structures, such as the L2 cache controller, vulnerable. Ultimately, an attacker controlling a malicious accelerator could (i) influence the signature verification process of TAs before their execution within OP-TEE and (ii) retrieve an AES key securely stored. In TZ-M systems, [63] highlights that despite the presence of a Peripheral Access Controller (PAC) in the SAML11-KPH MCU (a device designed specifically for security), its vendors have not considered system-level protections or differentiated privilege levels as part of PAC. As a result, the PAC was ineffective in providing proper security. Consequently, a secure module with access to DMA devices could bypass the Trusted OS security mechanisms (e.g., Kinibi-M), gaining access rights to read, write, and execute. *Attacks through energy management mechanisms:* Regarding attacks through energy management mechanisms, over the past six years, these attacks have been explored in TZ-M systems, particularly using glitching techniques that involve precise power cuts to skip CPU instructions (including world-switching instructions), corrupt memory, or bypass security configurations [55, 57].

Microarchitecture side-channel issues: In this section, we address all issues raised at the microarchitecture level, specifically involving timing side-channel attacks and long-term data remaining attacks. Timing side-channel attacks, as the name suggests, involves utilizing the execution time as a means to compromise system security. Unfortunately, despite microarchitectural designs not being explored extensively in MCU devices (due to their simplicity), a major issue has been identified in the last six years, potentially affecting the entire MCU spectrum. Regarding long-term data remaining attacks, as the name also indicates, it means that the attacker targets non-volatile memory to compromise the system's security. *Timing side-channel attacks:* Examples of timing side-channel attacks were explored by BUSted, an attack developed under CROSSCON acknowledgments, that exploits timing discrepancies in the arbitration logic of the bus interconnect, particularly caused by contention on the bus when multiple masters attempt to access the same slave [56]. Using this contention, we were able to retrieve the secret key of the TA residing on top of TF-M, thereby bypassing all the TrustZone isolation. Later, when tested in compute-intensive tasks, BUSted demonstrates its limitations [64], e.g., when used in cryptographic applications. Since it requires profiling each victim's clock cycle, researchers conclude that the profiling time grows linearly with the increasing number of clock cycles of the victim. In this sense, we extended BUSted with a profiling interrupt-based approach capable of being used even in the presence of cryptographic applications. *Long-term data remaining attacks:* Regarding long-term remaining attacks, an issue example appeared in the past six years[65]. This specific issue relied on the vitality of SRAMs to retrieve critical information. In summary, UnTrustZone applied an "aging" process to accelerate the burning process of the SRAM cells, which typically involves elevated voltage, temperature, and stress time. Such a process made data in SRAM cells remain there, and researchers proved that wiping the memory of burned cells is an ineffective process. This attack targets a list of several devices, the majority of which integrate TEE technologies from both architectures.

Overall, based on our analysis and using our last TEE study as a reference, two general conclusions can be drawn. First, vendor-specific filters of non-CPU bus masters are becoming increasingly integral to TZ-based implementations, enforcing memory partitioning beyond CPU access. However, such technologies are being implemented without insufficient checks, which has led to severe security breaches. Regarding TZ-M TEEs, vendors often fail to consider system-level protections by not integrating them.

Second, Hardware microarchitectural implications need a more in-depth study. The research community has been demonstrated as reluctant to understand the potential security implications of MCUs' microarchitecture. TZ-M presents microarchitectural weaknesses, but related issues remain under-explored. Comparing it to our first study, the number of microarchitectural issues in TZ-A TEEs significantly reduced since our analysis did not reveal information leakage through caches, branch prediction, or row hammer attacks. Instead, the identified vulnerabilities involve side-channel attacks related to timing and long-term remanence.

Our study confirms that widely known security flaws reported in our first study persisted in modern TrustZone-A TEEs in past six years, with some of them also propagated into TrustZone-M systems, exposing similar risks for microcontrollers.

3.1.4 TEE Isolation

Our analysis confirms that existing TEE implementations continue to be victims of numerous vulnerabilities, including some affecting modern designs based on Arm TrustZone-M. This highlights a persistent and unresolved security challenge, particularly relevant given the widespread adoption of TrustZone technology in embedded and IoT systems. This section aims to mitigate the majority of identified issues, particularly those that aim to escalate privileges to the S.EL1 secure-world privilege level in TrustZone for Cortex-A processors [1, 2]. In current implementations, the trusted OS typically runs at S-EL1 and retains unrestricted access to system resources, including those assigned to the high-privilege level EL3.

The TEE isolation mechanism proposed in this section aims to decompose the traditionally monolithic TrustZone architecture into multiple isolated domains. This approach enables the coexistence of multiple trusted OS instances within the secure world and facilitates finer-grained isolation, for instance, by separating VMs from untrusted or third-party TAs. By enforcing stricter boundaries between trusted components, this design aims to limit individual vulnerabilities and improve the overall robustness of TEE-based systems.

3.1.4.1 APU TEE Isolation

Several works have targeted the topic of TEE isolation [66, 3, 67, 2], with the main strategies: i) implementing software-based virtualization techniques in the secure world (e.g., using FF-A on Armv8.4), ii) utilizing existing control units and auxiliary processors [2], and iii) transferring the secure world software stack to the normal world using virtualization techniques [66].

Software-Based Virtualization: Works like TEEv [3] and PrOS [67] leverage software techniques to isolate and create multiple trusted OS environments. This allows for the decomposition of the system's TCB, preventing one single flaw in one environment from affecting the others. Although implementations vary, in this approach, the trusted OS code must be modified to prevent access to security-sensitive functionality such as the configuration of the page tables. There's also a need for well-defined secure entry points that transition the execution from the guests to the hypervisor and vice-versa.

Repurposed Control Units: Works like Rezone [2] uses repurposed control units to decompose Trusted OSes in the secure world. It is common for platforms to feature system-wide control units (e.g., the Peripheral Protection Controller (PPC)) to control access to the system's resources. This control applies not only to the I/O devices but also to the CPU itself. These units can then be leveraged to create isolated environments in the secure world, by reconfiguring the access control policy dynamically during context switches between the trusted OSes and the secure monitor software and by ensuring that the policy is not subject to change by the trusted OS.

Virtualization: Approaches such as vTZ [66], MyTEE [68], and TEEVseL4 [69] leverage normal world virtualization to execute trusted OSes in VMs. This shift allows TEEs to benefit from additional isolation guarantees provided by custom hypervisors enabling more secure deployment models. These solutions

encapsulate trusted OSes inside VMs using different strategies, including: (i) microkernel-based designs that aim to minimize the TCB and support formal verification; (ii) the combination of TrustZone and virtualization extensions to isolate guest TEEs, supported by secure world modules; and (iii) without relying on TrustZone-assisted primitives, relying solely on virtualization hardware. Encapsulating trusted OSes in this way helps overcome several limitations of traditional TrustZone-based systems, such as rigid hardware dependencies and limited scalability.

To maintain maximum compatibility, we decided not to implement software-based virtualization in the secure world. This decision is primarily due to the required modifications to the Trusted OS that such an implementation would require. Instead, **CROSSCON TEE isolation promotes the use of existing control units and advocates for hardware-assisted virtualization in the REE.**

APU TEE Isolation - Repurposed Control Units

To enable TEE isolation using repurposed control units, CROSSCON approach is based on ReZone [2] and thus relies on similar assumptions and mechanisms. ReZone requires that the platform features a PPC mechanism, to control access of bus masters to system resources, and an Auxiliary Control Unit (ACU), a co-processor that can be used to securely reconfigure access to the PPC. In ReZone the leveraged PPC is a platform MPU. For CROSSCON TEE secure world isolation we leverage System Memory-Management Unit (SMMU) as a PPC mechanism. We still require a PPC locking mechanism, which is based on a secure token to authenticate secure monitor code with the ACU. The Trusted OS is configured to be aware of the available memory regions, and the shared memory region is defined at compile time. During a context switch to a trusted OS, the secure monitor reconfigures the PPC, in this case reconfiguring the SMMU page tables, and locks the configuration using the ACU. Conversely, during a context switch from the trusted OS, the secure monitor requests the ACU to unlock access to the PPC, allowing execution to proceed. This approach ensures secure context switching and access control in the system. Figure 6 illustrates the architecture for this solution.

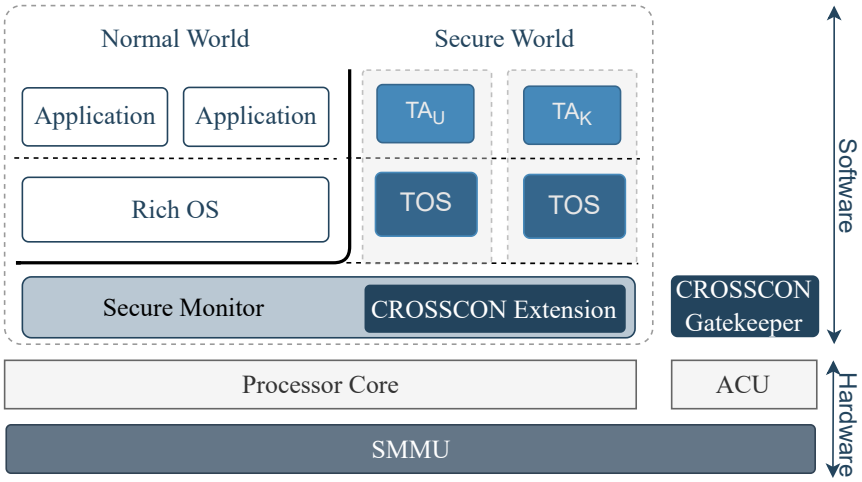


Figure 6: Overview of Trustzone's architecture decomposition on secure world.

Hardware Architecture. Hardware-wise, the system relies on a typical TrustZone-enabled platform for controlling memory access permissions. In addition to a TrustZone Address Space Controller (TZASC) controller, secure world TEE decomposition relies on a PPC hardware component. The PPC is dynamically configured to block secure world accesses from the processor based on the processor's bus master ID (MID_0). The PPC should be reconfigurable by a single bus master, predefined at bootstrapping time. In secure world TEE decomposition, this bus master is the ACU (MID_1). ACU and processor can commu-

nicate with each other efficiently using a message queue implemented by a hardware peripheral.

Software Architecture. Software-wise, secure world TEE decomposition comprises the *secure monitor* and the *gatekeeper*. The former consists of standard secure monitor software (e.g., implemented by Arm Trusted Firmware) augmented with a secure world TEE decomposition-specific sub-component named *trampoline*. The secure monitor (and trampoline) runs on the main processor core and the gatekeeper on the ACU; the PPC protects their private memory regions, which store security-sensitive context information. Taken together, the trampoline and gatekeeper manage the execution of zones in the system. They ensure that each zone can access only a private physical memory address space assigned to the zone, and take care of all context-switching tasks involving zone entering and exiting operations. These operations occur when an REE application makes a call to a zone's TA (*zone entry*), and the TA returns the results of the call (*zone exit*). REE and zone can share data through a shared memory region. Secure world TEE decomposition's software components are shipped with the platform firmware. When the system bootstraps, the firmware configures the memory layout and statically creates one or multiple zones indicating the composition of their respective software stacks, i.e., trusted OS and TAs.

APU TEE Isolation - Virtualization

To enable TEE isolation through virtualization, the CROSSCON approach leverages the isolation guarantees provided by virtualization technologies. It relocates the Trusted OSes from the secure world to the normal world, encapsulating TEE implementations within isolated VMs. This is achieved using the per-VM TEE mechanism built into the CROSSCON Hypervisor. As this functionality is a core part of the hypervisor, its design and implementation are described in detail in the relevant sections. For a complete overview, see Section 3.2.5.2.

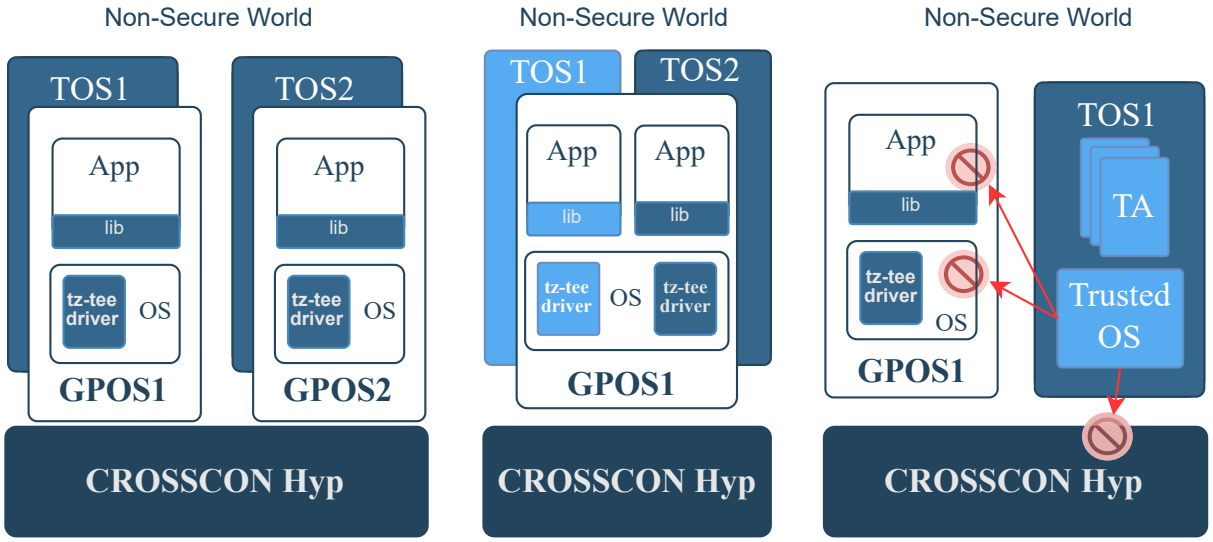


Figure 7: Advantages of per-VM TEE isolation.

The per-VM TEE design offers three key advantages, illustrated in Figure 7. First, it allows two GPOS VMs, running on separate Physical CPU (pCPU)s, to be bound to two independent and isolated TEE VMs. These Trusted OSes can serve different purposes, enabling each GPOS VM to rely on its own dedicated TEE. This configuration reduces the system TCB and improves isolation. Second, it supports the instantiation of multiple TEE VMs to serve a single GPOS VM. In this setup, one Trusted OS instance can be dedicated to handling core system security functions, while others can support application-specific services (e.g., DRM, electronic payments), with each TEE VM tailored to a specific trusted service developer. This approach not only reduces the system TCB but also increases system availability, if one Trusted OS fails, others remain operational. Third, the design enforces strict access control by preventing Trusted OSes running inside TEE VMs from arbitrarily accessing system-wide resources. Unlike traditional TrustZone-A

models, where the Trusted OS often holds excessive privileges, encapsulating it within a VM confines its access scope and reinforces overall system integrity.

3.1.4.2 MCU TEE Isolation

Currently, TEEs in MCUs are only implemented by Arm in Armv8-M CPUs with support for TrustZone-M. A proposal for isolating an unlimited number of equally secure software stacks that make use of Arm TrustZone for Crotex-M devices has been presented by uTango [70].

Isolating MCU TEE workloads in normal world The only proposed approach to improve TEE isolation in MCU TEEs that makes use of Arm TrustZone primitives, suggests to move the TEE software to the normal world and leveraging the secure world to isolated these workloads. This was proposed by uTango. uTango resides in a secure world to perform context switching between multiple software stacks (also treated as a workloads) running in a non-secure world. It enables the secure execution of different workloads by following a configuration file that defines resource allocation and partitioning across workloads and by statically partitioning platform resources at boot time using the SAU and platform-specific bus filters. This ensures memory and I/O isolation across workloads, preventing unauthorized access and avoiding runtime overhead from dynamic reconfiguration. Temporal separation is enforced through round-robin scheduling based on a system timer, with each workload associated with a dedicated context block to maintain its execution state. Context switching includes saving and restoring states, scheduling, and reconfiguring secure regions. Additionally, uTango supports secure inter-workload communication via a message-passing channel, allowing controlled data exchange between isolated software stacks.

CROSSCON TEE isolation on MCU-class platforms depends on features offered by the CROSSCON Hypervisor, which takes the lessons from uTango to perform TEE isolation and implement per-VM TEE feature. Just as per-VM TEE feature implemented for APU devices, the CROSSCON hypervisor allows users to instantiate VMs configured either as regular or as a TEE and associate them to attend services requested by regular VMs.

Similar to the benefits observed in APU-class devices, the per-VM TEE isolation feature in the CROSSCON Hypervisor provides key advantages for resource-constrained environments such as MCUs. As shown in Figure 7, this feature enables (i) the deployment of multiple Real-Time OS (RTOS) VMs, each with access to an independent and isolated TEE VM, which reduces the Trusted Computing Base (TCB) by avoiding shared trusted components, (ii) support for a single RTOS VM to interact with multiple TEE VMs, allowing separation of trusted services based on application or privilege level, and (iii) deprivileging of Trusted OSes by encapsulating each RTOS and TEE instance in a dedicated VM, which prevents access to non-secure workloads and enforces strict isolation.

3.1.4.3 RTU TEE Isolation

Currently, there are no TEE solutions explicitly designed for RTU. However, leveraging real-time virtualization support (based on a dual-stage MPU) allows the instantiation of multiple RTU-TEE instances within VMs, akin to the deployment of trusted OS in normal-world VMs on general application platforms. CROSSCON lays the ground work to implement TEE isolation for Cortex-R architecture. In this sense we intend to extend CROSSCON Hypervisor for Cortex-R architecture and integrate the support for per-VM TEE.

3.1.5 TEE Abstraction

Programming TEE components often presents significant challenges, especially when developing TAs that must interface correctly with multiple heterogeneous and proprietary trusted OSes. TA rely on well-defined APIs to interact with the trusted OS, but inconsistencies in these interfaces and differences

in TEE programming models can significantly hinder TEE developers, requiring them to be proficient in various TEE technologies.

To support TA developers and facilitate TA interoperability with heterogeneous trusted OSes, we initially envisioned a new TEE API specification. However, as discussed in Section 3.1.3, our analysis revealed that eight out of the fifteen analyzed trusted OSes (see Table 3) already conform to a well-defined standard, the GlobalPlatform. Given the widespread adoption of GlobalPlatform across trusted OSes targeting TEEs for both APU and MCU devices, we decided to reuse this standard, and consider extending it if necessary, instead of introducing a new API. As a result, TAs developed under CROSSCON project were developed with the primary goal of being GlobalPlatform-compliant.

In addition to the interoperability challenges faced by TAs across platforms with different TEEs, we observed that interoperability at the TEE model level is also a significant concern. As illustrated in Figure 4, various TEE models differ considerably in their system stack structures and the hardware technologies they rely on. For instance, while TrustZone extends the CPU privilege levels to create isolated execution environments, Intel SGX model operates at the same privilege level introducing additional protections to prevent unauthorized access by system-level software. Furthermore, TEE technologies are widely available across COTS platforms from different vendors, each relying on distinct hardware mechanisms such as Arm TrustZone, Intel SGX, or AMD-SEV. Such diversity results in highly heterogeneous TEE models, which complicates the reuse of trusted OSes across platforms.

To facilitate TEE model interoperability across platforms and consolidate heterogeneous TEE models on a single platform, we extended the CROSSCON Hypervisor to: (i) support the execution of heterogeneous trusted OSes across different ISA, enabling TEE portability; (ii) allow customized TEE implementations that adapt the available security properties to meet specific requirements, enabling TEE compatibility; and (iii) enable the simultaneous coexistence of diverse TEE models.

Solving TA Interoperability

Initially, our goal was to map interfaces between TAs and various Trusted OSes. However, given the widespread adoption of the GlobalPlatform TEE specifications [39] across different TEE implementations, interoperability at the TA level is no longer a major concern. GlobalPlatform APIs focus on the standardization of software interfaces between the TEE OS, TA, and Client Application (CA). These APIs are entirely independent of any underlying hardware architecture or chip manufacturer, which makes them suitable for heterogeneous environments. Furthermore, they currently represent the leading industry standard, with broad adoption by both hardware vendors and software solutions. Most widely used Trusted OSes, such as QSEE, TEEGRIS, mTower, OP-TEE, Kinibi, TZVault, TrustedCore, and TinyTEE, implement the GlobalPlatform APIs, making them inherently compatible with CROSSCON. For these reasons, instead of designing a new abstraction layer from scratch, we adopt the GlobalPlatform APIs as the TEE Abstraction Model in CROSSCON, ensuring compatibility and simplifying integration with existing trusted systems.

Global Platform APIs are divided into two major sets: the TEE client APIs and TEE internal core APIs [71]. Although both of these are important for the interoperability of security services, they define different interfaces. The client APIs regulate the interaction between an client residing in normal world and an arbitrary TA residing in secure world. The core APIs regulate the interaction between TA and Trusted OSes.

TEE Abstraction using a real world application

To demonstrate the GlobalPlatform standard as an effective TEE abstraction layer, we execute an unmodified, real-world GlobalPlatform-compliant TA across multiple GlobalPlatform-compliant Trusted OSes of different architectures. Figure 8 showcases the heterogeneous architectures that we address for running such TA. We demonstrate the TA practicability by running it on (i) Armv7-M and RISC-V(M+U), which

lack critical security hardware primitives, (ii) Armv8-M and (iii) Armv8-A and RISC-V(M+HS+VS+VU). Notably, all TEE system stacks are supported by a CROSSCON component, either the CROSSCON Hypervisor or the CROSSCON bare-metal TEE. These components are essential for enabling the deployment of multiple Trusted OS instances and for supporting a shared, GlobalPlatform-compliant TA across platforms, such as the one used in this case: the Bitcoin Wallet.

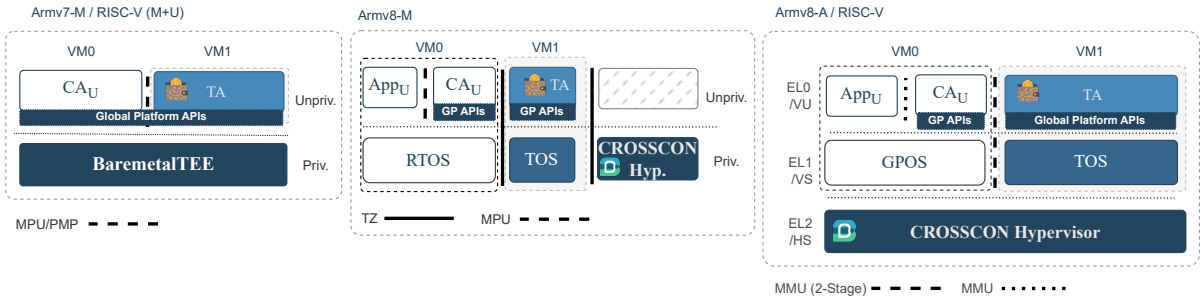


Figure 8: Bitcoin Wallet Compliance in TEE System Architectures on Arm (Armv7-M / Armv8-M) and RISC-V (M/U and M/HS/VS/VU Modes).

The Bitcoin Wallet application is composed of two components: the Client Application (BW CA), which runs in the normal world (i.e., Linux on APU platforms or FreeRTOS on MCU platforms), and the TA (BW TA), which executes in the secure world. The BW CA leverages the TEE Client API to request to TA operations listed in Table 8.

Table 8: Supported commands by the Bitcoin Wallet TA.

Command	Description
`Command 1`	Check whether a master key has already been generated
`Command 2`	Generate a new master key along with its mnemonic (i.e., seed phrase)
`Command 3`	Derive a master key from a provided mnemonic
`Command 4`	Delete the existing master key
`Command 5`	Sign a blockchain transaction
`Command 6`	Retrieve the associated Bitcoin wallet address

Bitcoin Wallet Client Application. The BW CAs are three versions of the same application, developed for Linux, FreeRTOS and Baremetal TEE (see Section 3.5) respectively. It provides a user interface for interacting with the wallet, and it is designed to perform a single operation per invocation. For each invocation, the BW CA opens a session with the BW TA, sends the command along with the necessary parameters, and closes the session once the operation completes. To invoke any of the six supported operations, the BW CA passes the following arguments: the command ID, a 4-digit access PIN, and any optional parameters required for the specific operation (e.g., a mnemonic for master key derivation).

Bitcoin Wallet Trusted Application. The BW TA is developed using GlobalPlatform-compliant interfaces, agnostic to the underlying platform type (MCU or APU). Upon session open/close requests, the TA simply acknowledges success. During invoke-command calls, the TA processes the received inputs, including the command ID, access PIN, and any operation-specific parameters. For some operations, such as command 3, arguments must be passed via shared memory, in this case the mnemonic seed phrase used to derive a new master key.

To validate all Bitcoin Wallet functionalities, the following test sequence was executed:

- Verify whether a master key already exists;
- Generate a new master key using a given mnemonic;
- Sign a transaction;
- Retrieve the Bitcoin receiving address;
- Delete the previously generated master key.

Solving TEE-model Interoperability

In addition to TA interoperability, achieving TEE-model interoperability remains an open challenge in the broader effort toward TEE abstraction. By TEE-model interoperability, we refer to the ability of different TEE models to operate seamlessly across heterogeneous architectures and platforms. Achieving this goal is essential to reduce TEE fragmentation. However, it is hindered by two main factors. First, the significant heterogeneity of Trusted OSes due to the tight coupling between TEE models and the underlying hardware. Trusted OSes are typically designed for a specific TEE implementation tied to a particular vendor. Second, there is a lack of transparency, as most Trusted OSes are proprietary, which complicates cross-platform support and demands deep expertise in each TEE model.

This heterogeneity has driven research into solutions such as emulation and hardware abstraction. Emulation aims to replicate TEE models on platforms that do not natively support them. For instance, Komodo [72] emulates the SGX protection model on Arm platforms, while HyperEnclave [73] brings SGX compatibility to AMD servers. However, these solutions generally fail to support multiple coexisting TEE programming models and restrict the development of novel or customizable TEEs. Alternatively, Keystone [74] adopts a hardware abstraction approach to enable customizable TEEs using the RISC-V PMP. Nevertheless, its dependence on the RISC-V architecture limits portability and does not ensure compatibility with established models, such as Arm TrustZone.

To overcome challenges and adhere to limitations of related work, CROSSCON TEE abstraction offers a virtualization solution that relies on CROSSCON Hypervisor to provide TEE-model interoperability. CROSSCON TEE abstractions provides three main properties: TEE portability, TEE configurability, and the TEE coexistence.

- **TEE Portability:** Enables TEE software stacks to run across diverse platforms and architectures, regardless of native TEE availability.
- **TEE Configurability**: Enables applications to leverage custom TEEs or adapt the security properties of existing ones to suit specific requirements.
- **TEE Coexistence**: Enables simultaneous coexistence of diverse TEE models on a single platform, allowing applications to utilize the most suitable one.

TEE abstraction design

Figure 9 illustrates the system architecture designed to support multiple TEE software stacks, which we refer to as Software-Defined TEE (sdTEE). The architecture consists of four main components: (1) the sdTEEs themselves; (2) the sdTEE configuration file responsible for configuring all sdTEEs (as detailed in D2.3); (3) the CROSSCON Hypervisor, which includes Handlers that manage sdTEE events using core mechanisms as building blocks; and (4) the host platform, which may incorporate its own TEE technology or a RoT.

Software Defined Execution Environment *sdTEE* enables a VM to operate either as a TEE (sdTEE) or as software running in the non-secure domain (typically Software-Defined RTOS (sdRTOS) or Software-Defined GPOS (sdGPOS)), capable of interacting with VMs acting as TEEs according to the defined TEE programming model. Each sdTEE runs as a VM under the *CROSSCON Hypervisor* and includes: (i) multiple

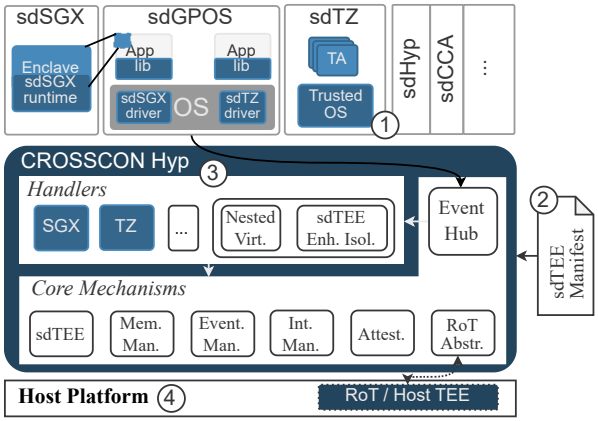


Figure 9: CROSSCON Hypervisor architecture supporting multiple sdTEE models.

Virtual CPU (vCPU), one per pCPU; (ii) corresponding CPU registers; (iii) nested page tables enforcing the sdTEE's access permissions to memory and I/O regions; (iv) a virtual interrupt control state for handling asynchronous events; and (v) an execution priority relative to other sdTEEs. This execution priority is required in scenarios such as when a secure-world, sdTEE, must preempt normal-world execution, sdR-TOS/sdGPOS to handle secure-marked interrupts. In the SGX programming model, the sdTEE running SGX must have lower priority than the sdTEE running the GPOS, allowing the GPOS to retain control over the enclave.

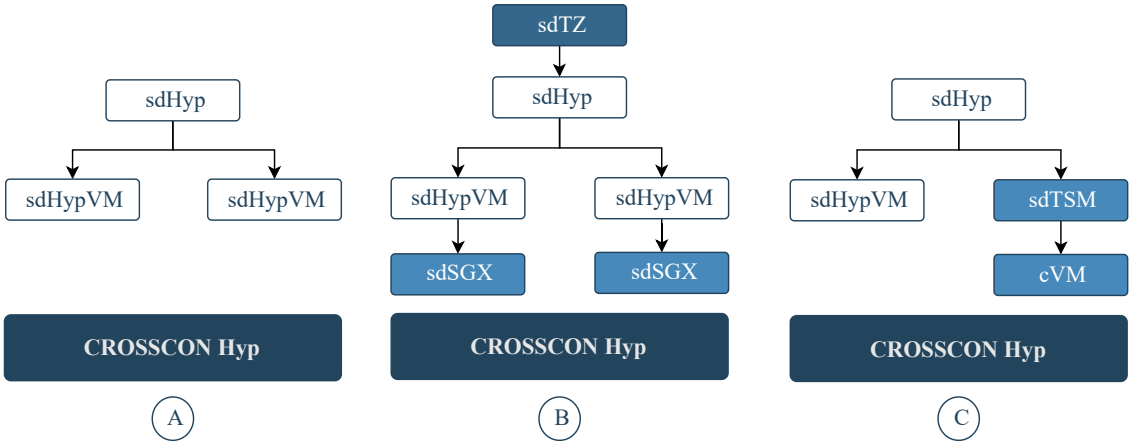


Figure 10: Hierarchical configuration models enabled by CROSSCON Hypervisor

sdTEE Hierarchy The execution priority value establishes a hierarchical model for TEE execution, which can be described using a parent–child relationship, where parent sdTEE control the execution of child sdTEEs. A practical example is found in the TrustZone programming model, where software executing in the secure world acts as the parent (sdTZ) of software running in the normal world. This concept extends to the scenario shown in Figure 10, where *sdSGX* is controlled by *sdSGPOS*, which is in turn controlled by *sdTZ*. Through nested virtualization, the *CROSSCON Hypervisor* also enables the execution of sdTEEs in configurations where a child sdTEE has exclusive access to certain resources that its parent (e.g., *sdTZ*) cannot access. This hierarchical model improves system compartmentalization and security by enforcing structured control and isolation between execution environments. Additionally, more complex configurations are possible, as shown in Figure 10, where all sdTEEs operate as children under the control of a GPOS.

Table 9: Interfaces between CROSSCON Hypervisor core mechanims and sdTEE Handlers.

Module	API	Description	Module	API	Description
sdTEE	`create`	Create an sdTEE	Mem. Man.	`add_mem_region`	Add memory region to an sdTEE dynamically
	`destroy`	Destroy an sdTEE		`rm_mem_region`	Remove memory region from an sdTEE dynamically
Event	`sub_fw`	Register callback on sdTEE firmware calls		`add_dev_region`	Add MMIO region to an sdTEE dynamically
	`sub_hvc`	Register callback on sdTEE hypervisor calls		`rm_dev_region`	Remove MMIO region from an sdTEE dynamically
	`sub_abort`	Register callback on sdTEE access faults		`mem_translate`	Translate an sdTEE addr. to a physical add.
	`sub_mem_region`	Register callback on sdTEE access to region	Attest.	`get_quote`	Obtain an sdTEE attestation quote
	`sub_interrupts`	Register callback on sdTEE interrupt	Exec. Man.	`push`	Add sdTEE vCPU to the execution stack
Int. Man.	`inject_interrupt`	Inject interrupt into sdTEE vCPU		`pop`	Remove sdTEE vCPU from the execution stack

sdTEE Configuration The configuration of each *sdTEE* is integrated into the configuration of the *CROSSCON Hypervisor*'s VMs. For instance, when adopting the TrustZone programming model, two VMs are configured: one acting as the parent (*sdTZ*) and the other as the child (*sdGPOS*). In this case, *sdTZ* is configured with access to the entire memory, MMIO, and interrupts, with the exception of the CROSSCON Hypervisor's private memory.

sdTEE Initialization The *CROSSCON Hypervisor* employs a modular design to emulate TEEs through the use of *sdTEE Handlers*. Each *sdTEE* is associated with a primary handler responsible for delivering interrupts and managing events according to its specific logic. These handlers integrate with the Hypervisor by utilizing its core mechanisms and event-driven architecture. During system initialization, the Hypervisor registers callbacks for events associated with sdTEE instances. When an event occurs, or a request is issued by an sdTEE, the Hypervisor iterates through the registered callbacks, invoking them sequentially to allow each handler to process the event as needed.

sdTEE Runtime Execution Execution is managed using a FIFO-based execution stack, where each element represents an sdTEE context. To initiate the execution of a parent sdTEE, its context is pushed onto the stack. When the parent delegates control to a child, the child's sdTEE context is subsequently pushed. For example, in a TrustZone-based programming model, the context for *sdTZ* is pushed first, followed by sdGPOS. When a trusted service is requested, the CROSSCON Hypervisor pops the active context to return back to *sdTZ*. In the event of an interrupt, the execution stack may need to be unwound to expose the correct context.

APIs for sdTEE Handlers To abstract heterogeneous TEE programming models, the *CROSSCON Hypervisor* provides a set of core mechanisms organized as building blocks. These mechanisms are used by *sdTEE Handlers* to support sdTEE creation, management, attestation, and interaction with the platform's host TEE. Handlers access these mechanisms through a set of primary interfaces, as shown in Table 9.

1. **sdTEE module.** sdTEE module comprises two APIs, i.e., `create` and `destroy`. An sdTEE can be created or destroyed either at boot time or during runtime using the `create` and `destroy` APIs. The create API takes the sdTEE configuration file as input, while the destroy API uses the sdTEE's UUID.

2. **Event module.** The Event module provides five APIs, each prefixed with `sub_*`. These allow handlers to subscribe to sdTEE-related events using the corresponding CROSSCON Hypervisor functions. Supported events include firmware events (which require minimal or no software changes), hypervisor calls (enabling handlers to implement hypercall interfaces), memory access traps (used for I/O emulation), as well as aborts and interrupts. In this latter CROSSON Hypervisor receives all interrupts, and handlers are the responsible for subscribing to them respective handlers need to subscribe to them.

3. **Interrupt management module.** The Interrupt management module provide only one API. De-

spite registering an interrupt to an handler, some cases require handler to inject interrupts into sdTEE, i.e., using `inject_interrupt` API.

4. **Memory Management module.** The memory management module exposes five APIs that enable handlers to enforce spatial isolation and prevent unauthorized access. This isolation is achieved through dynamic modification of access control during runtime. Handlers can add or remove memory regions associated with an sdTEE using `add_mem_region` and `remove_mem_region`. These APIs support both fine- and coarse-grained memory region assignments. Additionally, they allow assigning or revoking access to IO memory regions via `add_dev_region` and `remove_dev_region`. For non-contiguous physical address spaces, the CROSSCON Hypervisor relies on an IOMMU. In platforms without IOMMU support, IO binding is not available. To transfer a resource between sdTEEs, a handler must first obtain its physical address using the `mem_translate` API.
5. **Attestation module.** The attestation module exposes a single API to support sdTEE attestation. This API, get_quote, is used by handlers to generate attestation evidence, which includes the platform and sdTEE code software versions, allocated resources, and a cryptographic hash of the sdTEE payload. Internally, `get_quote` relies on AnyTEE, which utilizes the host platform's TEE or RoT to generate a hardware-bound public/private key pair, securely store the private key, and digitally sign the evidence. The resulting signed evidence list is then returned to the remote verifier, which evaluates its trustworthiness.

3.1.5.1 sdSGX Implementation

In this section, we describe how we implement the SGX TEE model (*sdSGX*) in CROSSCON Hypervisor, allowing the execution of legacy enclaves without modifying architecture-independent enclave source code. Our explanation is based on system architectures illustrated in Figure 11. In this model, we do not fully emulate SGX; we replace its functionalities with calls to *sdSGX Handler*. In the untrusted part, our model uses an untrusted Runtime System (uRTS) to interact with the SGX Kernel driver (at the OS level), which is then responsible for interacting with the *sdSGX Handler*. At the trusted part, we also have the trusted Runtime System (tRTS) to (i) support enclave execution, (ii) interact with the *sdSGX Handler*, (iii) initiate enclave, and (iv) manage invocation to the enclave and calls to untrusted applications.

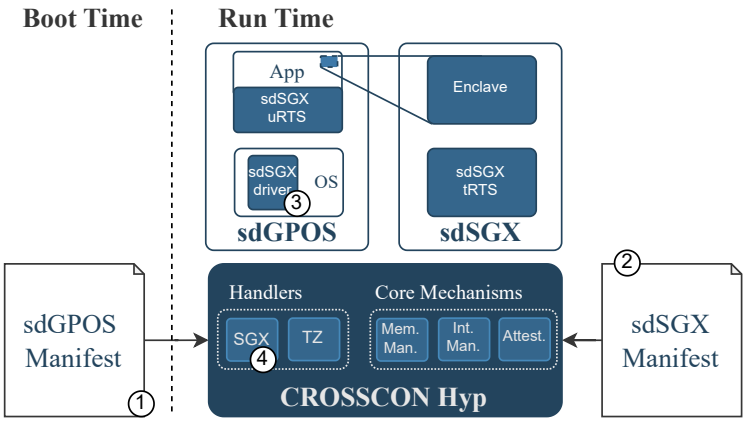


Figure 11: Software defined SGX implementation, including boot and run-time aspects, as well as *sdSGX* software.

Execution Flow. The execution flow is illustrated in Figure 11. First, during boot, the CROSSCON Hypervisor loads the sdTEE configuration file to instantiate the parent untrusted OS (sdGPOS). Second, the *sdSGX* enclave, pre-configured in a dedicated TEE configuration file, is loaded into memory, and the

corresponding enclave memory regions are allocated by the CROSSCON Hypervisor. Third, when an application requests the creation of an enclave, this request is passed to the *sdSGX* driver, which forwards it to the *sdSGX Handler* at CROSSCON Hypervisor. Fourth, the handler authenticates the configuration file, creates the *sdSGX* instance, and initializes it as a child of sdGPOS. Communication between the parent and child is performed using Ecalls, i.e., *sdSGX* specific hypercalls. Since *sdSGX* operates as a child, sdGPOS is responsible for handling enclave exceptions and faults. Within the CROSSCON Hypervisor, the *sdSGX Handler* manages interrupts and memory access faults and notifies sdGPOS when such events occur.

3.1.5.2 sdTZ Implementation

This section describes support for the OP-TEE Trusted OS on RISC-V and how the TrustZone programming model is replicated in *sdTEEs*. To emulate secure world functionality on the RISC-V architecture, OP-TEE is deployed as the Trusted OS. It manages TAs and facilitates communication between the secure and normal worlds. In the untrusted domain, VMs request trusted services through a GPOS that includes a TZ-TEE driver. This driver interfaces with the secure monitor. Unlike the Arm-based *sdTZ* architecture, the RISC-V *sdTZ* introduces key differences. Instead of using SMC, the TEE driver in Linux issues SBI calls to interact with the trusted monitor. These calls are processed by the *sdTZ Handler* within the *CROSSCON Hypervisor*. As in the Arm-based *sdTZ* implementation, the RISC-V *TZ Handler* manages context switching and enforces isolation during world transitions between the untrusted OS and OP-TEE. To ensure secure and exclusive access to memory-mapped I/O (MMIO) devices, the *CROSSCON Hypervisor* maps MMIO regions directly to *sdTZ* and binds the corresponding interrupts to the secure world.

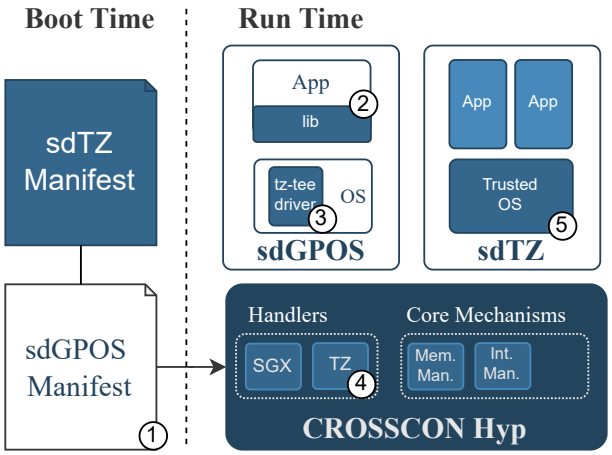


Figure 12: Software defined TrustZone implementation (sdTZ), including boot and run-time aspects, as well as *sdTZ* software.

Execution Flow. The execution flow is illustrated in the system architecture diagram (Figure 12). To emulate the TrustZone programming model, the *CROSSCON Hypervisor* begins by registering *sdTEEs* and parsing their configuration files to allocate resources accordingly. During initialization, the hypervisor pushes *sdTZ* onto the execution stack, then transfers control to *sdGPOS*. In this configuration, *sdGPOS* runs CAs that request the execution of trusted services. These CAs interact with the TEE driver within the untrusted OS, which issues VMM or SBI calls in RISC-V environments to initiate transitions into the secure world. Upon receiving this call, the *sdTZ Handler* intercepts the request, performs the required context switch, and pushes *sdTZ* onto the execution stack to handle secure-side operations.

3.2 CROSSCON Hypervisor

Many IoT applications and embedded OSes anchor their security to the correctness of the TEE. However, not all the applications could and should run inside the platform's provided TEE. Isolation should be extended also to applications outside the TEE in order to protect them from each other. The strong CPU and memory isolation that many platforms already offer is still not enough to guarantee full isolation. Based on our classification of TEEs derived from reported vulnerabilities (section 3.1.3), several critical issues persist. TAs often map memory into the Normal World without adequate restrictions. Developers frequently fail to remove debug information from TAs, leaving sensitive data exposed. Trusted OSes typically handle multiple services from TAs, resulting in excessively large interfaces between components. TrustZone-M TEEs exhibit microarchitectural weaknesses, yet related vulnerabilities remain underexplored. Furthermore, essential hardware resources—such as caches, interconnects, and memory controllers, continue to be shared across partitions, undermining strong isolation guarantees. Existing hypervisors, and container technologies (i.e., microK8) often depend on a large GPOS (typically Linux) either to boot, manage VMs, or provide a myriad of services, such as device emulation or virtual networks. Relying on a complex and monolithic software stack introduces a large TCB, increasing the attack surface and the likelihood of vulnerabilities. As a result, such platforms are unsuitable for hosting security-critical applications that require minimal and verifiable TCBs and strong isolation guarantees.

This section explores the development of the CROSSCON Hypervisor to increase isolation and security guarantees (at the architectural and microarchitectural level). CROSSCON Hypervisor aims to complement TEEs with a micro-kernel-like architecture with a thin Static Partitioning Hypervisor (SPH) layer. However, because SPH lack of flexibility to dynamically create and manage new VMs and services, the main key challenge relies on providing this required dynamicity and per-VM services without enlarging the TCB and lowering the isolation guarantees.

3.2.1 Virtualization and Virtualization Technologies

Virtualization enables the concurrent execution of multiple OSes on a single hardware platform through the use of a hypervisor, analogous to the role of an OS managing processes. The core functionalities of this system include resource management, abstraction, and isolation. Specifically, the hypervisor provides a VM abstraction layer for guest OSes, effectively separating and managing access to hardware resources.

Virtualization is extensively applied across various computing environments. In server settings, it aids in load balancing and power management, optimizing resource utilization and energy efficiency. Desktop applications benefit from cross-platform compatibility and enhanced systems development environments, allowing for seamless operation of multiple OSes on a single physical machine. Within embedded systems and Mixed-Criticality System (MCS), the hypervisor plays a crucial role in isolation, consolidation, and security.

CPU & Memory. CPU Virtualization extensions introduce an additional processor mode for hypervisor operation. The new privilege layer, often called hypervisor mode, sits underneath the pre-existing user and kernel modes. The new privilege level allows guest software to utilize CPU features as intended, facilitated by the replication and banking of system configuration registers across modes. CPU cores are often further enhanced with registers for configuring virtualization features, like selective trapping for sensitive instructions. Virtualization extensions typically introduce two-level translation hardware support. The MMU is enhanced with two levels of translation, translating guest-virtual to guest-physical addresses using guest-managed page tables in the first stage, followed by a translation from guest-physical to host-physical addresses via hypervisor page tables in the second translation stage.

Interrupts. In systems with limited virtualization support, certain challenges arise, particularly in the

context of interrupt handling. The interrupt controller, like other shared devices, must be emulated, necessitating the hypervisor to trap and emulate most, if not all, access to the interrupt system. Furthermore, this emulation process introduces significant interrupt latency since the hypervisor is required to intercept every interrupt before selectively injecting them into the appropriate VM. This step is crucial for maintaining system isolation between VMs but at the cost of increased latency and complexity in interrupt handling.

On the other hand, systems designed with full virtualization support offer more efficient mechanisms for interrupt management. Features such as direct virtual interrupt injection streamline the process, allowing interrupts to be delivered directly to the guest OS without the need for hypervisor intervention. This reduces the overall complexity of the virtualization layer and minimizes interrupt latency, enhancing system performance. Additionally, these systems provide guest interfaces for direct interrupt management, reducing performance costs associated with emulating the interrupt controller. This capability not only simplifies the virtualization architecture but also improves the efficiency and responsiveness of virtualized environments.

I/O Protection. An IOMMU is a hardware component that provides memory management and access control for I/O devices, handling address translation and isolation through virtual memory. Typically an IOMMU is managed by the OS to control which memory addresses a device can read from or write to, thereby enhancing the system's security. In a virtualization context, the IOMMU features a second level of translation tables, similar to virtualization support in the MMU. Another option is to use an IO Memory Protection Unit (IOMPU). An IOMPU focuses specifically on protecting memory for I/O operations, without applying translation tables. This could involve ensuring that devices only access authorized memory regions and protecting against unauthorized or malicious memory access attempts by I/O devices.

In the absence of an IOMMU or IOMPU, SoCs with multiple bus masters, such as DMA-capable devices, crypto-accelerators, and specialized processing units like FPGAs, or Graphic Processing Unit (GPU), face several challenges. In these cases there are no hardware mechanisms for permission checks or access control, exposing the system to buggy device drivers, malicious software, and misbehaving I/O devices. The solution is often to perform device emulation or mediation, however, this incurs performance overheads and increases the TCB, increasing the risk for security vulnerabilities.

Conversely, the incorporation of I/O protection transforms both non-virtualized and virtualized systems significantly. For non-virtualized systems, IOMMU and IOMPU provide memory protection against unauthorized access from other bus masters, with IOMMU further enabling the mapping of contiguous IO Virtual Addresses (IOVA) to fragmented physical addresses. In virtualized environments, IOMMU extends these benefits by facilitating memory protection, efficient virtual address translation for device DMA, sharing of virtual address space between I/O devices and CPUs, and interrupt remapping and virtualization.

Nested Virtualization. Nested virtualization involves running a guest hypervisor within a VM of a host hypervisor. This capability is particularly beneficial for cloud service providers offering Infrastructure as a Service (IaaS) and supports a variety of use cases, including full-stack deployment, mobile app development, testing, validation, and education and training. Essentially, nested virtualization allows for multi-level virtualization, enabling the deployment of VMs within VMs on a cloud platform that itself utilizes virtualization technology.

3.2.1.1 Application Class Virtualization

Application-class processors, commonly known as APUs, are processors tailored for general-purpose tasks such as managing user interfaces and executing various applications on devices like smartphones, tablets, and IoT devices. With the increasing demand for advanced functionalities and multimedia capabilities in modern electronic devices, APUs play a crucial role in enabling feature-rich user experiences

and powering a wide range of applications across diverse industries, including mobile computing, automotive, healthcare, and consumer electronics.

Virtualization in these systems enhances the capabilities of application processors by enabling the efficient and secure execution of multiple virtualized environments on a single physical device, thereby optimizing resource utilization and improving overall system flexibility and scalability. In the Arm architecture, virtualization support is well-established and implemented through hardware extensions, such as the Virtualization Extensions (VE), which are widely adopted across commercial platforms. In contrast, RISC-V introduced virtualization support more recently through the hypervisor extension (H-extension), which adds privileged execution levels (HS-mode and VS-mode) to support hypervisors and guest virtual machines.

In the context of CROSSCON, we focus on Arm and RISC-V architectures due to their growing adoption in embedded and security-critical domains, as well as their respective virtualization capabilities. Arm offers mature and widely deployed virtualization support across a broad range of commercial devices. At the same time, RISC-V provides a free and open ISA that supports custom extensions, enabling tailored implementations. This flexibility makes it suitable for a wide range of applications, from minimalist embedded systems to high-performance platforms.

Arm Virtualization

Arm's architecture is the most prevalent instruction set architecture in mobile devices, and it has also gained significant traction in embedded systems, wearables, and increasingly in server and PC applications [75]. Unlike traditional semiconductor companies, Arm does not manufacture the chips it designs; instead, it licenses its Intellectual Property (IP) to partners who fabricate and sell these chips.

CPU & Memory. Given the widespread proliferation of virtualization in the last decades, Arm implemented hardware support since version 7 of the ISA. The most recent versions of the architecture, i.e., Armv8/9-A, also feature an architecture with a dedicated hypervisor privilege mode (EL2) which sits between the secure firmware mode (EL3) and the kernel/user modes (EL1/EL0) [76] where guests execute. A hypervisor running at EL2 has fine-grained control over which CPU resources are directly accessible by guests (e.g., control registers). Attempted accesses to a denied resource by a guest OS results in a trap to the hypervisor. Additionally, it is possible to route specific guest exceptions and system interrupts to EL2. Other resources that can be managed by the hypervisor include the CPU-private generic timer and the Performance Monitor Unit (PMU). EL1/EL0 memory accesses are subject to a second stage of translation which is in full control of the hypervisor [76]. Any guest access to a memory region not mapped in the second stage of translation will result in a trap to EL2. Arm provides multiple "translation granules", resulting in pages of different sizes: 4 KiB, 16 KiB, and 64 KiB. For each page size, it is also possible to map large contiguous memory regions. These are known as superpages (or hugepages), which reduce TLB pressure. The more commonly used 4KiB granule allows for 1GiB and 2MiB superpages. Arm also defines the SMMU, which extends memory virtualization mechanisms from the CPU to the bus, to restrict VM-originated DMAs.

Interrupts. Arm virtualization acceleration spans the full platform, including the GIC. The GICv2 [77] standard has two main components: a central distributor and a per-core interface. All interrupts are routed first to the distributor, which then forwards them to the interfaces. The distributor allows the configuration of interrupt parameters (e.g., priority, target CPU) and the monitoring of interrupt state, while the interface enables the core management of interrupts. GICv2 provides virtualization support only on the interface; there is a fully virtual interface with which the guests can directly interact without VM exits. The distributor, however, must be fully emulated. Furthermore, all interrupts must first be handled by the hypervisor, which can then inject them into the VM, by writing to GIC list registers (LRs). These registers essentially take the place of the distributor for the virtual interface: when a given interrupt (along with metadata such as priority or state) is present on a register, it is forwarded to the virtual interface. The GICv2 spec limits the number of LRs to a maximum of 16. GICv3 and v4 [78] provide

support for direct delivery of hardware interrupts to VMs; however, this feature is only implemented for Inter-Processor Interrupt (IPI) and Message-Signaled Interrupt (MSI), i.e., interrupts implemented as write operations to special interrupt controller registers and propagated via the system interconnect. Standard wired interrupts, propagated by dedicated signals, are still subject to the mentioned limitation, i.e., hypervisor interrupt injection through the list register.

System Memory Management Unit. At its core, the Arm SMMU provides hardware support for memory address translation, enabling devices to use virtual addresses for memory access, which are then translated to physical addresses by the SMMU. This capability is critical for implementing virtualized systems where multiple VMs share physical hardware resources, enabling peripheral devices to only access the VM memory to which they are assigned to. The SMMU architecture is characterized by several key components. The translation context bank contains context descriptors for various devices, each holding the configuration for the address translation, including the base address of the translation table. These context descriptors enable the SMMU to perform address translations specific to each device, ensuring isolation and security between different VMs or applications. Devices are identified by stream IDs, which are mapped to specific context banks through the stream table. This mapping mechanism allows the SMMU to apply the correct translation context to the memory accesses made by different devices, facilitating device-specific memory management policies and isolation. To detect and respond to various types of access violations or translation faults, the SMMU supports interrupt generation upon fault detection. Often, the SMMU supports multiple stages of address translation, Stage 1 (S1) and Stage 2 (S2). S1 translation is applied first, translating device virtual addresses to Intermediate Intermediate Physical Addresses (IPA), which are then subject to the S2, mapping IPAs to system physical addresses. S1 is managed by the guest OSes. S2 is managed by the hypervisor, allows for separation between VM and address translations, enhancing security and flexibility in memory management. Through the use of translation tables, the SMMU controls access permissions and memory attributes for device accesses, including read/write permissions. This ensures that devices can only access memory regions they are authorized to, with appropriate memory type attributes applied. The evolution of the Arm SMMU architecture, including SMMUv2 [79] and SMMUv3 [80], has introduced enhancements like increased scalability. SMMUv3, introduces features such as fine-grained stream matching, enabling more precise control over which devices are subject to specific translation contexts, as well as memory-based configuration of the translation contexts, which eliminates the restrictions of a limited number of registers.

RISC-V Virtualization

RISC-V [81] is an open-source, royalty-free ISA for designing computer processors gaining significant traction in the last few years. Unlike proprietary ISA s, e.g., Arm and x86, RISC-V is openly available for anyone to use, modify, and implement, allowing companies and developers to innovate and create custom processors without licensing fees. It offers flexibility and customization, making it suitable for various applications, from microcontrollers to data center servers.

CPU & Memory. The RISC-V privileged ISA divides its execution model into 3 privilege levels [82]: (i) M-mode is the most privileged level, hosting the firmware which implements the SBI (e.g., OpenSBI); (ii) Supervisor mode (S-mode) runs Unix type OS that require virtual memory management; (iii) User mode (U-mode) executes userland applications. Although the RISC-V ISA allows the implementation of hypervisors resorting, for example, to classic virtualization techniques (e.g., trap-and-emulation and shadow page tables), such techniques incur a prohibitive performance penalty. Thus, the RISC-V privileged architecture specification introduced hardware support for virtualization through the (optional) Hypervisor extension [82, 83, 84]. The RISC-V Hypervisor extension execution model follows an orthogonal design where the S-mode is modified to a HS-mode well-suited to host both type-1 or type-2 hypervisors. Additionally, two new privileged modes are added and can be leveraged to run the guest OS at Virtual Supervisor mode (VS-mode) and Virtual User mode (VU-mode). The Hypervisor extension also defines a second translation stage (G-stage) to virtualize the guest memory by translating guest physical ad-

dresses into host-physical addresses. The HS-mode operates like S-mode but with additional hypervisor registers and instructions to control the VM execution and G-stage translation. For instance, the hgatp register holds the G-stage root table pointer and translation-specific configuration fields.

Interrupts. The Platform Local Interrupt Controller (PLIC) [85] was the first interrupt controller available for RISC-V architectures, offering a naive solution for interrupt management. The PLIC specification presents several limitations in terms of scalability and features. The global configuration registers are shared across two privilege levels: M-mode and S-mode, and lack support for MSI. MSIs offer significant advantages in the flexibility of interrupt management, by implementing interrupt requests as messages propagated through the system interconnect. The PLIC also lacks virtualization support, resulting in hypervisors needing to rely on techniques like trap-and-emulate decreasing performance [86]. In response to the PLIC limitations, the RISC-V community developed a new interrupt controller specification. The RISC-V Advanced Interrupt Architecture (AIA) [87] is the novel reference specification for interrupt-handling functionality. The AIA consists of (i) the Smaia/Ssaia RISC-V extension to the privilege ISA and (ii) two interrupt controllers, the Advanced PLIC (APLIC) and Incoming Message-Signaled Interrupt Controller (IMSIC). The protection against undesirable accesses is guaranteed at the core level, via the PMP, and at the system level via the IOPMP or RISC-V IOMMU. Virtualization support is offered through the IMSIC, while APLIC offers compatibility with interrupts signaled by wire on devices that do not support MSI [88].

I/O Memory Management Unit. The RISC-V IOMMU defines three methods for managing DMA-capable devices using virtual memory [89, 90]. The first method, device pass-through, permits direct control of a device by a guest OS with minimal hypervisor intervention. Alternatively, a guest OS can share its process address space with devices, which allows guest applications to program the device using IOVA. Lastly, a host OS or hypervisor may choose to retain direct control of a device. The IOMMU may optionally redirect MSI from guest-controlled devices to the corresponding guest interrupt controller. For this purpose, the IOMMU uses the MSI address translation data structures provided by the hypervisor and defined by the RISC-V AIA specification. The RISC-V IOMMU specification incorporates a memory-based mechanism for device and process context management, utilizing hardware-provided unique identifiers (device_id and process_id), device_id are similar to Arm's SMMU stream ID whereas process_id is similar to the SMMU's context banks, making the RISC-V IOMMU most similar to Arm's SMMUv3, and not SMMUv2. It employs a two-stage address translation and a page-based virtual memory system in line with the RISC-V Privileged specification, offering the flexibility to share or allocate distinct page tables for CPU MMU and IOMMU operations. Additionally, it integrates a method for identifying virtual interrupt files and MSI address translations through MSI page tables as delineated by the RISC-V Advanced Interrupt Architecture. The architecture supports both MSI and wire-signaled interrupts for software service requests, enhancing system efficiency and response.

3.2.1.2 Real-Time Class Virtualization

Real-time class processors, commonly known as RTU, are specialized integrated circuits designed to execute tasks with stringent timing requirements in embedded systems. These processors are engineered to provide deterministic and predictable performance, making them suitable for applications where timely response is critical, such as automotive systems, industrial control, and telecommunications.

Similarly to their APUs counterparts, virtualization in the context of real-time class processors, such as Cortex-R processors, primarily serves two purposes: consolidation and isolation. Virtualization allows multiple RTOS or real-time applications to run concurrently on a single RTU, while also providing the means to isolate critical real-time tasks from non-real-time or less critical processes running on the same hardware platform.

CPU & Memory. Virtualization in real-time processors is achieved by introducing a hypervisor privilege mode to the architecture, similar to application class processors. This mode controls access to security-sensitive system registers and resources. Unlike application class processors, real-time processors do not

feature MMU. Instead, they utilize MPUs to establish access control policies for system resources. Some processors can apply an offset to every memory or MMIO access performed by the guest, despite not offering virtual memory capabilities. In the context of Arm, real-time processors implement the Arm real-time architecture. Currently, Arm provides the Armv8-R architecture with optional virtualization extensions [91] for virtualization support. RISC-V is in the process of specifying real-time virtualization, primarily through discussions related to the supervisor Physical Memory Protection (sPMP) [92]. Current work in Work Package (WP)4 is integrating a preliminary version of this specification into a BA5x core.

Interrupts. Interrupt management in real-time virtualization processors closely follows the principles used in application-class virtualization. The interrupt controller is a shared peripheral, and the hypervisor must manage and mediate guest access to it. On Arm real-time platforms, this role is typically fulfilled by a GICv2 controller, such as the GIC 400. For RISC-V platforms, there are currently no commercial real-time platforms available. However, within the context of this project, we consider a soft-core under development as part of WP4 that includes a real-time processor (BA51H). This platform is currently integrating support for the APLIC interrupt controller, targeting this class of processors.

IOMPU. Platforms with real-time processors may or may not include SMMUs. When present, they are likely managed by the APU. However, real-time processors typically feature IOMPUs, which are often vendor-specific. Therefore, real-time hypervisor implementations must provide explicit support for the platform's IOMPU mechanisms. On Arm, IOMPUs vary across vendors. For instance, Xilinx uses the XMPU, NXP uses the Resource Domain Controller (RDC), and Texas Instruments uses Firewall. On RISC-V, working groups are developing the IOPMP specification [93]. Work is underway in WP4 to provide a device access control mechanism for RISC-V called perimeter-guard.

3.2.1.3 MCU Class Virtualization

MCUs are generally focused on simple control tasks, and application processors are characterized by their higher performance, often featuring multiple CPU cores, advanced instruction sets, and specialized hardware accelerators. Typically, they do not directly provide virtualization mechanisms.

CPU & Memory. MCUs typically lack hardware support for virtualization, such as a dedicated hypervisor execution mode or virtual memory capabilities. Nevertheless, several technologies enable the use of techniques that mitigate these limitations [94, 95, 96, 97]. This section focuses on platforms that support Arm TrustZone-M, specifically Armv8-M devices, which introduce a higher-privilege secure mode which we leverage to host a hypervisor. When the hypervisor executes in secure mode, it can control the execution of software in the normal world, including access to system resources and interrupt handling [96]. In this scenario, execution enters in hypervisor space when secure world interrupts, explicit invocations, or exceptions are triggered. To preserve memory isolation, the guest's MPU configuration must be saved and restored during context switches. Other MCU architectures, such as Armv6-M, Armv7-M, and RISC-V platforms that provide only machine and user modes, are outside the scope of this section. These platforms target more resource-constrained devices and are addressed in Task T3.5, which focuses on the development of bare-metal TEE.

Interrupts. Interrupt management in these platforms also needs to be overseen by the hypervisor. Specifically, this involves performing a context switch of the interrupt controller configuration itself. This means that whenever a guest is scheduled to run, its configuration of the Nested Vectored Interrupt Controller (NVIC) is restored, while also protecting the NVIC state of the other guests. This way each guest has exclusive access to its own interrupts and their state is securely maintain while the guest is not running.

I/O Control. Flexible I/O control may be limited in these systems, making DMA operations challenging to handle transparently. In Armv8-M platforms featuring TrustZone-M, I/O protection is achieved through vendor-specific controllers [56]. These units control devices' access to normal and secure world mem-

ory. A straightforward approach to solve this IO control limitation might involve reprogramming this protection units according to the currently executing guest. However, this could lead to violations of the intended access control policy for outstanding DMA operations after a context switch. Therefore, the most effective way to perform I/O control on these platforms is to implement trap-and-emulate techniques or a front-end back-end driver model. In both solutions the hypervisor mediates access to the DMA device and sanitizes the DMA configuration before configuring the DMA device, ensuring that all DMA operations adhere to the established access control policies.

3.2.2 Microarchitecture Isolation Techniques

A system's microarchitecture includes elements like the memory system, interconnects, and CPU design. Since modern processors prioritize performance, security considerations at the microarchitecture level are often overlooked due to their potential impact on performance. However, overlooking security at this level can leave systems vulnerable to various attacks, including side-channel attacks and speculative execution vulnerabilities.

3.2.2.1 Attacks

Microarchitectural attacks leverage the optimizations of microarchitectural components, exposing vulnerabilities in cryptographic computations, general-purpose computations, and the kernel. The leakage of sensitive information persists across common isolation boundaries, including processes, containers, and VMs. This section was written mostly based on existing surveys [98, 99, 100, 101, 56].

Cache. Cache attacks, particularly cache timing attacks, have primarily targeted cryptographic algorithms. Recent studies have identified three common cache attack techniques, which are agnostic to specific cache and hardware configurations: Evict+Time, Prime+Probe, and Flush+Reload. Evict+Time involves measuring how the execution time of an algorithm changes when a chosen cache set is evicted. Prime+Probe assesses whether a victim computation affects the access time to every cache way within a selected cache set. Flush+Reload entails flushing a shared memory location from the cache and then measuring the time it takes to re-access it.

Branch Prediction. The branch prediction functional unit leverages its own caches to store the branch-pattern table storing historical branch outcomes and the branch-target buffer storing past branch targets. Attackers targeting the branch prediction unit, prime the branch-target buffer by executing a sequence of branches. If the victim encounters a branch misprediction, it leads to the replacement of an entry in the branch-target buffer. Subsequently, the attacker observes an increased execution time due to a misprediction in one of its branches.

Speculative Execution. Processors engage in speculative fetching and execution, executing instructions before confirming the accuracy of predictions, with the ability to retract instructions in the event of a misprediction. Recent vulnerabilities such as Spectre [102] and Meltdown [103] have demonstrated security risks associated with speculative execution, by exploiting it to manipulate the processor cache state. Due to the inadequate cleanup of processor cache state in contemporary processors following misspeculation detection, a cache timing attack can be employed to extract sensitive information as a result of cache state modification during speculative execution.

Interconnect. Essential architectural components like the CPU, memory, and peripherals need to be interconnected. This connection is typically facilitated through a central interconnect, often referred to as a bus matrix. When a bus master broadcasts an address, the bus matrix establishes a communication channel between the main and secondary components. In situations where there are simultaneous accesses, the bus can concurrently execute multiple non-blocking full-bandwidth transfers between various bus masters and secondary ports. However, if two data transfers are directed to the same bus secondary, the bus arbitration policy determines the specific order in which the transfers are executed, causing a delay in accesses. The arbitration policy may result in leaking information by enabling an at-

tacker to detect delayed accesses, which can be used to infer the application's internal state. During CROSSCON development we've identified a novel instance of this attack in MCUs, which was published in IEEE Security & Privacy 2023 [56]. Such vulnerability was also counted as part of our TEE analyse in Section 3.1.3.

3.2.2.2 Countermeasures Available to the Hypervisor

The countermeasures used to prevent microarchitectural attacks are crucial for enhancing the security and resilience of modern computing systems.

Manipulation of timing sources. Microarchitectural attacks commonly rely on precise timing measurements. In modern cloud environments, each VM possesses its own timing offsets, encompassing low-level timers such as cycle counter registers. However, microarchitectural attacks appear to be largely unaffected by these variations.

Disabling cache-line sharing and shared memory. Disabling resource sharing can be implemented at various levels for different resources with distinct granularities. In the case of the Last-Level Cache (LLC), which is usually physically-indexed and physically-tagged, cache lines can only be shared among processes if they belong to a shared memory region. Adopting this approach leads to a substantial increase in memory utilization and results in longer execution times due to elevated cache miss rates.

Avoiding cache-set sharing. To mitigate cache-set sharing, cache-coloring has been proposed. This involves allocating cache colors, i.e., sets, to specific VMs when applied by a hypervisor. Additionally, one color can be reserved for the hypervisor itself.

Cache cleansing. Cache cleansing is employed to address the challenge of leakage that persists in the cache after a victim has been scheduled out, assuming that the attacker and victim cannot access any cache set simultaneously. The objective of cache cleansing is to maintain the cache in a state that reveals no information, thus preventing cache attacks. However, with the rise of multi-core processors, the practical relevance of cache cleansing has diminished. Disabling hyperthreading may be a feasible option, but disabling multi-core or the LLC is not practical. Even without the LLC, coherency protocols can maintain cache line coherence across processors and reintroduce timing differences that were thought to be eliminated.

Branch predictor cleansing. To address branch predictor-based channels beyond processor caches, a proposed solution involves clearing the branch predictor on a context switch. Regularly resetting the predictor state ensures that current predictions are not influenced by past inputs, thus minimizing information leakage. However, it's important to note that this defense mechanism comes at a cost to performance since branch predictors depend on learning the branching history of running programs to achieve a high hit rate.

Detecting Attacks. Continuous monitoring software has been proposed as a vigilant measure against malicious activities within a system, actively identifying and halting potential threats posed by attacking processes or VMs. Various detection approaches have been suggested: using performance counters to discern abnormal cache behavior for example through the incorporation of unsupervised learning, or monitoring performance variations in a program simulating a typical victim application.

3.2.2.3 Crosscon-Specific Isolation Mechanisms

CROSSCON prioritizes the identification and analysis of novel side-channel vulnerabilities, as demonstrated by the BUSTED study[56], rather than implementing defenses for previously known attacks. Nonetheless, inherits some isolation mechanisms from its foundation, the Bao Hypervisor, such as cache-coloring mechanisms. These mechanisms effectively mitigate a subset of known cache-based attacks. As a result, the CROSSCON hypervisor implements microarchitectural isolation mechanisms to enforce strong separation between VMs, particularly in MCS with both static and dynamic workloads.

These mechanisms adapt previously discussed countermeasures to meet the constraints of embedded and secure virtualization environments.

Cache Coloring for Static Partitioning and Per-VM TEE. The CROSSCON hypervisor uses cache coloring as a hardware-agnostic method to prevent cache set contention across VMs. This mechanism is applied in statically defined workloads, where VMs are configured at boot time with fixed memory allocations. In this context, CROSSCON assigns non-overlapping cache sets (colors) to each VM, including VMs configured as TEE in per-VM TEE feature (see Section 3.2.5.2 for more details about this feature). This ensures that cache usage remains isolated between domains without requiring hardware modifications.

Cache Cleansing for Dynamic VM Features. In dynamic scenarios, where VMs are instantiated at runtime, potentially as child VMs inheriting memory and context from parent VMs, static cache partitioning is not feasible. To address this, CROSSCON applies cache cleansing during context switches or after dynamic VM operations (see dynamic VM feature details in Section 3.2.5.1). This step removes residual microarchitectural state from the cache, preventing leakage between VMs that may share cache lines due to dynamic allocation.

3.2.3 Hypervisors Feature Analysis and Selection

To develop the CROSSCON Hypervisor we have used a thin static SPH as a starting point. A SPH provides strong isolation between different partitions. This isolation is not just limited to the architectural level, isolating VMs from each other, but extends to the microarchitectural level as well. CROSSCON Hypervisor achieves this by implementing mechanisms like cache coloring, which ensures isolation for the LLC.

The use of a thin SPH layer also helps in maintaining a minimal TCB, which is crucial for upholding high-security guarantees. While existing technologies often rely on a large general-purpose OS or hypervisor, CROSSCON's approach can be applied to a broader range of devices. However, this approach does come with its own set of challenges, such as the lack of flexibility to dynamically create and manage new VMs and services. These challenges will be addressed by enhancing the static partitioning design of the selected hypervisor.

3.2.3.1 Static Partitioning Virtualization

Static partitioning is the practice of, either at a build or initialization time, distributing all platform resources to different subsystems. This can be materialized in many shapes and forms, depending on the hardware primitives. Virtualization is a natural enabler for the static partitioning architecture, due to the strong encapsulation guarantees and flexible resource assignment. Hypervisors designed for the static partitioning UC (or providing such a configuration) have three fundamental properties: (i) exclusive assignment of vCPUs to pCPUs (i.e., no scheduler); (ii) static allocation, assignment, and mapping of all hypervisor and VM memory at build or initialization time; and (iii) direct assignment of devices to VMs (passthrough) and exclusive allocation of their interrupts to the same VM. To implement this efficiently, these hypervisors are highly dependent on virtualization hardware support both at the CPU and platform level (e.g., SMMU). SPH also has non-functional requirements centered around minimizing interrupt latency and inter-VM interference. Thus, over the past few years, there have been efforts to enhance SPH with mechanisms to address these requirements. These include cache coloring and, analogously to what has been done for x86 [104], direct injection in Arm processors. Furthermore, the code base needs to be minimal and follow industry coding standards (e.g., MISRA); this eases Functional Safety (FuSa) certification efforts.

Cache Coloring. In SPH, VMs still share microarchitectural resources such as the LLC. The behavior and memory access pattern of one VM might result in the eviction of another VM's cache lines, impacting the latter's hit rate and consequently its execution time. Thus, there is the need to partition shared caches, assigning each partition to a different VM. While in the past Armv7 processors provided hard-

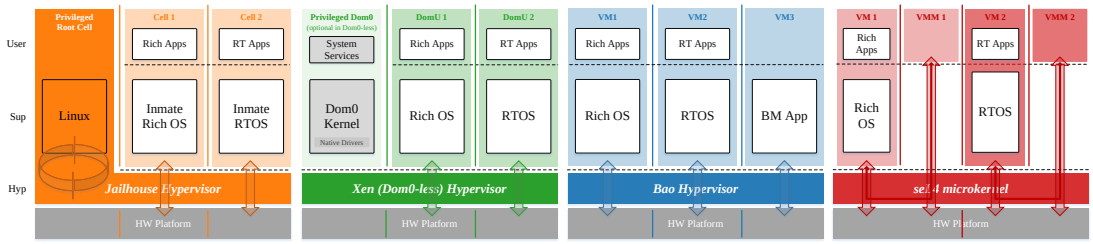


Figure 13: Architectural overview of the assessed hypervisors: Jailhouse, Xen (Dom0-less), Bao and seL4 CAmkES VMM.

ware means to apply this partitioning by way of per-master cache-locking, modern-day Arm CPUs do not provide those facilities. A solution is cache coloring, a software technique for index-based cache partitioning [105]. Cache coloring explores the intersection of the virtual addresses' cache index and page number when creating virtual-to-physical memory mappings. Each color is a specific bit pattern in this intersection that maps only to specific cache sets. Thus, hypervisors can control which cache sets are assigned to a given VM by selecting which physical pages are mapped to it. By exclusively assigning a cache partition (i.e., group of cache sets or colors) to a given VM, cache coloring fully eliminates the conflict misses resulting from inter-VM contention. Cache coloring can also be applied to the hypervisor itself by assigning it one or more specific colors.

Direct Interrupt Injection. Direct interrupt injection is a new technique implemented in Arm-based SPH to eliminate the need for the hypervisor mediating interrupt injection. With this technique, the hypervisor passes through the physical GIC CPU interface and routes all interrupts directly to the VM by configuring the CPU to trigger interrupt traps directly at EL1, i.e., kernel mode. The hypervisor must still emulate the shared distributor to ensure isolation between VMs, i.e., prevent misconfiguration of a given VM's interrupts by another VM. This allows physical interrupts to be directly delivered to the VM with no hypervisor intervention, reducing latency to native execution levels. The forfeiting of interrupts should not be a major issue as SPH does not directly manage devices. However, SPH still needs to communicate internally using IPI. Direct interrupt injection implementations address this issue by leveraging standard Software-Delegated Exception Interface (SDEI) [106] events instead of directly using IPI. SDEI is implemented by firmware, allowing the hypervisor to register an event during initialization. The hypervisor can then trigger the event by issuing a system call to firmware (via a secure monitor call instruction, SMC), which will result in diverting execution to a predefined hypervisor handler, similar to Unix signals. In reality, firmware maps these events to its own secure reserved IPI since, as part of TrustZone, the GIC provides further facilities to reserve interrupts to EL3 .

3.2.3.2 Static Partitioning Hypervisors

Multiple SPHs are currently available, developed by both academia and industry. This section highlights the main features of the currently available works (depicted in Figure 13).

Jailhouse Hypervisor. Jailhouse [107, 108] is an open-source hypervisor developed by Siemens. Unlike traditional baremetal hypervisors, Jailhouse leverages the Linux kernel to boot and initialize the system and uses a kernel module to install the hypervisor. Once Jailhouse is activated, it runs as a baremetal component, taking full control over the hardware. Jailhouse has no scheduler and only leverages the ISA virtualization primitives to partition hardware resources across multiple isolated domains, a.k.a. "cells". Guest OSes or baremetal applications running inside cells are called "inmates". The mainline includes support for x86 and Armv7/8-A, and a work-in-progress RISC-V port[109]. The research community has been actively contributing with mechanisms to enhance predictability, namely: cache coloring, DRAM bank partitioning [110], memory throttling, and device Quality of Service (QoS) regulation [111]. An unofficial fork including these features is available [112]. Direct injection [113] was also implemented.

Xen (Dom0-less) Hypervisor. Xen [114] is an open-source hypervisor widely used in a broad range of application domains. A key distinct feature of Xen is its dependency on a privileged VM (Dom0) that typically runs Linux, to manage non-privileged VMs (DomUs) and interface with peripherals. Xen was initially designed for servers and desktops but has found also adoption on embedded applications. For embedded and automotive applications, Xilinx has led the implementation of Xen Dom0-less. With this novel approach, it is possible to have a Xen deployment without any Dom0, booting all guests directly from the hypervisor and statically partitioning the system. A patch for guest and hypervisor cache coloring support [115] is available. There is also an SIG working towards facilitating downstream FuSa certifications by fostering multiple initiatives within the community including MISRA refactoring or providing the option of running Zephyr [116] as Dom0. Besides Armv8-A, Xen also supports x86, and Armv8-R and RISC-V ports are underway.

Bao Hypervisor. Bao [117] is an open-source SPH that was made publicly available in 2020. It implements the pure static partitioning architecture, i.e., a minimal, thin layer of privileged software that leverages the existing ISA virtualization primitives to partition the hardware. Bao has no scheduler and does not rely on any external libraries or privileged VM (e.g., Linux), consisting of a standalone component that depends only on standard firmware to initialize the system and perform platform-specific tasks such as power management. Bao originally targeted Armv8-A [117]. The mainline now includes support for RISC-V [118], Armv7-A, and Armv8-R ports are in the making. Bao was specifically designed to provide strong real-time and safety guarantees. It implements hardware partitioning mechanisms to guarantee true freedom from interference, i.e., cache coloring (VM and hypervisor), and direct interrupt injection. There are ongoing efforts to implement memory throttling.

seL4 CAmkES VMM. seL4 is a formally verified microkernel [119]. Its design model revolves around the use of capabilities. When used as a hypervisor, seL4 executes in hypervisor mode (e.g., EL2) and exposes extra capabilities and APIs to manage virtualization functionality [120]. A user-level VMM uses its resource capabilities to create VMs. As of this writing, only the seL4 CAmkES VMM [121, 122] code is open-source. Each CAmkES VMM manages a single VM. One current issue of the CAmkES VMM is that, although it supports multicore VMs, each VMM runs as a single thread pinned to a single CPU. seL4 supports x86, Armv7/8-A, and RISC-V, but the latter is not supported by CAmkES VMM. In CAmkES, resources are statically allocated to each component using capabilities. Originally, seL4 provided only a priority-based preemptive scheduler. The newest MCS kernel extends it with scheduling context capabilities, allowing time management policies to be defined in user space [123]. Cache coloring has also been implemented in seL4 [124], not only at the user/VM level, but also for the kernel, but it was not publicly available at the time of writing. seL4 has formal proofs for its specification, implementation from C to binary, and security properties [125, 126]. There are also ongoing efforts to extend the formal verification to prove the absence of covert timing channels [127]. Finally, CAmkES is being deprecated shortly in favor of the seL4 Core Platform (seL4CP) [128].

3.2.4 Static Partitioning Hypervisor Analysis

We have performed an analysis of SPHs in [129]. The following are the work's main insights.

Takeaway 1. Due to the lack of efficient hardware support for directly delivering interrupts to guests in Arm platforms, all SPH increase the interrupt latency by at least one order of magnitude. However, by design, SPHs such as Jailhouse and Bao can achieve the lowest latencies as they provide an optimized path for hardware interrupt injection.

Takeaway 2. Interrupt latency increases tenfold under interference workloads. Applying cache coloring to VMs proves very beneficial, but for it to be fully effective, it is imperative to reserve a color for the hypervisor itself.

Takeaway 3. The direct injection technique is effective in addressing the shortcomings of GIC interrupt virtualization, as results demonstrate that interrupt latency overhead is reduced to near-native laten-

cies.

Takeaway 4. Only Xen and Bao respect interrupt priority order. Additionally, we observe that for all SPH, if multiple interrupts are triggered simultaneously, there is a partial priority inversion as lower priority interrupts take precedence due to the need for the hypervisor to handle and inject them.

Takeaway 5. IPI latency reflects the same overheads of external interrupts. Future Arm platforms might reduce them with GICv4.1 [78]. In the short term, direct injection might alleviate this issue. However, both approaches fall short of achieving native latency as they still pay the price of emulating the write to the "IPI send" register.

Takeaway 6. Inter-VM notification latencies are significant and, as is the case for hardware interrupts, very susceptible to the effects of interference. However, for bulk data transfers, it does not seem to significantly affect throughput if the shared buffer size is chosen on a range of about one-fourth to half the LLC size (i.e., 256 KiB to 512 KiB).

Takeaway 7. The major bottleneck for the VM boot time is caused by the bootloader, not the hypervisors. Notwithstanding, the hypervisor can significantly increase the boot time of a critical VM (small RTOS) when booting it alongside a larger VM (e.g., in a dual-OS Linux+RTOS configuration).

Takeaway 8. Hypervisors specifically targeting static partitioning have the smallest code bases. Despite facilitating certification, none of the evaluated SPH provides other artifacts (e.g., requirements specification, coding standards). Xen is the first to take steps in this direction; nevertheless, seL4's formal proofs provide the most comprehensive guarantees.

Takeaway 9. SPHs do not incur meaningful performance impacts due to: (i) modern hardware virtualization support; (ii) 1-to-1 mapping between virtual and pCPUs; and (iii) minimal traps. However, one key aspect is that SPH must have support for / make use of superpages to minimize TLB misses and page-table walk overheads.

Takeaway 10. Multicore memory hierarchy interference significantly affects guests' performance. Cache partitioning via page coloring is not a silver bullet as despite fully eliminating inter-core conflict misses, it does not fully mitigate interference (up to 38 pp increase in relative overhead).

3.2.4.1 Hypervisor Selection as Basis for CROSSCON

The CROSSCON Hypervisor should prioritize security while maintaining low levels of performance degradation, and offer mechanisms to minimize interference and side-channels. This section presents a comparison between the SPHs seL4 CAmkES, Xen (Dom0-less), Bao, and Jailhouse, across the following dimensions:

Performance Impacts. Xen (Dom0-less), Bao, and Jailhouse exhibit similar levels of performance impact (<1%), whereas seL4 CAmkES can reach as high as 7%.

Interference mitigation. Interference significantly affects the benchmark execution over all hypervisors. On Jailhouse, Xen, and Bao performance is degraded by a similar factor, i.e., to a maximum of about 105%; seL4-VMM is more susceptible to interference, reaching up to 125% in the worst case. Coloring can only reduce interference but not completely mitigate it. In the experiments, the interference workload runs continuously, however, in a more realistic scenario, it might be intermittent. seL4 CAmkES VMM cache coloring feature is not openly available yet.

Interrupt Latency. Bao and Jailhouse incur the smallest increase, albeit significant, to an interrupt latency of about 4x (840ns) and 5x (1090ns), respectively. Xen shows an increase of about 14x (2800ns). seL4-VMM presents the largest interrupt latency (47x, 9400 ns), an order of magnitude higher than Jailhouse and Bao.

Interference impact on latency. When enabling coloring, we measured no significant difference in interrupt latency compared to the base case. When enabling cache coloring in the presence of inter-VM interference, there is a visible improvement in average latency and variance. By applying coloring also to the hypervisor, Bao latency is reduced to almost no interference levels with negligible variance. Xen latency also drops considerably to an average of 6300ns.

Direct Injection. For the base case, i.e., no interference, the interrupt latency is near to native (about 210ns). Interference somewhat increases latency, but much less than in the previous experiments. By enabling coloring, it is possible to lower the average latency to near-native, 243 and 232 ns for Bao and Jailhouse, respectively.

Interrupt Priorities. Only Xen and Bao respect interrupt priority order.

TCB. Regarding Source Lines of Code (SLoC) Bao and Jailhouse have the smallest code base with about 8400 and 9900 SLoC. The hypervisor SLoC does not directly reflect the VM TCB, however. Although by design SPH such as Bao has a smaller SLoC count, the seL4-VMM is vastly superior from a security perspective: shared TCB is limited only to the formally verified microkernel because each VM is managed by a fully isolated VMM. From a functional safety certification standpoint, however, the VMM would still need to be considered. Moreover, seL4 formal proofs are limited to a set of kernel configurations, currently not including multicore. Regarding Jailhouse, despite its small size, the root cell is a privileged component of the system. It executes part of all VM management logic, being in the critical path for booting all other VMs. It is arguably part of all VM's TCB, increasing it significantly. Analogously, Xen must depart from true Dom0-less to leverage richer features (e.g., PV drivers, dynamic VM creation).

Cross Architecture Support. Bao has support for Armv8, Armv7, and RISC-V (w/ H extension). In the near future, Bao will support micro-controllers featuring TrustZone-M. Jailhouse supports x86_64 and Armv7 and Armv8. seL4 CAmKES VMM supports the major architectures, x86, Arm, and RISC-V. Xen supports x86, x86_64, and Arm architectures.

Hypervisor Selection. Overall Bao is the best candidate for CROSSCON Hypervisor. It offers low TCB, and low-performance impact, as well as state-of-the-art mechanisms to mitigate interference between guests, while offering compatibility with the major embedded systems architecture used in IoT devices, including RISC-V which is set to become an architecture with increasing presence in this space in the future. The additional flexibility required by CROSSCON will be incorporated into a fork of Bao to become the CROSSCON Hypervisor.

3.2.5 CROSSCON Hypervisor Features

The CROSSCON Hypervisor is based on the Bao SPH. However, the inherent constraints of pure SPHs limit their applicability in IoT systems. Two primary limitations hinder widespread adoption in this domain: (i) the absence of dynamic VM creation and management and (ii) the incapacity to deliver per-VM TEE services. This section outlines the preliminary design and implementation of these two features: Dynamic VMs and per-VM TEE services.

Sharing Physical CPUs. A common prerequisite for both Dynamic VM Creation and Management and per-VM TEE support is the ability for vCPUs belonging to different VMs to share a single pCPU, i.e., multiple VMs executing on one pCPU. This is foundational for both features, as they depend on a primary VM that can handle requests or provide services to secondary VMs running on the same pCPU, thereby establishing a cooperative relationship between them.

Parent-Child VM relationship. Figure 14 illustrates the currently implemented relationship model between VMs. We refer to VMs that can invoke other VMs as "parent VMs", and those that are invoked as "child VMs". The presented model transparently supports three general scenarios: (A) a parent VM with more than one child VM; (B) two statically isolated VMs (i.e., running in different cores), each featur-

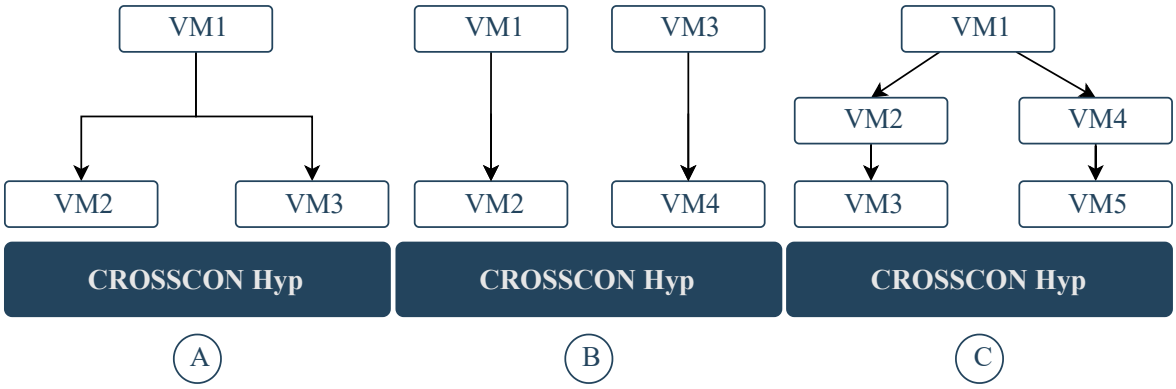


Figure 14: VM parent and VM child hierarchy.

ing its own child; and (C) child VMs featuring their own children. Combinations of these instantiations are supported. Such hierarchical VM model serve as the basis for TEE abstraction and isolation feature used in Section 3.1.5. To configure such scenarios and allow for the establishment of the parent-child between VMs we specified a CROSSCON Hypervisor's configuration file. This file defines various properties for each VM, such as the number of cores, size, and optionally, the location of memory regions and VM devices. Further details about configuration file were provided in D2.3.

Runtime execution stack design. During runtime execution, CROSSCON Hypervisor must provide interfaces with its VMs that allow for the invocation of, and the return of execution from, child VMs. These APIs are listed in Table 9. These interfaces are used from both primary and secondary VMs, with primary VMs controlling which VMs to invoke and when. Our design represents execution requests as a stack, adhering to the principles of the VM stacking mechanism, i.e., when a parent VM (VM1 in Figure 15) yields execution to a child VM (VM2), the parent VM is placed into the execution stack. Importantly, even while a secondary VM is executing, the primary VM remains capable of receiving and handling interrupts in a timely manner.

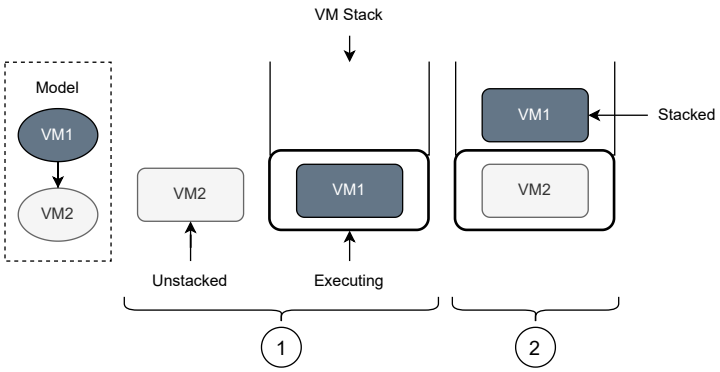


Figure 15: VM execution stack for CPU sharing.

For scheduling purposes, each child VM is treated as a substack. This means that scheduling a child VM actually schedules the last VM that the child pushed onto the stack. This scheduling approach is, in a sense, recursive, as some logic applies to a substack of the child VM and its child VMs.

3.2.5.1 Feature 1: Dynamic VM Creation and Management

This section describes the design and implementation of Dynamic VM Creation and Management features, which allows VMs to be instantiated, managed and destructed during runtime execution. The

implementation of dynamic VMs was also explored also as a basis for the CROSSCON Abstraction implementation, specifically to allow the concurrent execution of multiple heterogeneous TEE models.

Boot and Run time execution of Dynamic VMs. To implement this feature, the CROSSCON Hypervisor initializes the VM setup at boot time by parsing configuration files. At runtime, guest VMs can request the creation of new VMs using the same configuration file infrastructure. To support concurrent execution of multiple VMs on a single CPU, and to allow a parent VM to manage its child VMs, our design leverages a VM stacking mechanism and a hierarchical execution model. Figure 16 illustrates the architecture for dynamic VM creation. During boot time, a configuration file is used to instantiate VM1, while additional configuration files are preloaded to support the later creation of VM2. At runtime, VM1 issues a request to create VM2 dynamically. The resulting runtime hierarchy represents how VM1 (the parent VM) manages the execution of VM2 (the child VM).

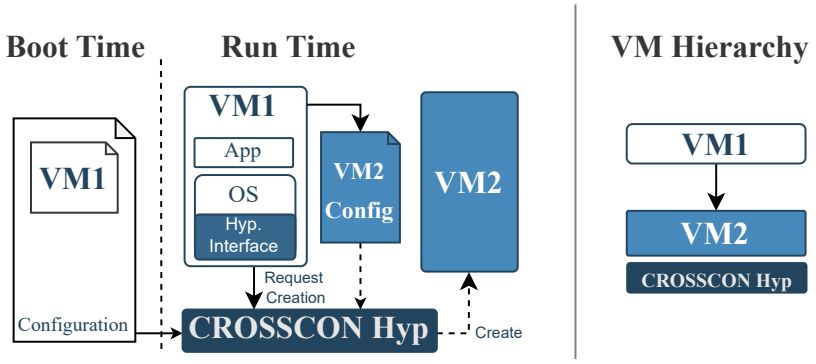


Figure 16: CROSSCON Hypervisor dynamic VM support: Dynamic VM Architecture and execution hierarchy.

Dynamic VM operations To support dynamic VMs, the parent VM must be able to create, invoke, and destroy child VMs. These operations require communication between the parent VM and the CROSSCON Hypervisor. To enable this interaction, the primary VM includes a CROSSCON Hypervisor driver that communicates with the hypervisor via the CROSSCON hypercall interface. This interface supports three core operations: VM creation, VM destruction, and VM invocation.

VM Creation. When a parent VM requires the creation of a child VM, it will issue the request through the CROSSCON Hypervisor driver to allocate memory for the child VM. The child VM will then take some of parent VM memory. To support the allocation process the child VM image is previously created and stored in a file). After the parent VM allocates the necessary resources, an application will copy the child VM information to the allocated memory, and issue a request to create it. This request is first received by the CROSSCON Hypervisor driver in parent VM, and then a similar request is sent to CROSSCON Hypervisor. CROSSCON Hypervisor will then take the memory region that the parent VM allocated to the child VM and remove it from the primary VM physical address space while mapping that same physical memory to the child VM. After the child VM is fully created, CROSSCON Hypervisor will give back execution control to the parent VM, which can then invoke the child VM.

VM Destruction. Destroying a child VM requires the execution of similar steps to its creation but in reverse. When the parent VM no longer requires the child VM services, it issues a child VM destruction call to the CROSSCON Hypervisor driver. The OS will issue a call to CROSSCON Hypervisor to destroy the child VM to regain access to the memory it donated. In the destruction process, CROSSCON Hypervisor will write the child VM's memory region to zero, thus preventing the OS from learning secrets when it regains access to the memory region. After this, CROSSCON Hypervisor will remap the memory region unto the primary VM's address space, control is given back to the OS, and eventually the application.

The application will issue a call to the OS to free the memory allocated for the child VM and finally, the OS will resume the application.

VM Invocation. Our current approach to child VM invocation simply resumes the execution of a VM. This presumes that a child VM is executing in a loop that parses the request from the parent VM and then serves it. Other more flexible approaches, such as entry point definition will be developed later on in the project. The CROSSCON Hypervisor provides the VM Invoke hypercall, which takes arguments to be delivered to the child VM, as well as an identifier of the child VM to be invoked.

Enclave programming Model Support. Our current implementation of dynamic VMs follows an enclave programming model, which will serve the coexistence of heterogeneous TEE programming onto a platform (see Section 3.1.5). Our implementation relies on a user-space runtime and a tailored CROSSCON Hypervisor kernel driver. The user space runtime provides three core functions to host applications: `create_vm`, `destroy_vm`, and `invoke_vm`. The kernel driver manages interactions with the CROSSCON Hypervisor. To support application execution within a VM, we developed a specialized runtime environment that is designed to invoke pre-registered handler functions inside the created VM and also forward requests to pre-registered handler functions in the host application.

Future Prospects: While the current support for dynamic VMs is primarily designed to create isolated execution environments during runtime, serving, for example, the support for TEE abstraction and the maintenance of multiple TEE models coexisting on the same platform (see Section 3.1.5), the underlying mechanism offers broader potential. In the future, the same dynamic VM infrastructure could be extended to support the creation of full-featured GPOS during runtime. Such functionality would support more flexible system designs, where resources can be partitioned and delegated at runtime, allowing one OS to manage other OS instances dynamically. Enabling the creation, invocation, and destruction of VMs as needed.

3.2.5.2 Feature 2: Per VM TEE service support

This section describes the design and implementation of per-VM TEE feature, which allows to VMs to request trusted services from one or multiple VMs operating as a TEE. This concept is further explored in Sections 3.1.4.1, 3.1.4.2, 3.1.5, where we provide a solution that allow us to shift from the traditional TEE model with a single trusted OS in the secure world to multiple trusted OS VMs in the normal world. This feature was designed and implemented for APU and MCU devices.

Similar to the dynamic VM features, per-VM TEE support leverage the VM stack mechanism to enable VMs to share pCPUs. Furthermore, we utilize the VM stacking model to bind VMs together, thereby linking a GPOS/RTOS VM with a Trusted OS VM.

Software Architecture:

To support per-VM TEE services, we introduce the concepts of *TEE VMs* and *GPOS/RTOS VMs* within the CROSSCON Hypervisor. TEE VMs host a trusted OS, such as OP-TEE or mTower, and are paired with a single GPOS/RTOS VM, typically running Linux or FreeRTOS.

CROSSCON Hypervisor extension: The CROSSCON Hypervisor adopts a modular design to handle events from heterogeneous VM types and TEE models through dedicated software modules, referred to as sdTEE, described in Section 3.1.5. Events from GPOS/RTOS VMs are managed by a dedicated GPOS/RTOS module. Similarly, TEE VM events are processed by a TEE-specific module, which handles both TEE operations and related GPOS/RTOS events, including interrupts and TEE-related calls. To identify (i) which VM operates as a TEE VM and (ii) which TEE VM the GPOS or RTOS is requesting interaction with, we introduce modifications at the GPOS level by providing a dedicated TEE driver instance for each supported TEE. As a result, when issuing SMC calls, the OSes uses different identifiers depending on the targeted trusted OS.

CROSSCON Hypervisor configuration: For configuration per-VM feature we also extend CROSSCON Hypervisor's configuration file in two ways. First, we introduce support for defining parent-child relationships between VMs. Second, we add a VM type field to classify each VM as either GPOS or TEE, enabling the CROSSCON Hypervisor to manage events according to the VM type.

CROSSCON Hypervisor role during context-switching: During runtime, requests for the TEE VM typically originate from applications running on GPOS/RTOS VM depending on the TEE model the VM is configured for. In the TrustZone model for APU devices, VMs interact with a trusted OS driver, which initiates SMC requests to the secure monitor running at EL3. These calls are intercepted by the hypervisor, which then performs a VM context switch to the corresponding TEE VM. In the TrustZone model for MCU devices, there is no monitor mode, and the processor's secure state is determined by whether the code is executed from memory regions mapped as Secure or Non-Secure. When VMs running on the RTOS VM attempt to access regions marked as Secure, execution is redirected to a NSC region managed by the CROSSCON Hypervisor. This region performs two actions: (i) it triggers the world switch by executing the SG instruction and (ii) it redirects execution to a common handler in the CROSSCON Hypervisor. This handler manages the context switch between the RTOS VM and the TEE VM in both directions. Such context switching from GPOS/RTOS to TEE VMs is handled by the CROSSCON Hypervisor, which (i) ensures switching of execution to the correct TEE VM and (ii) protects all data in general registers. Currently, the TEE VM supports OP-TEE as the trusted OS for APU devices and mTower for MCU devices. Support for other trusted OS may require dedicated integration.

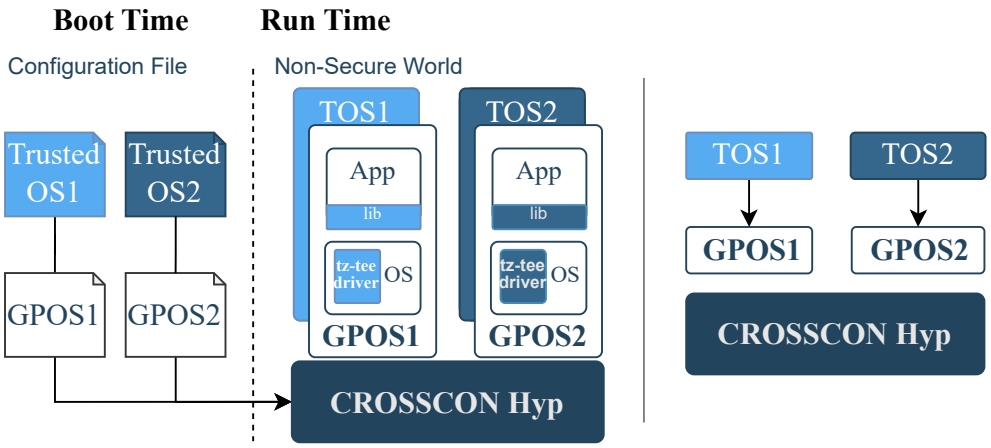


Figure 17: CROSSCON Hypervisor per-VM TEE support for TrustZone-A TEE and VM Hierarchy.

Figure 17 illustrates the per-VM TEE architecture during boot and runtime stages for APU devices. This configuration demonstrates the assignment of two RTOS VMs to two independent TEE VMs. At boot time, a configuration file defines the resource allocation for each VM, assigning two isolated GPOS VMs to two independent TEE VMs. During runtime, the CROSSCON Hypervisor hosts all VMs in the Non-Secure world, including both GPOS and TEE VMs, as specified in the configuration file. Assuming both VM pairs follow the TrustZone-A TEE model, each parent TEE VM is responsible for serving one dedicated child GPOS VM, as indicated by VM hierarchy. In the illustrated scenario, each VM pair runs on a separate core, ensuring strong isolation.

Figure 18 illustrates the per-VM TEE isolation architecture for MCU devices during both boot and runtime. This configuration demonstrates the assignment of a single RTOS VM to two distinct TEE VMs. Unlike the per-VM TEE implementation in the TrustZone-A model, the TrustZone-M model executes the CROSSCON Hypervisor in the Secure World. During boot, the hypervisor initializes all VMs and allocates resources according to a predefined configuration file. The same configuration format is used for both Armv8-M and Armv8-A architectures to maintain consistency. At this stage, TEE VMs are associated with their corresponding RTOS VM. To preserve compatibility with the traditional TEE model, the CROSSCON Hypervisor initially redirects execution to the parent TEE VM (e.g., TOS1), which then transfers control

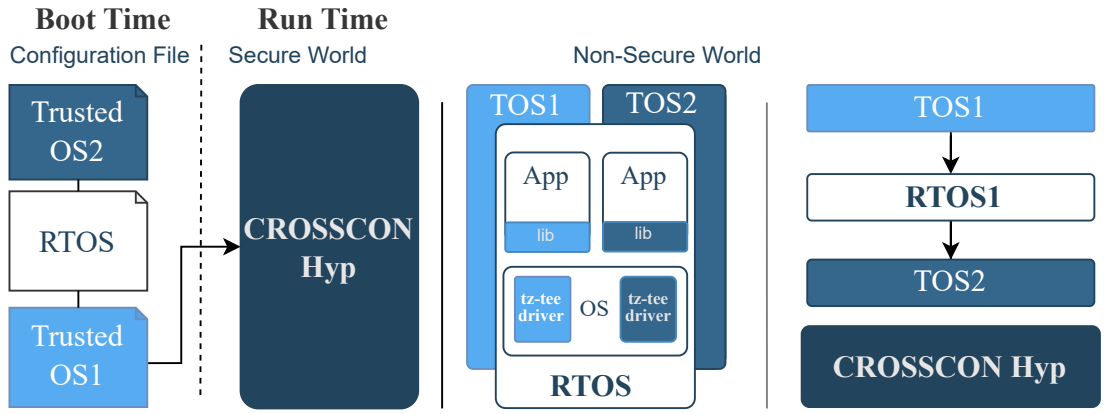


Figure 18: CROSSCON Hypervisor per-VM TEE support for TrustZone-M TEE and VM Hierarchy.

to the RTOS VM, reflecting typical boot flows where the TEE initializes first. During runtime, the RTOS VM may request services from either TOS1 or TOS2. To replicate this setup on MCU or APU devices, a single physical core is sufficient to support three vCPUs.

Augmented TrustZone and limitations of per-VM TEE feature. Compared to native Arm TrustZone, per-VM TEE feature introduces several advantages in terms of security and flexibility. While the per-VM model replicates the native TrustZone architecture, it also provides additional features that address some of its limitations, such as the excessive privileges granted to the trusted OS. In per-VM, the memory access rights of the trusted OS can be restricted, improving isolation. The model also enables the coexistence of multiple trusted OS instances in the same environment. However, there are current limitations. At this stage, only OP-TEE and mTower are supported as trusted OSes for APU and MCU devices, respectively. Supporting additional trusted OSes requires porting efforts. Furthermore, some trusted services depend on hardware components that are physically bound to the secure world. For example, authentication services may need access to cryptographic accelerators that retrieve keys stored in secure fuses or secure elements. Since our encapsulated VMs run in the normal world, they cannot directly access such components, which limits the support for certain hardware-backed trusted services.

3.2.6 Multiple VMM Support

In this section, we address the limitations of SPHs in supporting diverse execution environments and introduce a key CROSSCON feature that enhances their flexibility. The increasing heterogeneity of IoT workloads, which range from safety-critical, real-time tasks to general-purpose applications, calls for hypervisor architectures that go beyond simple, static partitioning designs. While SPHs provide strong isolation and are well-suited for mixed-criticality systems, their minimalism often limits their ability to support full software stacks with varying requirements. To address this, we argue future systems must adopt more modular designs. A key feature toward this goal is multi-VMM support, which enables the coexistence of different VMM, each tailored to specific VM needs. This approach enhances flexibility without compromising the safety, real-time guarantees, or certification benefits of SPHs.

Hypervisors range from "feature-rich and flexible" to "safe and real-time." SPHs, such as Bao, as previously discussed in Section 3.2.4.1, offer strong safety, real-time behavior, and certification support for critical guests and are suitable for MCSs, due in large part to their simplicity. However their limited features results in poor flexibility. SPHs typically lack dynamic memory management, support for multiplexing multiple VMs on a single CPU, device sharing, VM introspection, and health monitoring capabilities.

Ideally, low-criticality partitions could support more features. However, adding these features increases the size of the TCB, makes certification more difficult, and reduces real-time guarantees. We argue that

the challenge of balancing strict static partitioning for safety with the need for flexibility comes from the common practice of using virtualization to implement partitioning. Because of this, most hypervisors rely heavily on virtualization features and often ignore other hardware-based partitioning options. For example, while IOMMUs are widely used for I/O isolation, ignoring other platform-level isolation features can lead to lower performance and less predictable timing. In addition, many systems do not make use of newer memory protection technologies. Solutions such as RISC-V's ePMP [82], RISC-V Supervisor Domains Access Protection (SmMTT)[130], and CHERI[131] could provide stronger isolation with low runtime cost and should be considered in future system designs.

To overcome SPH flexibility limitations without compromising their VM requirements, CROSSCON offers multiple VMM support, allowing VMMs addressing different VM requirements to coexist on the same platform. This feature follows two main principles: staticity and decoupling.

- **Staticity:** Resource assignments, number of partitions, and communication channels are defined in advance. This makes it possible to pre-compute access control rules and system initialization offline. As a result, the system can rely on a minimal static code base, which is well suited for safety-critical environments;
- **Decoupling:** Virtualization and partitioning are cleanly decoupled. In traditional microkernel systems, a user-space VMM handles virtualization tasks, while the kernel enforces partitioning. Similarly, our architecture delegates VM management to per-partition VMMs operating at hypervisor level, while a higher-privileged "partitioner" enforces partitioning policies. This separation minimizes the shared TCB, allows each VMM to be tailored to the specific needs of its associated VM, and enhances fault containment and system availability by preventing faults in one TCB from impacting others.

Following the aforementioned principles, our design aims to enhance the flexibility of SPHs, enabling support for multiple VMM implementations, meeting different VM requirements, without compromising critical partitions' TCB and real-time properties.

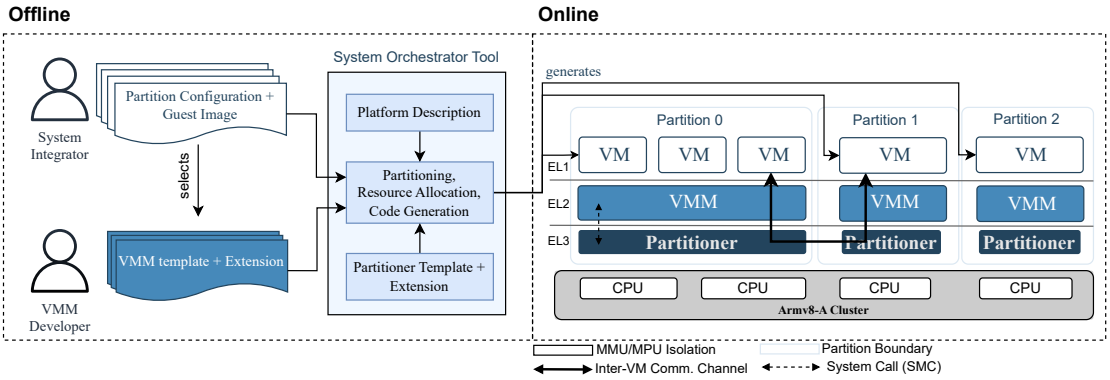


Figure 19: Overview of offline and online architectures in Multi-VMM systems.

System architecture description: Our solution is divided into two main phases: an offline phase for system configuration and image generation, and an online phase for runtime enforcement and virtualization. Figure 19 presents an overview of our system, covering both the offline and online phases. The design involves three main components: (i) the System Orchestrator Tool (SOT) in the offline phase, (ii) the partitioner, and (iii) the VMM in the online phase. The SOT applies the principle of staticity by (i) assigning resources based on platform descriptions and system configurations and (ii) generating the final system images. In the online phase, the partitioner and VMM follow the principle of decoupling. The partitioner is responsible for enforcing partitioning, while the VMM handles virtualization.

Table 10: Partitioner API by capability type (Arm ISA).

Type	Op	Param	Description
IRQ*	`set_enable`	bool	Enable/disable the IRQ
	`set_state`	enum	Set IRQ to active, pending, both, or inactive
	`set_priority`	int	Set IRQ priority
	`set_cfg`	enum	Set IRQ as edge or level triggered
	`set_type`	enum	Set IRQ to be forwarded as an *IRQ* or *FIQ*
	`set_target`	CPU cap	Set CPU as IRQ target
CPU	`start`	address	Start the CPU at the specified address
	`stop`	--	Stop the CPU
	`send_ipi`	int	Send the *nth* IPI to the CPU
vCPU	`restore`	--	Restore the vCPU critical registers
Notif	`send`	int	Send the notification with an event ID
Log	`print`	string	Print to the log a limited-size message

*All IRQ functions can be consolidated into a single system call to minimize the number of traps to the partitioner when configuring an interrupt.

System Orchestrator Tool

The SOT operates in the offline phase and takes three main inputs: (i) a platform description that defines hardware resources and their connections, (ii) partition configurations that specify the VM, their requirements, and the VMMs managing them, and (iii) VMM-specific extensions to support different VMM implementations. Both the platform description and partition configurations are expressed using Device Tree (DT).

The workflow begins by allocating resources to each VM. It then configures, generates, builds, and inspects the VMM binaries. This is followed by similar steps for the partitioners, which set up isolation mechanisms. The process ends with the generation of the final system images.

During configuration, the SOT creates a partition manager for each partition. These managers enforce access control using a switch-case structure indexed by unique capability IDs. When a VMM requests access to a resource, it invokes its partition manager with the corresponding capability ID. The switch-case logic dispatches the request to a handler that applies memory permissions defined for that capability. This mechanism enables VMMs to safely access privileged resources such as the interrupt controller, critical CPU registers, CPU power states, inter-partition notifications, and shared logging (see Table 10). All arguments are passed through registers, and there is no shared memory between the partitioner and the VMM. This design follows established conventions such as Arm Secure Monitor Call Calling Convention (SMCCC) and RISC-V SBI.

Partitioner

The partitioner components are generated by the SOT based on the system configuration files. Each partitioner runs at the highest privilege level, EL3 monitor mode on Armv8-A processors, and responds only to system calls from its associated VMM. When a VMM makes a request, the partitioner checks whether the VMM has the required permissions and either carries out the operation or returns an error. Each partitioner instance is designed to operate with minimal dependencies on other instances, and shared state between partitioners is kept to a minimum.

VMMs are designed to meet the specific needs of their partitions, and these differences are reflected in their corresponding partitioner instances. For example, the system may run one VMM with minimal functionality to support full virtualization, while another third-party VMM supports multiple dynamically

scheduled VMs. Both can coexist on the same platform, each with its own dedicated partitioner instance tailored to its partition configuration.

Virtual Machine Monitors

In this framework, VMMs run at the hypervisor privilege level with restricted capabilities. Access to critical operations is mediated by the partitioner. This includes access to privileged system registers and shared hardware resources such as interrupt controllers and power management units.

Interrupt Management A key detail regarding interrupt controllers, like Arm's GIC, is their split architecture. Modern interrupts controllers consist of a central shared component (e.g., the distributor in Arm's GIC architecture), responsible for interrupt configuration, and a set of per-CPU interfaces that handle the local interrupt state (e.g., interrupt acknowledgement). Because the distributor is shared across all CPUs, access to it must be mediated by the partitioner. In contrast, VMMs can directly access their local CPU interfaces to handle and inject interrupts without partitioner involvement.

System Registers VMMs use hypervisor-level system registers to perform operations such as virtual memory management. In a system with multiple VMMs, uncontrolled access to these registers can break isolation between partitions. For example, one VMM could potentially map and access the memory of another VMM. To prevent this, we classify these registers as critical and assign the partitioner the responsibility of configuring them before VMM execution. During runtime, any access to critical registers must go through the partitioner. On RISC-V platforms, the use of PMP-enhancement extension (ePMP), configured by the partitioner at initialization, enables VMMs and VMs to safely manage their own page tables and directly access certain critical registers. As a result, only access to shared components, such as the interrupt controller, CPU state, and IOMMU configuration, is mediated by the partitioner.

IOMMU Configuration To ensure that each VM can access only its assigned resources, platform-specific filters are used to control access from each bus master, such as a DMA controller or CPU, to shared resources like memory. For systems with an IOMMU, the tool generates filter tables for each partition. These tables are embedded into the partitioner's code. At boot time, the partitioner programs these access control settings into the hardware, ensuring that all protections are in place before the system begins multi-partition execution.

Inter-VM communication Inter-VM communication relies on two key primitives commonly provided by secure partitioning hypervisors: shared memory and asynchronous notifications. These are combined into communication objects that can be shared across multiple VMs, with the tool configuring memory mappings and virtual interrupts accordingly. Notifications are triggered via hypercalls, and depending on whether the communicating VMs are in the same partition or not, the virtual interrupt is either injected directly by the VMM or relayed through the partitioner via VMMs.

Multiple VMM Implementation

We prototyped this feature on both Arm and RISC-V platforms. For Arm AArch64, we target Xilinx's Zynq UltraScale+ (ZUS+), which features a quad-core Cortex-A53 cluster, a GICv2 interrupt controller, an SMMUv2, and custom firewall components (XMPU and XPPU). For RISC-V, we use QEMU as no silicon with hypervisor extensions is currently available. The RISC-V setup includes a PLIC interrupt controller and an IOMMU.

To support multiple VMMs, we implemented a set of functionalities to meet the goals described above. These include an Symmetric Multiprocessing (SMP) bare-metal runtime, synchronization primitives, Translation Lookaside Buffer (TLB) shootdown mechanisms, VM and vCPU management, basic trap and hypercall handling, virtual device emulation, virtual interrupt controllers, and emulation of standard

firmware services such as Arm's Power State Coordination Interface (PSCI) and RISC-V's SBI. Using this library, we developed two lightweight VMMs. The first, VMM-static, is a minimal and static VMM that hosts a single VM and replicates the behavior of traditional SPHs. The second, VMM-rr, is a more advanced VMM that supports time-sharing across multiple vCPUs using a round-robin scheduler. VMM-static required only minimal integration effort. In contrast, VMM-rr required the implementation of context switching and a timer driver to support preemptive scheduling.

Binary Inspection We use same-privilege isolation techniques on the Arm architecture to enforce isolation between partitioner instances. Same-privilege isolation technique is also used to isolate VMMs on Arm platforms. On RISC-V, however, isolation is achieved by trapping accesses to critical registers in the partitioner, making same-privilege isolation technique unnecessary for VMMs. To apply same-privilege isolation technique for VMMs on Arm, binaries must be inspected to ensure they do not contain instructions that access critical system registers. These registers control virtual memory and other privileged features. For Armv8-A, the critical registers include `SCTLR_EL2`, `TTBR0_EL2`, `MAIR_EL2`, `HCR_EL2`, `VTTBR_EL2`, and `VTCR_EL2`. On RISC-V platforms, inspection is not required, as the partitioner can trap and control access to critical registers at runtime.

System Initialization During initialization, each partitioner sets up its runtime environment by configuring critical system registers, virtual memory, VMM state, and IOMMU or firewall settings. After the setup is complete, unnecessary partitioner privileges are revoked by invalidating access to the initialization code and MMIO regions. This is done through a single update to the root page table, ensuring that these regions are no longer accessible during runtime.

3.3 New Trusted Applications

This section covers the development of novel TAs, also known as trusted services, in which we aim to develop new applications to complement or enrich existing applications such as secure boot, remote attestation, or cryptographic functions. The need for novel TAs is specified in the requirements defined in WP1 and stems from the specifications defined in WP2.

Specifically, we aim to devise a set of novel TAs for deployment within our UCs to complement or enhance the security and privacy. This set of TAs is associated with a certification manifest that is used to attest the correctness of the services to compositionally verify an entire IoT application. In the following we present our applications, namely PUF-based authentication, context-based authentication, FPGA-related secure provisioning, behavioral-based anomaly detection, and control flow integrity.

3.3.1 PUF-based Authentication

This TA aims to enhance the security of authentication protocols by leveraging properties of the device hardware, providing alternative factors. During the manufacturing process of semiconductors and integrated circuits, physical variations occur naturally and are tolerated as long as they remain within specified tolerances and do not impede the desired functionality of the hardware.

In the case of PUF, it is a hardware-based primitive that relies on these inherent hardware variations. It refers to a physical object whose operation cannot be reproduced physically, for example, by reproducing the employed system. When challenged by a verifier with a given input and conditions, a PUF provides a physically unique response, resulting in a digital fingerprint.
Thus, authentication may be based on PUFs. In PUF-based authentication, the devices possessing the PUF must convince another party, which possesses certain PUF-related information, that they indeed have the authentic PUF. These parties are referred to as provers and verifiers.

During authentication the PUF is fed with a challenge, which is essentially an input value acting as a

stimulus to which the PUF must respond. When the challenge is applied, it interacts with the unique physical characteristics of the device. The PUF uses an internal mechanism (e.g., a specific circuit layout designed to amplify these variations) to process the challenge. This process transforms the challenge into an output through a complex interaction that is unpredictable and depends on the device's specific physical properties. The result of this transformation process is a unique response, which is essentially unique for the device for that specific challenge. The response depends both on the challenge presented and the unique physical characteristics of the device. Therefore, even if two devices receive the same challenge, their responses will differ due to their unique physical variations.

The use of a PUF effectively eliminates the need to store secret information on the device but rather centres around challenging the PUF and generating the necessary information in the form of PUF responses generated on demand.

Background on PUF-based Authentication In PUF-based authentication, the exceptional and unpredictable responses produced by the PUF serve as the basis for verifying the identity of a remote device. In most solutions, there are two entities involved: a PUF-enabled *Prover*, and a powerful server known as the *Verifier* [132, 133, 134, 135]. The Verifier executes the protocol to verify the identity of the Prover.

The standard protocol is depicted in Figure 20 and comprises two steps:

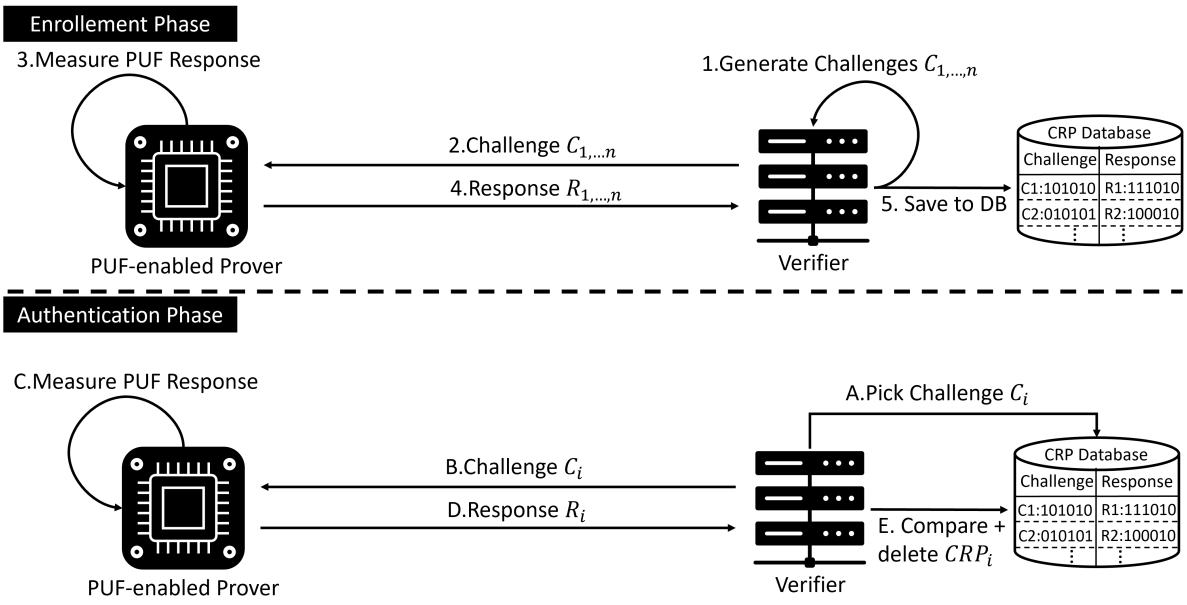


Figure 20: Standard PUF-based authentication protocol.

1. **Enrollment Phase:** During the enrollment phase, the Verifier initiates a random set of challenges, denoted as $\{C_1, C_2, ..., C_n\}$, and requests the corresponding responses from the Prover's PUF, denoted as $\{R_1, R_2, ..., R_n\}$. The resulting Challenge-Response PairsChallenge-Response Pairss (CRPs) $\{R_1C_1, R_2C_2, ..., R_nC_n\}$ are securely stored in a database DB maintained by the Verifier. This enrollment is a one time procedure and must be conducted offline in a protected environment to ensure the confidentiality and integrity of the responses.

2. **Authentication Phase:** The Verifier initiates the authentication by randomly selecting a Challenge-Response Pair (C_i, R_i) from its database (DB). The Verifier then transmits the challenge C_i to the Prover, who inputs it into its PUF to generate a response $\tilde{R} = PUF(C_i)$. The Prover sends this response back to the Verifier who compares the received response with the one stored in its database and accepts the authentication if they match and rejects it otherwise. Afterward, the used CRP gets deleted from DB.

Related work on PUF-based Authentication Research in PUF-based authentication can be categorized into two primary domains: (i) Device-to-server authentication, where a single trusted and powerful verification server authenticates a multitude of potentially resource-limited prover devices, and (ii) device-to-device authentication, where PUF-based authentication between two equal and potentially resource-limited devices is considered. The CROSSCON approach is particularly interested in the prospects of the latter domain, exploring authentication directly between devices without relying on a central server.

Device-to-server approaches are considered in works [136, 137, 138, 139, 140, 141], which adopt the traditional approach generating a comprehensive CRP database during the enrollment phase, using and then discarding a single challenge per authentication. Some schemes reduce storage requirements on the server side by training a Machine Learning (ML) model that models PUF, and use it instead of the database [142, 143]. However, the common property of all device-to-server schemes is the necessity for the Verifier to store a significant amount of data, either in the form of a CRP database or a ML model. Furthermore, this design assumes a trustworthy Verifier who can keep the CRPs database or its ML model confidential from other parties.

To realize PUF-based device-to-device authentication, several proposed schemes such as [144, 145, 146, 147] introduced a trusted third party that maintains confidential PUF information on behalf of the Verifier. Further schemes opted to introduce a trusted intermediary that mediates communication between parties [135, 148, 149, 150, 132, 151, 152, 153, 154]. These schemes rely on a trusted third party to authenticate a Prover on behalf of the Verifier or authenticate both parties and support them by establishing an authenticated channel between the devices.

By delegating the storage burden of confidential PUF information to a trusted third party, these schemes enable PUF-based authentication between two resource-limited devices, even though at the cost of requiring an additional trusted third party. In large and heterogeneous IoT networks, meeting this requirement can be particularly challenging. Agreeing on a centralized trusted party to maintain confidential information on behalf of multiple mutually distrusting stakeholders can be especially difficult. Consequently, these schemes are not well-suited for IoT networks.

Further schemes, like [155, 156, 157, 158, 159], leverage PUFs to generate asymmetric keys for signature-based authentication. However, these schemes provide protection only against relatively weak adversarial models, as they reuse the PUF response to generate the asymmetric key. Consequently, if an adversary gains access to the devices during key generation, the device becomes compromised since no new PUF-response is utilized, in contrast to classical PUF-based authentication schemes. Moreover, some of these schemes employ public software-based PUF-simulators for response verification [160, 161]. Security in such scenarios relies on the assumption that an honest prover, possessing the genuine PUF, can generate a PUF response significantly faster than a dishonest party equipped only with the simulator. However, this impractical assumption about execution time renders these schemes unfeasible for IoT devices. Further schemes, like [155, 156, 157, 158, 159], leverage VMMs to generate asymmetric keys for signature-based authentication. However, these schemes provide protection only against relatively weak adversarial models, as they reuse the PUF response to generate the asymmetric key. Consequently, if an adversary gains access to the devices during key generation, the device becomes compromised since no new PUF-response is utilized, in contrast to classical PUF-based authentication schemes. Moreover, some of these schemes employ public software-based PUF-simulators for response verification [160, 161]. Security in such scenarios relies on the assumption that an honest prover, possessing the genuine PUF, can generate a PUF response significantly faster than a dishonest party equipped only with the simulator. However, this impractical assumption about execution time renders these schemes unfeasible for IoT devices.

In the context of device-to-device authentication within IoT networks, it is crucial that devices, regardless of their resource constraints, are capable of performing dual roles: Provers, i.e., authenticating themselves to other devices, and Verifiers, i.e., verifying the authenticity of incoming connections. The

existing landscape of PUF-based authentication solutions does not align well with these practical requirements of IoT networks, highlighting a critical need for an approach that is both flexible and universally implementable.

Objectives for PUF-based Authentication Based on the aforementioned challenges, the primary aim of our research is to develop a secure, efficient, and scalable PUF-based authentication scheme that empowers every IoT device to operate as both a Prover and an untrusted Verifier. The scheme must be lightweight to accommodate the inherent hardware limitations of IoT devices, particularly regarding storage and computational power. Our ambition is to craft an innovative solution that enables PUF-based device-to-device authentication, streamlining both authentication and verification processes for resource-constrained devices and heterogeneous IoT networks.

Approach The innovative aspect of this work lies in its handling of confidential PUF responses. The actual responses are never exposed, ensuring they remain fully protected. Instead, the scheme derives public information from these confidential responses using techniques such as zero-knowledge proofs, one-time signatures, or one-time chains. This public information is correlated with the underlying PUF responses but reveals nothing about them, allowing it to be safely stored on untrusted platforms such as bulletin boards or distributed ledgers.

For authentication, a device (referred to as the Prover) leverages the PUF and our authentication scheme to produce a specific set of information. This information indicates the possession of the concealed response without revealing it. The verifier then uses this information in conjunction with the public data to confirm the authenticity of the Prover, thus completing the verification process.

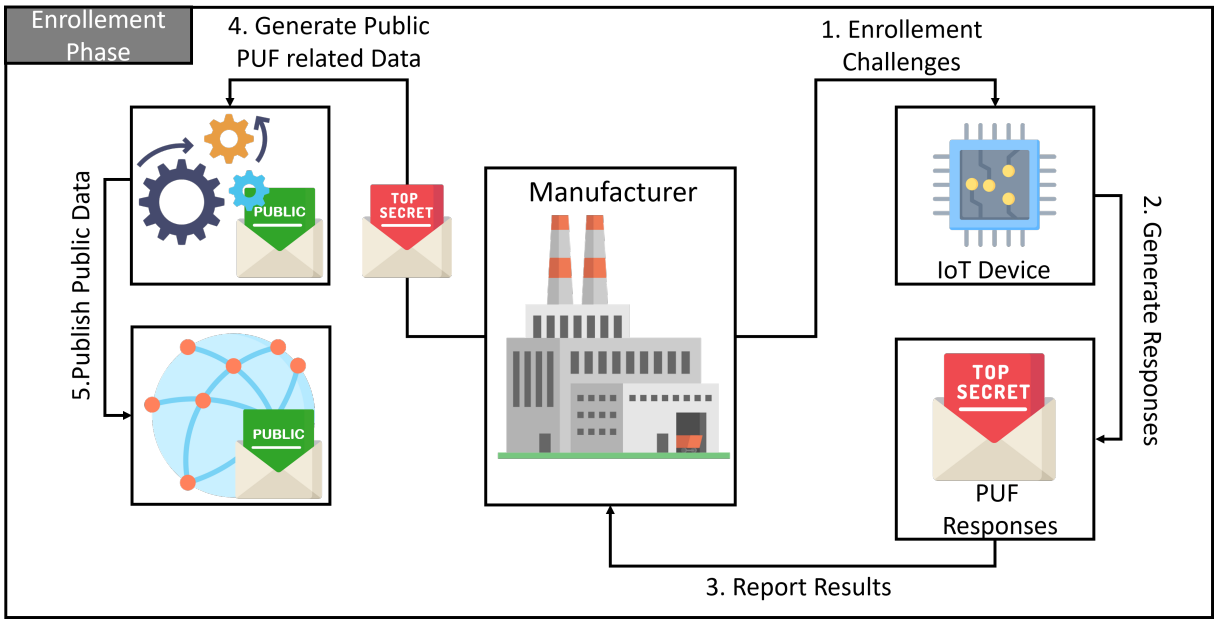
System model The system model of the proposed system incorporates three parties: the prover, the verifier, and the manufacturer, and it is divided into two distinct stages: the *Enrollment Phase* and the *Authentication Phase*, as depicted in Figure 21. While the manufacturer is trusted, its involvement is limited solely to the Enrollment Phase. During the Authentication Phase, both the prover and the verifier are considered untrusted entities, ensuring a decentralized and secure authentication process.

Enrollment Phase The enrollment phase is a critical initial step, usually conducted in a secure environment during the manufacturing process. In this phase, the manufacturer begins by querying the device's PUF with a predefined set of challenges. The device generates unique PUF responses to these challenges, which are then relayed back to the manufacturer. These initial three steps (depicted as steps 1-3 in Figure 21a) form the common foundation for most PUF-based schemes.

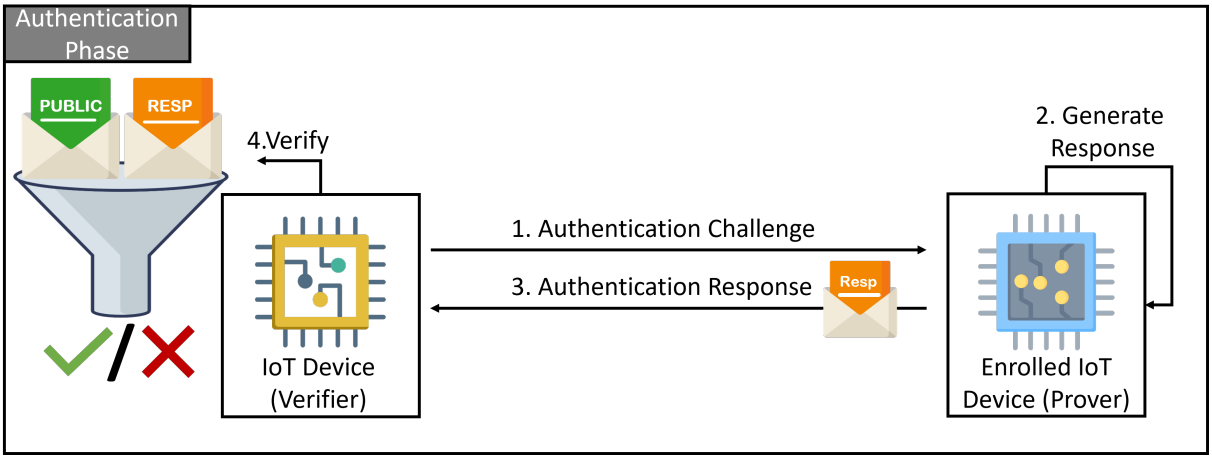
Unlike other schemes that typically conclude the enrollment phase after this third step and require the manufacturer to store confidential PUF-responses for later authentication, our approach incorporates two additional steps (steps 4 and 5 in Figure 21a. These steps are designed to transform the confidential PUF responses into public information. This transformed public information is then stored in a publicly accessible integrity-protected repository, effectively preparing the devices for their subsequent deployment. By storing this information using a public repository, we ensure that the manufacturer's role is limited to the initial setup.

Authentication Phase. In the authentication phase shown in Figure 21b, which commences post-deployment, an enrolled Prover device, upon receiving an authentication request, generates a confidential PUF response. Utilizing the proposed scheme, the Prover formulates an authentication response, which is then transmitted to the requesting entity, i.e., the Verifier. The Verifier employs a specialized verification algorithm that processes the received authentication response in conjunction with the public information previously disseminated by the manufacturer. It is essential to underscore that the IoT devices acting as Verifiers can also undergo enrollment during the manufacturing phase, enabling them to function as Provers.

By utilizing the described approach, two main challenges of device-to-device PUF-based authentication have been addressed:



(a) Enrollment Phase



(b) Authentication Phase

Figure 21: System Design Overview

- **Untrusted Verifier:** Previous PUF-based authentication schemes necessitate the Verifier to store confidential PUF-related data, usually in the form of Challenge-Response Pairs. As this information could potentially enable Verifiers to impersonate associated Prover devices, Verifiers are traditionally presumed trustworthy at all times. However, this assumption is impractical and unrealistic, especially when Verifiers may have limited resources. Therefore, our approach provides verification without depending on confidential information, effectively eliminating the necessity for trusted Verifiers. While some schemes attempt to mitigate this challenge by utilizing PUFs for generating asymmetric key pairs, they come with reduced security guarantees.
- **Authentication Proof generation and verification on resource-limited platforms:** Our approach eliminates the need for Prover or Verifier to store any confidential information. Instead, it allows them to outsource the storage of necessary public information to distributed ledgers or file systems. Furthermore, the developed scheme is lightweight enough to run on resource-limited MCUs, effectively enabling IoT devices to function as PUF-based Prover and Verifier.

The high-level architecture depicted in Figure 21 was realized in three different novel and distinctive methods that will be outlined in the following.

3.3.1.1 ZK-PUF: PUF-based authentication utilizing zero-knowledge proofs

The first scheme called **ZK-PUF: PUF-based authentication utilizing zero-knowledge proofs** adopts a novel strategy by employing a single challenge for multiple authentication sessions, which diverges from the conventional practice in PUF-based authentication. In standard PUF authentication, the response to a given challenge is disclosed, rendering it unusable for future sessions. However, this new approach introduces reusability by leveraging zero-knowledge proofs of knowledge. By proving that the prover possesses the response without actually revealing it, the scheme ensures the confidentiality of the PUF response remains intact. While this scheme is PUF agnostic, it might be of particular relevance for weak PUFs since they offer only a very limited set of CRPs and reusing them might be vital. The cryptographic principles employed, especially zero-knowledge proofs, entail intricate cryptographic operations. Despite our implementation being sufficiently fast, even on resource-constrained MCUs, our objective was to further enhance the efficiency of the authentication scheme. Additionally, although the zero-knowledge proof utilized does not leak any information about the underlying PUF response, reusing said response could pose challenges in the presence of a strong adversary capable of gaining runtime access to the victim prover. For example, such an adversary might attempt side-channel attacks to extract the secret response or dump temporary memory holding intermediate proof values. Replay attacks can be effectively mitigated by using a sufficiently large nonce (e.g. 64-bit).

System architecture illustrated in Figure 22 can be discussed starting with the individual enrollment process for each device, conducted by the manufacturer:

1. **Selection of Enrollment Challenges:** The manufacturer selects two distinct random challenges, C_1 and C_2 , for the device. It is imperative that $C_1 \neq C_2$. These challenges may be unique to each device or the same across all devices.
2. **Generating PUF Responses:** The chosen challenges C_1 and C_2 are input into the device's PUF, generating two confidential responses, R_1 and R_2. These responses are only required during the next step and deleted afterwards, so no confidential storage is required.
3. **Commitment to PUF Responses:** The device commits to the values R_1 and R_2 using a Pedersen Commitment [162]. We decided to use this commitment scheme since it allows for fairly efficient zero-knowledge proofs. The Pedersen commitment scheme includes the following algorithms:
   3.1. $KGen(1^n)$: Produces the public commitment data pd, comprising the protocol parameters G_p and the generators g and h. The commitment scheme requires two generators and, therefore, also two PUF-responses.

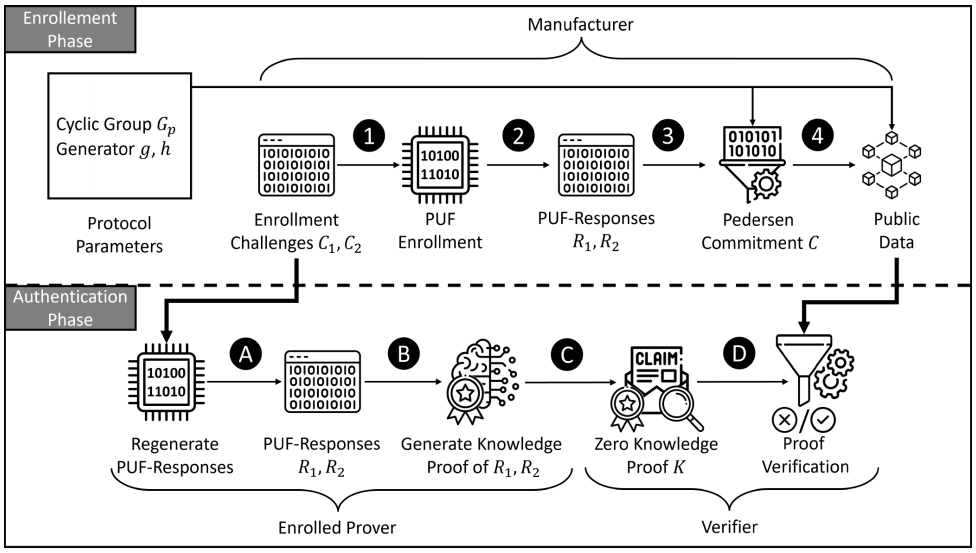


Figure 22: PUF-based Authentication leveraging Zero-Knowledge Proof System

3.2. $Com(pd, m)$: Creates the commitment $COM = g^{R_1} \cdot h^{R_2} \bmod p$, with pd as the protocol parameters and m as the responses R_1 and R_2.

3.3. $Ver(pd, m, COM)$: Enables verification of the commitment by checking if $Com(pd, m) = COM$.

4. **Publishing the Commitment:** The commitment COM is published in a public data storage. Importantly, there is no requirement to store any information confidentially.

With these steps, the enrollment phase is completed, establishing a secure foundation for device authentication. The device can now be deployed and transitioned into the authentication phase.

1. Once the device is enrolled, it can use the scheme to authenticate itself to other devices. The process is initiated by the verifier, which sends the prover's enrollment challenges C_1 and C_2 , along with a nonce n, to the prover. The nonce ensures the freshness of the authentication process and prevents the replay of old or recorded protocol runs. In response, the prover feeds the enrollment challenges C_1 and C_2 into its PUF to regenerate the responses R_1 and R_2.

2. These responses are then used to generate a zero-knowledge proof of knowledge for R_1 and R_2. For this, we utilize the Schnorr zero-knowledge proof system [163] for the knowledge of discrete logarithms. The protocol follows these steps:

   2.1. The prover picks two random values r and u.

   2.2. The prover calculates the Pedersen Commitment $P = g^r \cdot h^u \bmod p$.

   2.3. P , along with the nonce n, is then hashed using the public hash function H to generate the value $\alpha = H(P, n)$, without interaction with the verifier.

   2.4. The prover computes two zero-knowledge proofs, denoted as v and w, where $v = r + \alpha R_1$ and $w = u + \alpha R_2$. These proofs enable the prover to demonstrate knowledge of R_1 and R_2 to the verifier, without disclosing the actual values of R_1 and R_2.

   2.5. Finally, P, v, and w are transmitted to the verifier to confirm the authentication proof.

3. The verifier now checks the received values by applying the following steps:

   3.1. First, it generates α by applying the public hash function H on P and the previously transmitted nonce n, resulting in $H(P, n) = \alpha$.

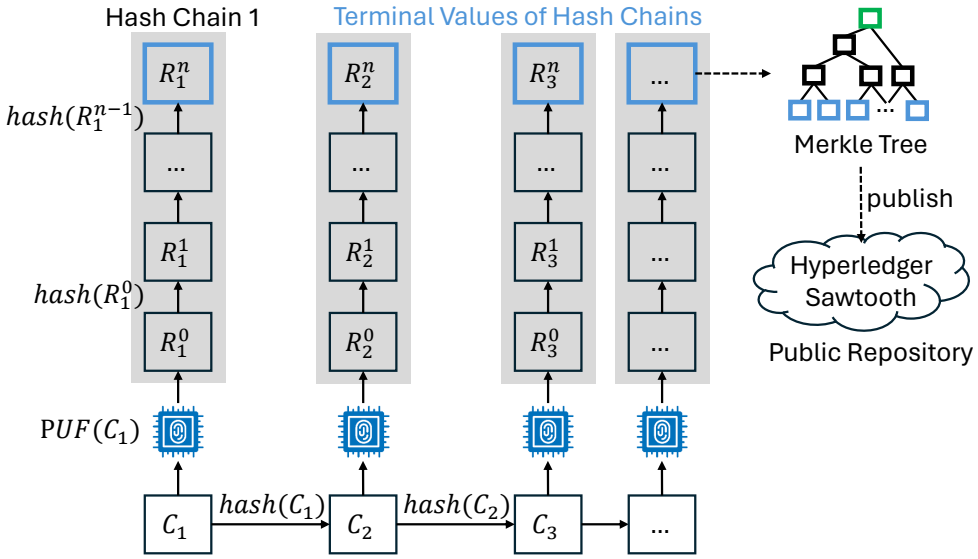


Figure 23: PAVOC construction of authentication keys.

3.2. It retrieves the public commitment COM, as well as g and h, and verifies whether $g^v h^w = P \cdot COM^\alpha$.

4. If the equation holds, the prover is authenticated; otherwise, the authentication process fails.

Correctness directly follows from this equation:

$$g^v \cdot h^w = g^{r+\alpha R_1} \cdot h^{u+\alpha R_2} = g^r \cdot g^{\alpha R_1} \cdot h^u \cdot h^{\alpha R_2} = P \cdot (g^{R_1} \cdot h^{R_2})^\alpha = P \cdot COM^\alpha$$

This verification process guarantees the verifier that the prover indeed holds the two PUF responses, R_1 and R_2, and further ensures the session's freshness. For the zero-knowledge proofs v and w to be validated, the prover is required to use the same α as the verifier, thereby incorporating the nonce n. This mechanism ensures the uniqueness and security of each session, effectively preventing replay attacks.

3.3.1.2 PAVOC: PUF-based authentication via one-way chains

The pursuit of a more efficient PUF-based authentication scheme protecting against even stronger adversaries motivated the development of our second solution called **PAVOC: PUF-based authentication via one-way chains**. This scheme incorporates cryptographic hash functions known for their efficiency and focuses on optimizing the use of the PUF. Contrary to the previously presented scheme that reuses a fixed Challenge-Response Pair, this scheme exploits the multitude of Challenge-Response Pairs a PUF can produce. This method involves enrolling n distinct Challenge and Response pairs. The responses are then converted into one-way chains, each capable of facilitating up to m authentications, where m denotes the length of the chain. This is done by applying a public hash function H repeatedly for m times to the underlying PUF response R_i. The one-way chains built from PUF responses are then used in reverse order for authentication. The sole prerequisite for its implementation is a loose time synchronization between the prover and verifier, necessitating both parties to maintain awareness of the current state of the one-way chain. This is necessary to ensure that both parties prover and verifier are aware of which chain element to be used and when the prover must reveal the used element and move to its pre-image for the next authentication.

Enrollment Phase: This phase involves several steps conducted before device deployment, which are illustrated in Fig. 23. Initially, the IoT device receives a challenge C_1. This challenge is processed by the

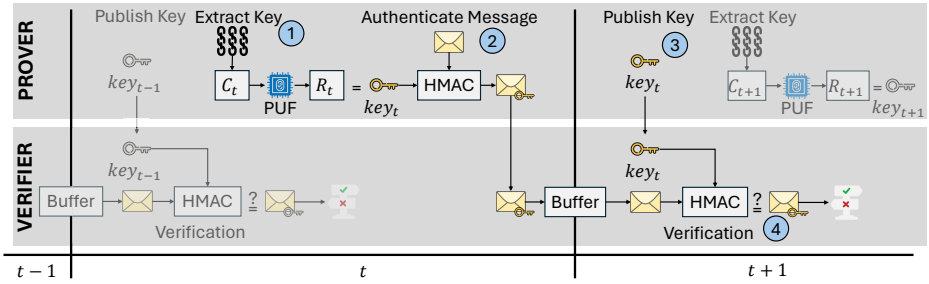


Figure 24: Chronological sequence of the authentication process with PAVOC, which spans over the two timestamps t and $t+1$.

PUF to generate a corresponding response R_1^0, which is then iteratively hashed based on the predefined size n of the hash chain. R_1^n represents the terminal value of the first hash chain. To expand the number of potential authentication keys, PAVOC constructs multiple hash chains using the same procedure. New challenges, e.g., C_2, are generated by iteratively hashing C_1, as shown at the bottom of Fig. 23. The manufacturer determines the number and length of these hash chains to ensure the device has sufficient keys for authentication throughout its life cycle.[1]

The terminal values of each chain marked as blue boxes in Fig. 23 are aggregated into a single, fixed-size representation. While any cryptographic accumulator could be used, we opted for a Merkle tree [164]. To ensure accessibility and transparency, the accumulator, which serves as the public identifier for the device, is published on a publicly accessible storage platform, such as a distributed ledger like Ethereum [165] or Hyperledger Sawtooth [166]. This approach eliminates the need to regenerate the Merkle tree [164] on the IoT device for each authentication, which would be impractical. Note, that such platforms do not have an availability issue. Due to the limited storage capacity on distributed ledger platforms, only the root of the Merkle tree [164] (green in Fig. 23) is stored on the ledger, with the rest of the tree being offloaded to a different publicly available storage platform. Since the tree does not contain sensitive information, any public storage can be used. [2]

This enrollment procedure prepares IoT devices for authentication without requiring significant computational resources or reliance on the manufacturer. Upon completion of the enrollment phase, the IoT device is ready for deployment and transition into the authentication phase. From this point onward, the device can use PAVOC authentication scheme independently, without reliance on any external parties, including the manufacturer. This ensures that the manufacturer is no longer involved or needed for the authentication process, enhancing security and autonomy.

Authentication Phase: Post-deployment, in the authentication phase, the heartbeat provider is introduced. This component synchronizes prover and verifier in the time domain by providing time slots based on an increasing counter. Those time slots are visualized as $t-1$, t, and $t+1$ in Fig. 24. The authentication process consists of four major steps and begins by determining the current element of the active hash chain according to the time slot. For instance, if the current time slot is 600 and the chain length is 1000, the corresponding chain element is $R_1^{1000-600} = R_1^{400}$. If the heartbeat exceeds the chain length, the next chain is utilized. Thus, for a heartbeat of 2500, the valid chain element is the 500th element of the third chain, R_3^{500}. This time-slot-based approach prevents the reuse of outdated chain elements.

To authenticate a message, the prover computes the hashed PUF response of the current element by iteratively hashing the first challenge to obtain the respective chain and then iteratively hashing the PUF response to get the corresponding chain value used as key, as shown in the first step in Fig. 24. It is

[1] Note, that only the PUF is mandatory to be executed on the IoT device. The hashes can also be executed on a separate machine of the manufacturer.

[2] We suggest using InterPlanetary File System (IPFS) [167], a free decentral file system. In IPFS [167] we only store leaves as membership tests are naively supported. Merkle tree [164] paths are automatically computed during requests.

only necessary to generate the chain up to the current value determined by the time slot instead of the entire chain. As second step, the resulting key (key_t in Fig. 24), along with the current time slot, is used as a symmetric key for authenticating messages via HMAC (step two in Fig. 24).

The message, along with the HMAC, is sent to the verifier. While the verifier directly receives the message, verification occurs in the next time slot (step three in Fig. 24). Once the heartbeat provider broadcasts the next time slot, the prover discloses the used key to the verifier and derives the next key for future authentications. Upon receiving the key, the verifier verifies the integrity and authenticity of the received message by checking the HMAC (step four in Fig. 24).

Further, the verifier computes the current hash chain's terminal value by hashing the provided key according to the chain length. This value is verified against the publicly accessible Merkle tree [164] via membership test to validate the key's authenticity and association with the prover. Note, that IoT devices acting as verifiers can also be enrolled during the manufacturing phase, allowing them to operate as provers.

Heartbeat provider: To synchronize the prover and verifier in the time domain regarding the active element within the hash chains, PAVOC employs a global heartbeat mechanism which serves as a reliable time reference. This lightweight component broadcasts periodic signals based on an increasing counter at predefined intervals (e.g., every 10 seconds, one minute, or a configurable interval).

Upon receiving a heartbeat signal, both, prover and verifier, advance to the next element in their respective hash chains. Additionally, the previously used hash chain element is disclosed to the verifier (step three in Fig. 24)). The heartbeat mechanism can configure the duration for which a specific chain element remains valid. Hence, by adjusting the frequency of the heartbeat signal, the system can adapt to different application scenarios, providing flexibility in terms of authentication frequency and granularity.

This approach transforms secret PUF responses into data that can be publicly verified, removing the need for a trusted third party to manage sensitive information. It avoids costly public-key operations, making it lightweight and suitable for constrained IoT devices. By not reusing PUF responses and combining them with hash functions and accumulators, it provides stronger security and scalable mutual authentication between devices.

3.3.1.3 PAWOS: PUF-based Authentication with One-time Signatures

Since PAVOC is built only utilizing cryptographic hash functions, it is significantly faster than ZK-PUF. Also, the impact of a runtime adversary is limited since each chain is only used a maximum of m times and a new response is used afterwards. However, it introduces an added layer of complexity: the requirement for maintaining time synchronization. This pursuit of a solution that leverages the effectiveness of hash functions without necessitating time synchronization resulted in the development of a third solution, dubbed **PAWOS: PUF-based Authentication with One-time Signatures**, which was later published under the name **AuthentiSafe** [168]. Within PAWOS, each PUF-response is transformed into a one-time signature. They utilize the strength of cryptographic hash functions to generate secure, single-use signatures. This approach not only retains the efficiency benefits of hash functions but also sidesteps the issues inherent in time-based systems. The aim is to build a system that upholds the hash function's efficacy while removing time synchronization challenges, offering a more adaptable and scalable solution that is applicable to a broader spectrum of applications.

For the verifier, this scheme simplifies the process to simply verifying the authenticity and correctness of the received one-time signatures. On the other hand, the prover's responsibility is to guarantee that each Challenge-Response Pair and its corresponding one-time signature pair are not reused. This aspect is crucial because, without it, the One-Time Signature could become vulnerable, allowing an adversary to deduce the used response and subsequently impersonate the prover. To prevent the reuse

of challenges, a minor modification to the PUF hardware is necessary, incorporating an additional hardware component - a small amount of memory directly connected to the PUF and exclusively writable by it. This memory component is initialized with the initial challenge during the manufacturing phase and subsequently updated after each PUF query. Given that the challenge's integrity is important but confidentiality is not required, basic security measures, such as write protection outside of the authentication process, are considered sufficient. A feasible approach would be to integrate a compact, specialized memory component into the PUF's architecture, specifically for storing and updating the challenge.

The design of PAWOS enables us to eliminate the need for time synchronization, a requirement in PAVOC, while maintaining significantly greater efficiency than ZK-PUF. Furthermore, since each response is utilized precisely once, any attempt by an adversary to obtain a response provides minimal advantage, as it becomes invalid in subsequent authentications.

As this work has been published under the name **AuthentiSafe** [168], detailed implementation and protocol descriptions are omitted from this document and can be found in the referenced publication.

3.3.1.4 GlobalPlatform Perspective

The ZK-PUF was also implemented as TA, as part of UC1.1, demonstrating CROSSCON Stack utilization on a constrained embedded platform without an MMU. The prototype adheres to the GP Client communication and lifecycle management model, while remaining agnostic to the specific PUF architecture and post-processing logic.

The primary goal was to provide a proof of concept on the portability side, illustrating how PUF-based Trusted Services can be standardized across heterogeneous platforms. Details regarding the hardware PUF primitive and its low-level access mechanisms are discussed separately in the D4.3 deliverable.

The platform selected was the NXP LPC55S69, one of the few commercially available devices with native PUF hardware support. To realize the TEE model, we utilized the CROSSCON Hypervisor to run two isolated VMs: one acting as a Trusted Application, effectively extending the secure world managed by the hypervisor.

At the time of development, no suitable MCU TEE implementation was available for deploying our TA. Consequently, we used a standard Zephyr-based VM to run the TA. Later in the project, the mTower project was ported to the NXP LPC55S69 platform, making it a viable option for future deployments. We then wrapped the CROSSCON Hypervisor's communication mechanisms with the GP Client API to enable interaction between the VMs in a manner consistent with GlobalPlatform's TEE Client API model.

Figures 25 and 26 illustrate the enrollment and authentication phases of the ZK-PUF Trusted Application prototype, respectively. These sequence diagrams capture the interaction between the Verifier (which may be a server or another device), the Guest VM (representing the prover in the normal world), the PUF VM (representing the prover in the secure world), and the underlying PUF hardware.

3.3.2 Context-based Authentication

Similar to PUF, context-based authentication may leverage the imperfections of the hardware, thus the physical layer of wireless transmitting devices in its environment. This is achieved by analyzing fluctuations in the signals induced by the corresponding hardware component and its manufacturing variations. This process is known as Radio Frequency Fingerprinting (RFF) and can be employed to, e.g., assign an identity to a device or a type of device. This trusted service aims to enhance the authentication process by developing a context-based authentication approach, which leverages the environment of a device regarding the wireless transmitters around it. The environment should be embedded into a digital fingerprint to enable a device to proof that it is located within a predetermined secure environment, which

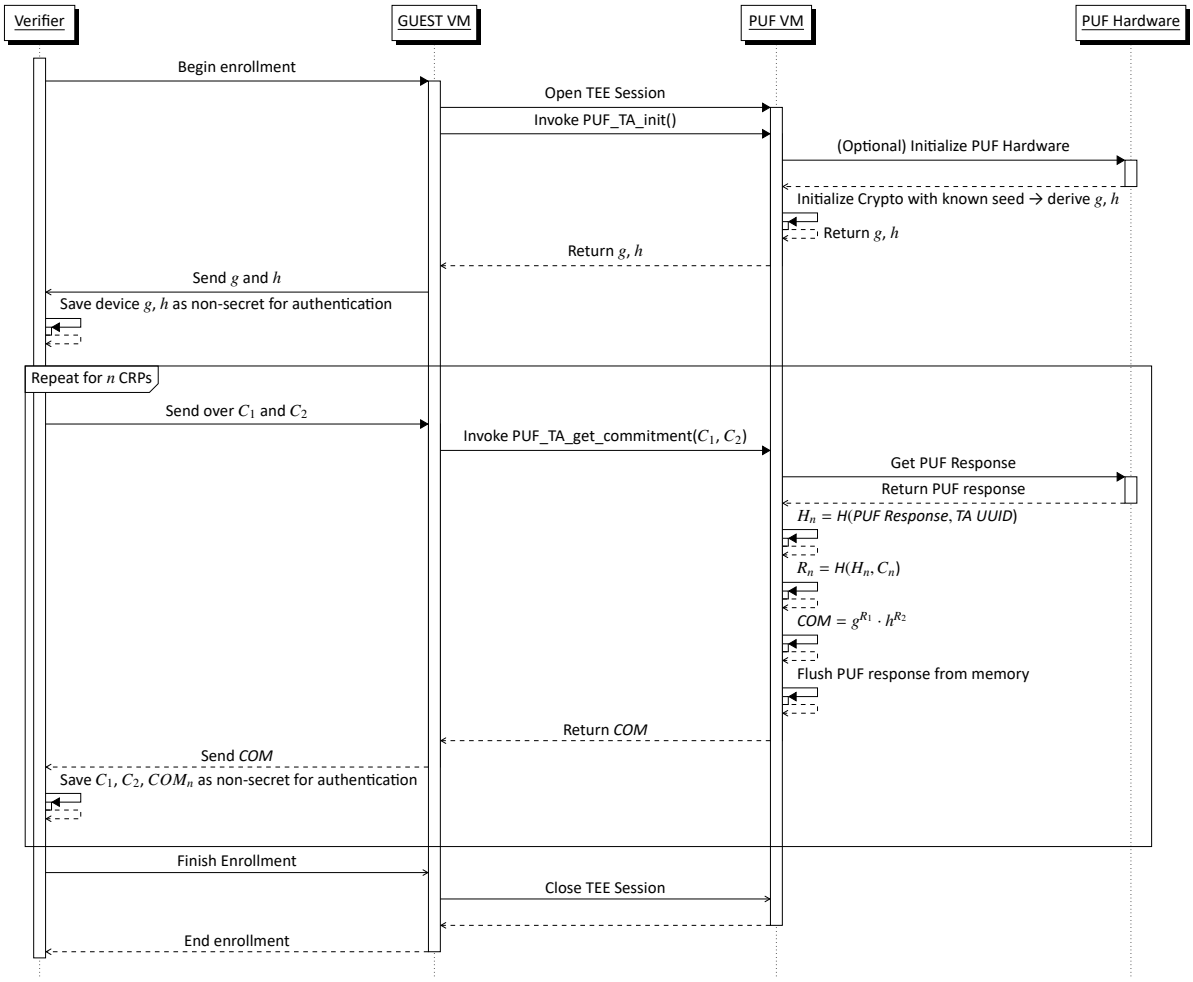


Figure 25: ZK-PUF scheme TA enrollment phase

is known to a verifier. One exemplary UC is the process of firmware updates by ensuring that the environment can be considered secure.

3.3.2.1 Background

In an enrollment phase a device can register the location's fingerprint, which is being compared in terms of similarity to consecutively generated fingerprints in the subsequent verification phase.

For the purpose of fingerprinting, we utilize RFF and target the Wi-Fi protocol as it is a well established protocol for IoT devices [169]. RFF is a technique that exploits hardware imperfections that occur during the manufacturing process in wireless transmitting hardware, resulting in distinctive patterns on the physical layer during transmissions or responses to incoming signals [170].

Conventional device identification methods often rely on cryptographic schemes that share a common secret for challenge-response protocols or utilize software-defined device identifiers such as IP or MAC addresses. In the context of IoT , cryptographic approaches may impose a significant overhead, resulting in impracticability due to resource constraints of devices. Moreover, software-defined identifiers can be manipulated easily or spoofed, making them unsuitable for security-critical operations like authentication. RFF, however, offers a promising solution to address these challenges because it relies on the unique and inherent imperfections in hardware components that occur during the manufacturing process. These imperfections can, e.g., include power amplifier fluctuations, mixer imbalances, or oscillator variations [171].

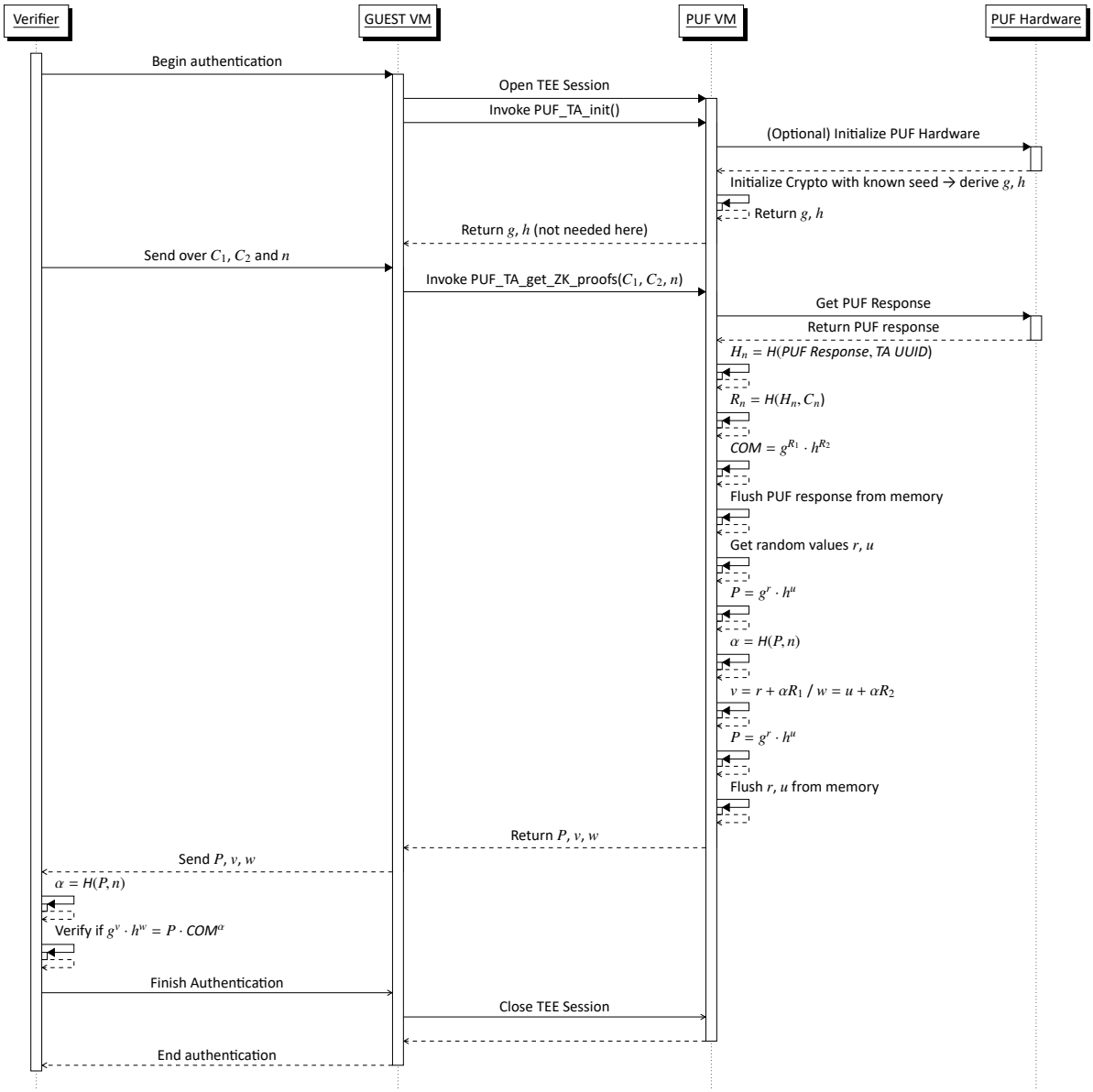


Figure 26: ZK-PUF scheme TA authentication phase

During transmission, the signal is being influenced by the aforementioned hardware imperfections, allowing a receiver to passively listen and analyze incoming transmissions to differentiate the origin of the signals. Hence, these imperfections can be assigned to a unique digital fingerprint, therefore, enabling the identification of devices or its type, which is being referred to as RFF Identification (RFFI) [171]. For instance, deployment use cases are various such as intrusion detection on the network level or localization-based techniques, e.g., for estimating the position of a device. A receiver such as a gateway or router can fingerprint the properties of surrounding devices to detect rogue or unauthorized devices, therefore are unknown devices, based on their transmission characteristics, or estimate the relative distance of identified devices [172][173][174][175].

To achieve this, a receiver passively captures transmissions and analyzes them for the specific characteristics and patterns. As no overhead is induced to the transmitting devices that can simply continue with their intended functions, this approach is well suited for IoT . Further, ML algorithms, specifically deep learning approaches, can facilitate automation by learning to find inherent patterns in radio frequency fingerprints. Thus, deep learning models may be well-suitable in the analysis of subtle fluctuations, hence eliminating the need for labor-intensive manual feature engineering and enabling learning from

raw data [171].

As we deal with wireless transmissions, which rely on electromagnetic waves to carry information that is being created by devices and modulated onto a carrier wave, which is a signal wave used to physically convey information, we utilize the physical properties of signal waves. This modulation process involves adjusting properties like amplitude, frequency, or phase to encode information onto the carrier wave. Amplitude represents the strength of the signal, frequency is measured in Hertz (Hz) as the number of cycles per second, and phase refers to the position of a wave at a specific point in time on its waveform cycle. Subsequently, an antenna converts the electrical signal into an electromagnetic wave that radiates into the surrounding space. After the signal has propagated through the medium, a receiving device demodulates the transmitted wave to recover the encoded data.

Since Wi-Fi is a multi-party communication protocol that is required to accommodate a varying number of participants, the available frequency band is split into several channels, therefore, each channel represents different frequencies. Additionally, Wi-Fi utilizes a modulation technique known as Orthogonal Frequency Division Multiplexing (OFDM), which is a method to divide the frequency band into a large number of closely spaced subwave carriers. Each sub-wave carrier operates at a specific frequency and is orthogonal to others. The advantage of this approach is the enhanced resistance to noise, as transmissions would be disturbed on only some of the sub-wave carriers, affecting sub-bandwidth instead of the entire band [176]. Therefore, the physical properties of a channel can be analyzed, e.g., for the improvement of the channel quality.

Channel State Information (CSI) encompasses such information about the physical state of a wireless communication channel between a sender and receiver, offering the potential to enhance RFF techniques with detailed information to improve the quality of available information. CSI includes details such as the phase and amplitude of the received signal across multiple sub-wave carriers within a Wi-Fi channel. By analyzing these sub-wave carriers, a comprehensive understanding of how the signal is impacted by various environmental factors, including interference, fading, attenuation, distortion, reflections, and fluctuations in transmitting power, can be concluded. This information includes changes in the transmission channel over time by consideration of a timeline of measurements [176]. This enables the development of applications such as localization, indoor tracking, and gesture recognition by utilizing predictions of signal propagation in complex environments. Consequently, CSI facilitates the adaptation of a transmission channel to the environment, enhancing the reliability of communication.

We focus on utilization of CSI for the purpose of RFF. An excerpt of characteristics contained in CSI that may be useful for our trusted service are as follows [177]:

- **Amplitude and Phase:** CSI provides insights into changes in both amplitude and phase. Amplitude refers to the signal's strength when it reaches the receiver, influenced by factors such as distance from the transmitter, environmental obstacles, or interference. The phase may undergo shifts due to reflections and delays caused by the environment.
- **Multi-path effects:** Multi-path effects describe the phenomenon where wireless signals travel multiple paths to reach the receiver, resulting from reflection, diffraction around obstacles, or scattering. This results in the receiver sensing the originally same signal under varying conditions, with each path having distinct propagation delay, phase shift, and attenuation.
- **Channel Impulse Response (CIR):** CIR refers to a short, high-amplitude probing impulse signal that may unveil information about the delays and strengths of multi-path propagation. Furthermore, it provides insights into how the channel is affected over time by measuring the CIR at distinct temporal moments.
- **Channel State Matrix (H-matrix):** The H-matrix is a representation of channel conditions between the transmitter and receiver, considering different sub-wave carriers. It depicts the relationship of signals across the frequency spectrum, including sub-wave carrier variations over time due to environmental

factors mentioned in the points above. Different sub-wave carriers may be affected differently based on the channel conditions.

CSI information can be obtained by analyzing the physical layer of wireless transmissions. Moreover, so-called pilot signals, which are also known as reference signals, can be sent by a transmitter to the receiver. The receiver compares the anticipated reference signal to the received ones and, therefore, is able to estimate the characteristics of the communication channel. Based on this information, the receiver can report the current state of the channel to the transmitter. Therefore, CSI may be leveraged for enhanced device fingerprinting.

3.3.2.2 Threat Model

We assume our attacker to be of remote nature and, therefore, being unable to forge the fingerprint to deceive the verifier due to the unique environment resulting from the intrinsic characteristics of the transmitters around the benign device, which is placed inside the predetermined location. Further, also the layout of the room in combination with the transmitters is unique. Ultimately, an attacker must resort to brute-forcing the fingerprint and guessing a similar fingerprint.
While existing work leveraging CSI as an RFF technique primarily focus on fingerprinting the identity of a device or localizing a device within its environment, we adopt a different perspective, aiming to fingerprint the environment of a device and consider the layout of a location, including other transmitters.

3.3.2.3 Our Approach

Given the goal of context-based authentication service to verify whether the receiver resides in a familiar and secure environment, we compare the digital environmental fingerprints generated during enrollment and authentication. This comparison is conducted and verified by the external verifier. To evaluate the similarity of fingerprints and to leverage the advantages of deep learning techniques for assessing the similarity between two samples, we plan to employ a Siamese network. A Siamese network uses deep learning models in a twin fashion, with both networks sharing the architecture and parameters to achieve a unified understanding of evaluating the similarity of two fingerprints. The similarity or dissimilarity between two samples is measured as a distance, for instance, as Euclidean distance. Therefore, the network aims to minimize the distance between similar pairs while maximizing the distance between dissimilar pairs during training [178]. In our context, this means that a receiver captures and collects CSI information from the transmitters belonging to the known environmental context to generate a fingerprint to prove that the receiver is placed within the predetermined environment because fingerprints from the same location should be similar.

Based on these objectives, our approach of providing authentication to be location-bound is to utilize the TA to verify the similarity of fingerprints being collected during enrollment and verification phase. For this purpose, a receiver collects environmental CSI data for the proof.
We assume the transmitters to be connected to a common access point that is part of the location, which is a reasonable assumption as IoT devices are oftentimes connected to a common gateway. Further, the receiver has to enable monitor mode in order to arbitrarily capture all transmissions within a Wi-Fi channel. Therefore, the receiver can listen to all transmissions of the network established by the access point. After the CSI collection process, the receiver sends the data which act as the fingerprint embedding the transmitters characteristics and location layout resulting in signal disturbance such as deflections, scattering, or obstruction, to the verifier over a secure channel. The verifier runs the Siamese network to assess the similarity of fingerprints between enrollment phase and verification phase.

3.3.2.4 Implementation

The implementation consists of two parts: One part is the trusted service as it is running on a device with the CROSSCON Hypervisor. The other part is the verifier, which is implemented as a remote service running on a separate machine. Communication between the two parts is facilitated over a secure channel.

Trusted Service on the Hypervisor:

Our implementation targets the Raspberry Pi 4 platform with the CROSSCON Hypervisor and with OP-TEE OS as the TEE running in a VM. The trusted service itself is implemented as a TA running in OP-TEE, which is called from the REE.

The Raspberry Pi's Wi-Fi chipset does not disclose CSI data by default. However, a firmware modification exists which makes the CSI samples measured by the Wi-Fi hardware accessible by code. For a detailed description of how the hardware is accessed from within OP-TEE OS, refer to Deliverable 4.3. TAs can request the collection of CSI samples via a pseudo Trusted Application (pTA) added to the OP-TEE OS Core. This works similar to calling any other TA. Since CSI data is captured for a specific Wi-Fi channel and with a given channel bandwidth only, these configuration options are passed to the pTA alongside a maximum collection duration and a maximum number of samples collected per device. Furthermore, a MAC address filter can be applied during the collection process, which only collects samples from certain devices.

The trusted service is implemented as a TA in the TEE. It uses the pTA described above to collect the CSI samples, and communicates with a remote server via the network. The network communication uses the TCP functionality built into OP-TEE OS. The parameters required for CSI data collection, such as Wi-Fi channel, channel bandwidth, collection duration and samples per device, are configured at compile time. The MAC filter is disabled by default.

OP-TEE uses the REE's networking stack to establish a connection, and, as a result, a networking device must be attached to the VM which invokes the trusted service. Because of how the CSI collection process is implemented, this can not be the on-board Wi-Fi chipset, so we use the Pi's Ethernet interface for that. The connection uses the TCP protocol, which does not provide any security features by itself. Because the communication between the TA and the remote verifier must occur via a secure channel, the TCP socket is wrapped with a mutual TLS (mTLS) connection inside the TA. For this, the server and the client must own a certificate for establishing the connection. The server certificate is provided to the TA at compile time, and the client certificate is generated inside the TA during enrollment (see below) and saved in the TEE-provided secure storage for subsequent usage.

Remote Verifier:

The second part of the implementation is the remote verifier. It runs on a separate device, and is implemented in Python. The verifier has two main tasks: It receives CSI samples from the TA and decides if the authentication is valid. In addition, it participates in the certificate enrollment process by verifying the device's identity and signing the certificate.

The remote verifier compares two environmental fingerprints on a per-device basis. For each device, a series of CSI samples are collected and compared with a machine learning model. The model takes two inputs and consists of two main components: the encoder and the discriminator. Each input is first passed through the encoder, which generates an embedding vector of length 52. The absolute difference between the two embedding vectors is then fed into the discriminator, which produces a value between 0 and 1, representing the probability that the two inputs originate from the same device. The input series must consist of a certain number of samples. If a given device does not produce a sufficient amount of traffic, it is excluded from the environment.

The encoder follows a convolutional neural network (CNN) architecture and comprises four convolu-

tional layers, each followed by a Tanh activation function. A max pooling after the second CNN layer is used to reduce the input dimensionality and three dropout layers are used to help prevent overfitting. The CNN part of the model resembles the Siamese network, and the samples from the two devices are passed through it independently.

The discriminator is a single layer fully connected network that takes the two embeddings from the encoder and outputs a single value between 0 and 1 by using a sigmoid activation function.

The verifier must be configured with three cryptographic key pairs, which are used during various steps of the process's lifetime. The communication is secured with mTLS, so the verifier needs a server certificate. During enrollment, the TA enrolls a client certificate, for which the verifier is the certificate authority. For authentication, the TA provides a nonce alongside the gathered environment fingerprint. If authentication succeeds, this nonce is signed by the verifier and the signature is returned to the TA. These three keys must be configured for the verifier, and the public parts of the mTLS key and nonce signing key must be provided to the TA at compile time.

Enrollment:

Before authentication can be attempted, a device must be enrolled first. Enrollment is split into two steps: During the first step, the TA generates keys for the secure communication channel between it and the remote verifier. In the second step, a baseline fingerprint is collected from the environment and sent to the remote verifier, where it is stored for later verification.

The communication channel between TA and remote verifier is secured with mTLS. When using mTLS, both the server and the client are authenticated with a certificate. The server certificate is provided to the TA at compile time, and the TA's client certificate is generated during the enrollment. First, a Certificate Signing Request (CSR) is generated by the TA. This CSR contains a device ID (for simplification reasons replaced by a random number in this implementation) and is sent to the remote verifier via a reugular TLS connection, secured by the server certificate only. Once the verifier confirmed the device's identity, it signs the CSR and returns a valid client certificate to the TA (the identity confirmation step is skipped in our implementation). The TA stores the client certificate in the TEE secure storage. This certificate contains the device ID from the CSR, and the server can use it to map the certificate to the device.

After successful certificate enrollment, the TA collects the enrollment fingerprint. For that, it invokes the pTA and requests the collection of CSI samples with the parameters provided during compilation. The MAC filter is disabled. Then, it sends the collected samples to the remote verifier.

The remote verifier receives the CSI samples from the device and verifies the format. Since the machine learning model requires a minimum number of samples from each device, the samples are sorted by device and filtered based on the number of available samples. Samples from the devices which meet the threshold are saved for later authentication attempts, and the list of devices is returned to the TA. The TA saves this list for use during authentication. No machine learning is invoked during the enrollment.

Authentication:

The authentication process is started when an application from the REE invokes the TA. It must specify a nonce as a parameter, which is used at a later step during the authentication process. This nonce must be unique for every authentication attempt to prevent replay attacks. It can be generated with help of the TA, but also by a remote party which requests the authentication.

Upon invocation, the TA requests the collection of CSI samples via the pTA. The parameters are, again, specified during compile time. Unlike during enrollment, at this point the MAC filter is enabled based on the list of devices returned by the verifier during the enrollment process. This reduces the volume of data transmitted to the remote verifier, and, since only the devices with saved MAC addresses are

enrolled at the remote verifier, only those devices can be validated as part of the environment.

Once the collection process is completed, the samples are forwarded to the remote verifier together with the nonce. Similar to enrollment, the samples are sorted by devices and filtered with a threshold. Then, devices with a corresponding dataset created during enrollment are compared using the machine learning model. The authentication attempt is considered successful if sufficiently many devices in the current environment match their enrolled fingerprints. This threshold is provided to the verifier as a configuration option. Upon a successful authentication attempt, the verifier signs the nonce sent alongside the CSI data, and sends the signature back to the TA which, in turn, forwards it to the calling application.

The REE application can verify the validity of the signature using the TA, where the public part of the signing key is enrolled. This verification can occur on the same device, or the application can forward the signed nonce to a remote party to signal successful authentication. The nonce's uniqueness is of paramount importance, as reusing a nonce enables a replay attack on the signature.

3.3.2.5 GlobalPlatform Perspective

The implementation of this service uses OP-TEE OS as a TEE. Because OP-TEE is compliant with GP [31], all interactions between REE and TEE as well as the TA's and pTA's lifecycle management also adhere to the GP specification. For a more detailed view on GP compatibility regarding the hardware access, please see D4.3.

3.3.3 Remote Attestation

Remote attestation is a basic security mechanism designed to allow a device or a system to verify the integrity and authenticity of another remote entity. The fundamental concept behind remote attestation is to enable the verification of the software state of a remote system, ensuring that it has not been compromised. This is particularly crucial in environments where trustworthiness and security are paramount, such as in cloud computing, IoT ecosystems, and distributed networks. At the heart of remote attestation is the exchange of evidence between the device being attested (the prover) and the entity seeking assurance of the device's integrity (the verifier). The prover generates and sends a summary of its current state called evidence. This evidence is typically generated by measuring the devices state (e.g., hashing the content of the device memory or tracing the programs execution).

3.3.3.1 Threat Model

In the context of CROSSCON the adversary can inject malicious code and has full control over the system software. Further, the attacker can tamper the Control-Flow of a software through control-data and non-control data attacks (e.g., ROP and DOP attacks) [179, 180]. Therefore, our Attestation framework finds its application in attesting vulnerable/security-relevant application within CROSSCON stack.

3.3.3.2 Our Approach

Within CROSSCON, we implement a Remote Attestation scheme where an entire VM running on the CROSSCON Hypervisor can be attested at runtime. The attestation process itself is started from another VM, also running on the Hypervisor, which can then take actions to correct the non-conforming VM's behavior, e.g. by restarting it. The verification of the proof generated on the device is performed by a remote verifier, and the security-critical operations on the board take place in a TEE.

This attestation scheme thus requires the following parties: A VM running on the hypervisor, which initiates the attestation request, is called VM A. It requests the attestation of an untrusted VM-under-test, called VM B. The role of the prover is occupied by the TA running the Remote Attestation's trusted service, and the verifier is located on a remote device.

The attestation flow is as follows: VM A initiates the process by invoking the Remote Attestation's trusted service's TA to attest the integrity of VM B. The trusted service then collects attestation evidence from VM B through the CROSSCON hypervisor. Once the necessary proof is gathered, the trusted service signs it and transmits the signed attestation to the remote verifier over a secure communication channel. The remote verifier evaluates the validity of the received evidence and returns a signed verdict to the trusted service, indicating whether the attestation is accepted or rejected. The trusted service then forwards this response back to VM A, which can independently verify the signature to confirm the authenticity and integrity of the verifier's decision and to take further action.

In this Remote Attestation scheme, the attestation proof is derived by computing a cryptographic hash over a specific region of the VM B's memory. This region is identified at runtime based on a unique starting byte pattern and a predefined length, enabling selective and context-aware measurement of memory content. The remote verifier, having received the expected memory contents through an out-of-band channel, can recompute the hash independently and compare it to the received proof.

This remote attestation scheme enables the measurement of arbitrary memory regions within VM B. By leveraging a flexible mechanism for selecting memory segments, any portion of VM B's memory can be targeted for attestation. This design allows fine-grained verification of security-critical data or code, without requiring modifications to the B's software stack or prior knowledge of fixed memory layouts.

A key feature of the scheme is that the VM-under-test remains unaware of the attestation process. Memory measurement is conducted transparently via the CROSSCON hypervisor, which operates below the VM's execution layer. This ensures that the integrity checks cannot be subverted or influenced by a potentially compromised VM B, and that attestation can occur without cooperation or instrumentation within the VM itself.

3.3.3.3 Implementation

The Remote Attestation scheme is implemented on a Raspberry Pi 4 platform as part of the CROSSCON stack, leveraging the CROSSCON Hypervisor in conjunction with OP-TEE OS as the TEE running as a VM. For attestation purposes, at least two additional VMs are required, where A initiates the attestation process and B serves as the VM-under-test.

As the trusted service is implemented inside the TEE, OP-TEE requires access to the complete memory space of B. The CROSSCON Hypervisor's Inter-Partition Communication (IPC) mechanism can not be used for this purpose, because this would require mapping B's memory region as both a memory region and an IPC to it. Thus, the hypervisor is modified to allow the mapping of a single memory segment to multiple VMs, thereby enabling the trusted service to collect the required evidence about B. This bypasses the IPC mechanism all together.

TAs running within OP-TEE OS can not read or write physical memory directly because of hardware security primitives managed by OP-TEE, and OP-TEE does not provide direct memory access mechanisms for TAs. However, the trusted service running inside OP-TEE, in the role of the prover, must access VM B's memory in order to generate the proof required for attestation. This issue is addressed by adding a pTA to OP-TEE OS. pTAs are part of the OP-TEE OS and are added at compile time. They run in kernel mode, can read and write physical memory locations identified by their address, and can be called by other TAs. As a result, the attestation service on the device is split into two parts, a TA and a pTA: The pTA reads B's memory to find the supplied byte pattern and, once found, computes the has over the requested number of bytes. It is called by the TA, which forwards the collected hash securely to the remote verifier and receives the attestation result. The TA is invoked by A, to which it also returns the attestation result.

The secure communication between the trusted service and the remote verifier works similar to the scheme described above for Context-based Authentication. mTLS is used to secure the connection, with

the verifier's server certficate provided at compile time and the client-side certificate generated during the enrollment phase. See the relevant section above for more details.

The remote verifier is implemented as a separate application and runs on a separate device on the network, responsible for validating the integrity of VM B's memory based on attestation data received from the TA. Prior to the attestation process, the verifier is provisioned with the expected memory content that should reside in VM B. Using this reference data, the verifier computes its hash value, which serves as the baseline for verification. When the TA establishes a connection via the mTLS channel and transmits the attestation proof, the verifier compares the received hash against its pre-computed value. If the two hashes match, the attestation is considered successful; otherwise, it is deemed a failure. The result of this verification is then signed using a dedicated private key held by the verifier. This signed response is returned to the TA and can subsequently be validated by VM A, either directly or with the assistance of the TA. The corresponding public key required for signature verification is embedded in the TA at compile time, ensuring that only responses from authorized verifiers are accepted.

It is important to note that VMs running on the CROSSCON Hypervisor can have a wide range of complexity, from a bare-metal application to a complete OS, e.g. based on the Linux kernel. The attestation scheme makes no assumptions about the complexity of the VM running, thus possible memory abstractions are not taken into account. It lies in the responsibility of the user to make sure that the sequence of bytes, which is expected in memory, is available as one continuous sequence and not separated into multiple parts and spread across the VM's memory.

3.3.3.4 GlobalPlatform Perspective

The implementation of this service uses OP-TEE OS as a TEE. Because OP-TEE is compliant with GP [31], all interactions between REE and TEE as well as the TA's and pTA's lifecycle management also adhere to the GP specification.

3.3.4 FPGA Related Trusted Services

Secure FPGA provisioning supports the secure operation of FPGA-enabled CROSSCON devices, primarily enabling trusted execution of compute-intensive or even general-purpose computing tasks on the FPGA. Given the propensity for users to incorporate sensitive information and proprietary code/circuitry, safeguarding their IP is paramount. Secure FPGA provisioning service maintains the security of FPGA configuration files, known as, bitstreams, throughout provisioning and configuration, thereby preventing unauthorised access to the user's code or data. This assurance instills confidence in users concerning the protection of their IPs. On the other hand, proactive measures should be taken to fortify the device against potential threats stemming from malicious circuits uploaded by users, including workloads capable of espionage, disruption of concurrent processes on the same FPGA, or even causing physical damage to the FPGA hardware.

In the fast-paced evolution of computing environments, including cloud and traditional setups, incorporating virtualization technologies is crucial. Consequently, FPGA virtualization schemes often propose to abstract FPGA resources into a pre-defined pool of coarse-grain regions; each can be allocated to a different user. This shift introduces the concept of FPGA spatial multi-tenancy, where the physical FPGA architecture is partitioned into logically isolated regions. Each region can be configured or reconfigured independently by leveraging modern FPGAs' partial reconfiguration capabilities. Despite their dedicated FPGA resources, these partially reconfigurable regions share underlying clock and power distribution networks. For clarity, we refer to these logically isolated regions as vFPGAs and the bitstreams designed for these vFPGAs are called *partial bitstreams*.

In this section, we present the *trusted services* that are primarily required to enable secure FPGA provisioning. We look at the four key trusted services relevant to Secure FPGA Provisioning: the *vFPGA*

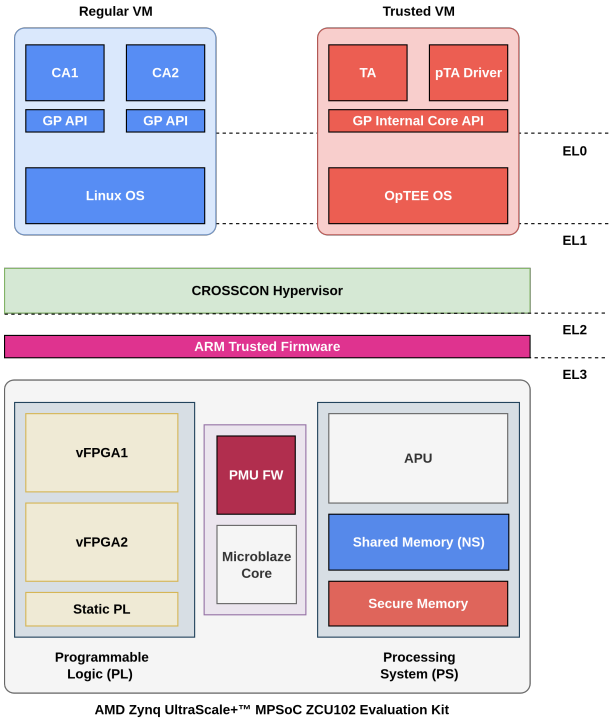


Figure 27: CROSSCON Stack with FPGA Trusted Application (TA).

Client Registry Service, the *vFPGA Configuration Service*, the *vFPGA Deallocation Service*, and the *vFPGA Management Service*.

3.3.4.1 Underlying Architecture

Partial reconfiguration, a fundamental concept in FPGA technology, refers to the capability of dynamically reconfiguring regions of the FPGA while the remainder of the logic continues to function seamlessly. A physical FPGA device comprising various configurable resources can be logically partitioned into n partially reconfigurable regions, referred to as virtual FPGA (vFPGA), with n being determined by the system administrator. For instance, Figure 27 demonstrates two vFPGAs.

Table 11 describes the responsibilities and capabilities of different components involved in realizing secure FPGA provisioning. These components are: Client Applications (CAs) running on Regular VM, TA & pTA on the Trusted VM, CROSSCON Hypervisor (Hypv.) hosting the two VMs, ARM Trusted Firmware (TF-A) running at the EL3 privilege level, and entities on the FPGA board. In our description, we are using an AMD Zynq Ultrascale+ MPSoC ZCU 102 Board, which has a PS and a PL part. The key components of the PS part are: Application Processing Unit (APU), consisting of ARM Cortex-A53 processors and memory partitioned into shared and secure regions by the CROSSCON Hypervisor. The PL part includes vFPGAs and a static PL component that supports the vFPGAs. Further, there is a Platform Management Unit, which is responsible for configuration after booting, and the corresponding Platform Management Unit Firmware (PMU FW) runs on a separate Microblaze core.

3.3.4.2 vFPGA Client Registry Service

This service is critical to establishing a link between a client application and the TA. It includes two fundamental requirements for secure FPGA provisioning as shown in Figure 28. The first is the registration of the client with the TA using an identifier. The second is the exchange of public keys between the client and TA. This key exchange is indispensable for the vFPGA configuration service, which involves cryptographic operations on the partial bitstream.

Table 11: Key components of the architecture along with their responsibilities and capabilities.

Component	Responsibilities	Capabilities
Client Applications (CAs)	- User/Client applications. - Request different trusted services offered by the TA.	- Full access to the normal-world file system.
Trusted Application (TA)	- Expose trusted services to CAs.	- Access to secure memory, cryptographic HW/TEE services. - Can invoke sessions to pTA.
pseudo TA (pTA)	- Bridge between TA and TF-A. - Invoke calls to TF-A.	- Runs in the kernel context of OP-TEE. - Access to secure memory.
CROSSCON Hypervisor (Hypv.)	- Partition PS resources between Linux VM and OP-TEE VM. - Memory management. Bridge between TA and TF-A.	- EL2 privilege over both VMs.
ARM Trusted Firmware (TF-A)	- Handle invocation from pTA. - Run at EL3; decide if reconfiguration is permitted. - Call PMU services through PMU-FW wrapper.	- Full secure-monitor access; can reconfigure PL via PMU. - Code is ROM-resident; updates require re-flash.
Platform Management Unit Firmware (PMU FW)	- Load partial bitstream to vFPGA.	- Runs on a dedicated MicroBlaze core in the PMU. - Full control over platform management registers.

Trigger: A CA that is unknown to the TA can request this service to register itself with the TA.

Parameters: (1) CA's public key to be used for authentication in the vFPGA configuration service by the TA. (2) A buffer to receive the TA's public key. (3) CA's identifier.

Result: The TA returns its own public key to the CA. For unknown clients, a new entry is created in the registry with the CA's identifier and public key.

3.3.4.3 vFPGA Configuration Service

This service aims to securely deliver and configure the user's designed partial bitstream onto the vFPGA. Each vFPGA can be managed and configured separately. User CAs, intending to deploy their IP designs, must meet the defined physical constraints to place their designs within the boundaries of vFPGAs. However, users also have the flexibility to provide multiple instances of a task, each targeting a different vFPGA. This flexibility enhances the likelihood of successful deployment on the FPGA. Figure 29 describes the typical workflow of the vFPGA Configuration Service.

Trigger: A known CA (now with an entry in the client registry) can invoke the vFPGA configuration service to configure a partial bitstream on a free vFPGA.

Parameters: (1) A package with an encrypted partial bitstream, signature, wrapped key for encryption, and necessary metadata. The wrapped key can only be unwrapped by the TA to decrypt the btistream after performing authentication checks (Figure 29). (2) Valid scalar value selecting the vFPGA.

Result: On success, the partial bitstream is loaded into the matching reconfigurable partition. The TA returns a successful acknowledgement to the CA, and the status update path of the vFPGA management service is called to mark the CA as the owner of the corresponding vFPGA.

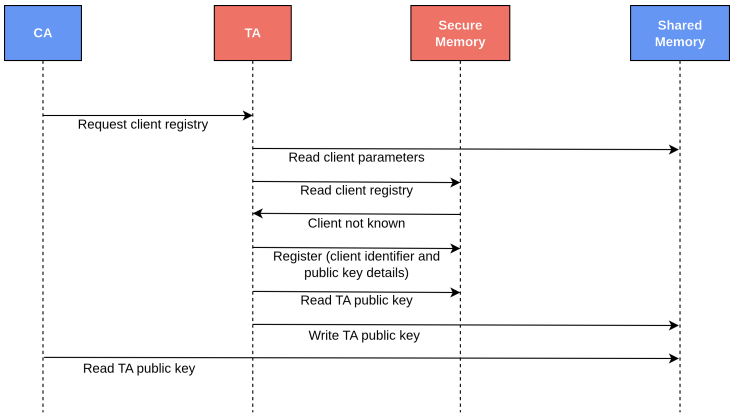


Figure 28: vFPGA Client Registry Service Workflow.

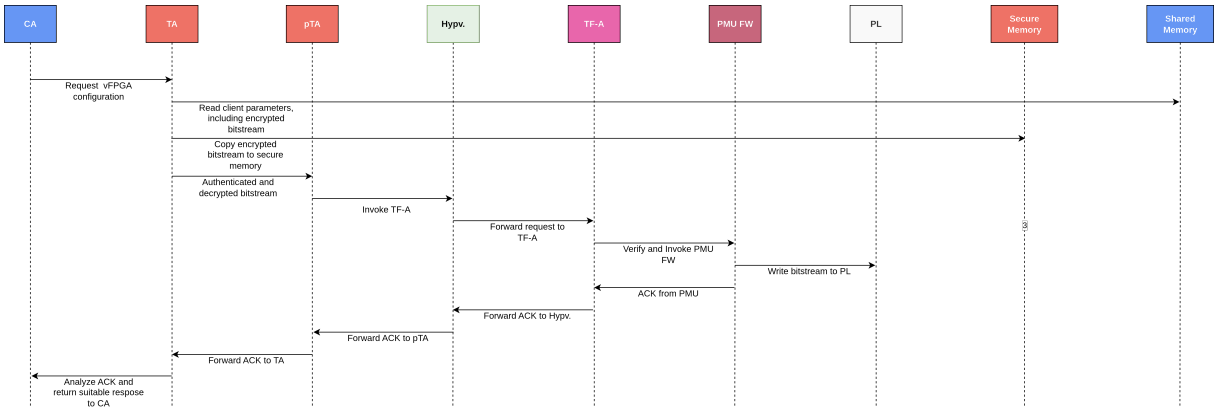


Figure 29: vFPGA Configuration Service Workflow.

Pre-conditions for success: (1) The CA is known to the client registry. (2) Requested vFPGA is valid and currently free. (3) CA passes the ownership check of the package. (4) Package passes the expected sanity checks. (5) RSA signature over the wrapped encryption key verifies successfully. (6) Encryption authentication verifies successfully.

3.3.4.4 vFPGA Deallocation Service

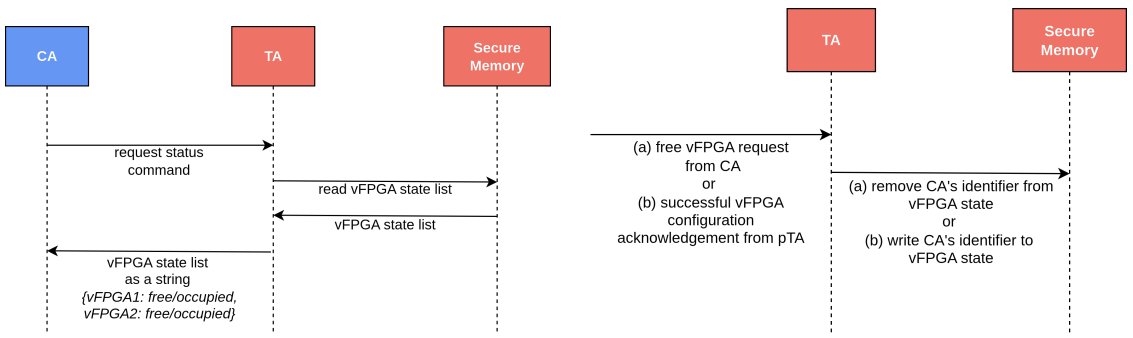
This service enables users to deallocate the vFPGA after configuration. This ensures efficient resource utilization and effective management of FPGA resources. Upon vFPGA deallocation, a special default partial bitstream is configured to erase the previous configuration, and the status of the vFPGA is marked free through the vFPGA management service.

Trigger: The CA invokes the vFPGA deallocation service when it no longer needs exclusive access to a vFPGA slot.

Parameters: (1) Any package previously produced by the CA (the TA uses only it to verify the caller is authentic). (2) Scalar value identifying the vFPGA slot to be released.

Result: The corresponding entry in the vFPGA state maintained in the secure memory is cleared, and a special default partial bitstream is configured on the corresponding vFPGA. Another CA or the same CA again may now claim the vFPGA.

Pre-conditions for success: (1)Package is well-formed. (2) vFPGA-ID is valid. (3) The CA passes the ownership verification of the package.



(a) vFPGA Status Check (b) vFPGA Status Update

Figure 30: vFPGA Management Service Workflow.

3.3.4.5 vFPGA Management Service

This service keeps track of which client owns which reconfigurable partition and offers two stateless operations: a *vFPGA status check* for monitoring and a *vFPGA status update* used by the vFPGA configuration and free actions. Figure 30a and Figure 30b.

vFPGA Status Check

Trigger: The status check path in vFPGA Management Service can be initiated directly by the client application.

Parameters: Status check does not require any parameters; the invocation presents the state of both the vFPGAs.

Result: If successful, the service results in a string with the state of two vFPGAs, accessible to the caller CA.

Pre-conditions for success: The status check has minimal authentication done by the TA. The only point of failure is a malformed invocation by the TA.

vFPGA Status Update

Trigger: The vFPGA status update path is *not* invoked directly by a client but automatically by the TA when the state of a vFPGA changes from free to occupied or vice-versa. This happens in two situations: (i) immediately after a successful vFPGA configuration and (ii) after a successful vFPGA deallocation.

Parameters: None. The TA operates purely on its vFPGA state list present in the secure memory.

Result: In case of a vFPGA configuration, the TA copies the caller CA's identifier to the corresponding vFPGA state in secure memory, marking the slot as *occupied*. In case a CA frees the vFPGA, the TA updates the state of vFPGA in the secure memory, marking it *free*. The modification becomes visible to any subsequent status check query through the vFPGA management service.

3.3.5 Behavioral-Based Trusted Service

Behaviour-based security is a method in which all relevant activities of a device are monitored to identify any deviations from normal behavioural patterns [181]. Given the scope of the project on the security of connected devices, this service is defined on the behaviour of the device in terms of its network traffic generated as a result of its operation and communications with other devices or entities.

In the last few years, ML and its sub-field of Tiny Machine Learning (TinyML) [182] have shown significant advances and improvements in terms of algorithms and scalability to constrained environments [183].

As such, TinyML is expected to play an important role in security of computing environments at the edge of IoT networks [184].

We defined the behavioural-based trusted service as a *network anomaly detection* service that:

- Runs in an isolated environment from the rest of the applications and services on the device that are to be monitored for anomalies, and
- Can access (has a visibility of) all network packets from any of the device's applications and services.

Traditional intrusion detection systems are predominantly signature-based, where software monitors network traffic and compares packet data against known threat signatures [185]. In contrast, anomaly detection systems also monitor network streams (or flows), but instead of relying on predefined signatures, they compare traffic to a baseline of normal behavior to identify deviations [185].

The service adopts TinyML, specifically, a deep leaning (DL) algorithm called Autoencoder, to achieve efficient learning process of the baseline behaviour and to flag events that are statistically significant from the baseline. The aim of the Autoencoder is to learn a good representation of IoT network traffic data by applying unsupervised training [186].

3.3.5.1 Innovation Aspects

There are two main innovation aspects addressed:

- *Network telemetry* suitable for IoT protocol behavioural analysis. Ensures the necessary visibility of lower IoT network protocols and access to network traffic of a device. Furthermore, the service offers an efficient handling of network traffic features (statistics) of numerical and categorical type data suitable for ML anomaly detection.
- *Lightweight deep learning* model suitable for IoT devices. Adoption of TensorFlow lite library[3], and study different optimization and reduction techniques for DL models such as quantization and pruning[4] to scale to resource-constrained environments. The aim is to reduce the DL model size under some controlled performance reduction (e.g., accuracy, precision, etc.) but gain much more efficiency on computing in terms of inference on anomaly detection. Define suitability and limits of online on premise vs offline back-end training. The preferred choice for the service is the on premise training where the DL model stays on the trusted service's dedicated and isolated environment.

3.3.5.2 Main Functionalities

The network anomaly detection service will allow to monitor the network traffic of a device and detect any deviations from the baseline. This is a complementary view to other security services or trusted applications deployed on the device. For instance, in addition to the remote attestation or secure firmware update, the anomaly detection service will report if the traffic fingerprint of the device during such services, prior or after, results to any anomalous behaviour such as suspicious port numbers exposed by the device, or suspicious IP addresses and/or port numbers or protocols used for communications with other devices or servers.

Given that anomaly detection reports any significant deviations from the baseline, it is important not only to flag if any network traffic portion is anomalous, but also to explain why they are considered anomalous, i.e. what aspects of the network traffic are suspicious and significant for the decision of anomaly. The explainability is important to make the anomaly useful for further decision making by higher level solutions such as Security Information and Event Management (SIEM) or any risk-mitigation modules.

[3] https://www.tensorflow.org/lite

[4] https://www.tensorflow.org/lite/performance/model_optimization

There are two main modalities that form part of the behavioural-based trusted service:

- *Training modality* that trains the Autoencoder algorithm to learn the legitimate patterns of network traffic. It is necessary to specify the amount of time needed to capture legitimate traffic that the algorithm will be trained on. The duration of training is a key factor to ensure enough variety of system behaviour. It can last from few hours to days. Once the training modality is completed, the service automatically switches to the monitoring modality, also called inference or prediction.
- *Prediction modality* that monitors all incoming traffic in soft real-time for anomalies. The anomalies are stored in an event log file that can be sent to or used by any SIEM or server module for decision making.

3.3.5.3 Workflow

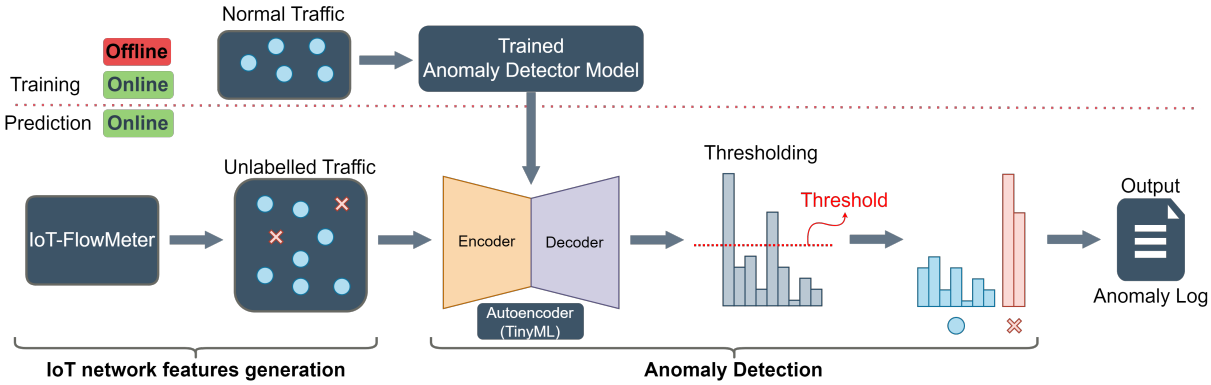


Figure 31: Behavioral-based trusted service workflow.

Figure 31 shows the high-level view of the service's workflow. The initial phase involves training deep learning models using legitimate network traffic (packets). This process can be conducted online or offline. Once the model is trained, the tool switches to monitoring or inference mode.

The first module is an IoT -FlowMeter (IFM), which feeds the Brain module with a set of network behavioural features extracted from the monitored network traffic. The IFM is based on the well-known open-source community CICFlowMeter[5] tool but customised to extract more network traffic features necessary to detect anomalies.

There is a Brain module that contains a deep learning model. It processes traffic collected by the IFM for training the Autoencoder algorithm. The Brain detects possible deviation from the pattern learned during the training phase. Through the utilization of error reconstruction, coupled with a predetermined threshold, it classifies incoming traffic into legitimate or anomalous, and provides additional explainability information to understand the nature of the anomaly. For instance, it shows what features have been the most critical for detection, and gives evidence showing the deviation of the anomaly from the normal training data.

3.3.5.4 Deep Learning Models Optimisation

The main novelty in the second half of the project was the quantization (optimisation) achieved on the deep learning models for the detection of cyber-attacks on the Edge, and performance evaluation of such models on IoT devices, such as Xilinx Arty Z7.

Particularly, we found an optimal size for the state of model parameters (float32) that allowed efficient execution of models and, at the same time, maintained a high detection rate of the attacks performed, such as False Data Injection (FDI) at various rates. Furthermore, the size of the model is a factor *several*

[5] https://github.com/ahlashkari/CICFlowMeter

times smaller in memory than the full model parameters size, and the prediction time per single flow telemetry *improved 99%*.

We evaluated the performance of our deep learning models on (i) bare-metal RISC-V implementation on Xilinx Arty Z7 board; and (ii) Linux system on ARM Cortex A76 on a Raspberry Pi board.

Figure 32: Behavioral-based trusted service DL Model Quantization Performance

Figure 32 shows the performance of DL model optimisation for the different sizes of the parameter state - float64, float32 and int8. We note the float64 is the baseline for us. A significant performance gain was achieved for float32 where we observed decrease of decision time from 300ms for float64 to 3ms for float32. This was the default model for protocols on OSI layers 2, 3, and 4. The model size based on 180 features of telemetry has also reduced from 2.04MB float64 to 0.233MB float32. Regarding the model for IEC61850 protocols (OSI layer 7) based on 480 features, the decision time dropped from 370ms to 4.4ms, while the model size from 12.5MB float64 to 1.13MB float32.

Based on these results, we are considering the use of dual models of float64 and float32 in future deployments of our technology L-ADS[6], especially in HPC, where high-volume decision-making is expected and can be switched to float32 models when needed.

We recognize that there is always a price in decreasing the size of the state of the parameters where some deviations of traffic may not be detected as in the full state, but we have not observed any false negatives for the cyber-attacks performed. All attack instances have been correctly detected by the float32 models.

Figure 33 shows the performance of the model trained for the communications of IEC61850. The attacks referred to the GOOSE and SV protocols of the IEC61850 standard. We selected the F1-Score as it interprets the harmonic mean of precision and recall, where the F1-Score reaches its best value at 1 and its worst score at 0. We found similar performance for other metrics, which also confirm the conclusions of this experiment. The most important result regarded the F1-Score with model parameters of type float32 vs. float64. We observed very similar model capacity of attack detection equivalent to the float64, the most precise model parameters. As opposed to Int8 where we achieved the worst F1-Score.

3.3.5.5 Reference Architecture for Service Provisioning

The reference architecture for the provisioning of the network anomaly detection service is depicted in Figure 34. The service is provisioned in a dedicated (memory-isolated) VM with a standard Linux distro

[6] https://booklet.evidenresearch.eu/lads

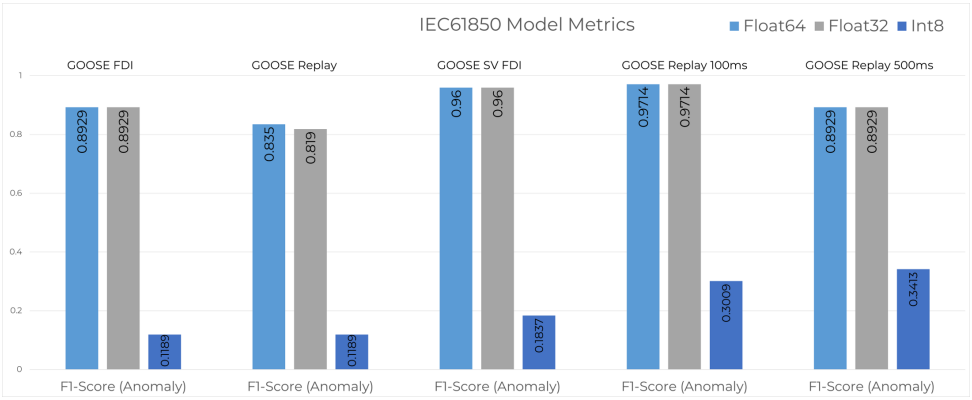


Figure 33: Behavioral-based trusted service F1-Score for False Data Injection and Packer Replay attacks, on protocols of IEC61850, and on model parameters type float64, float32, and int8.

and the VirtIO-net[7] backend (host) configured. The other VMs act as a VirtIO-net front-end.

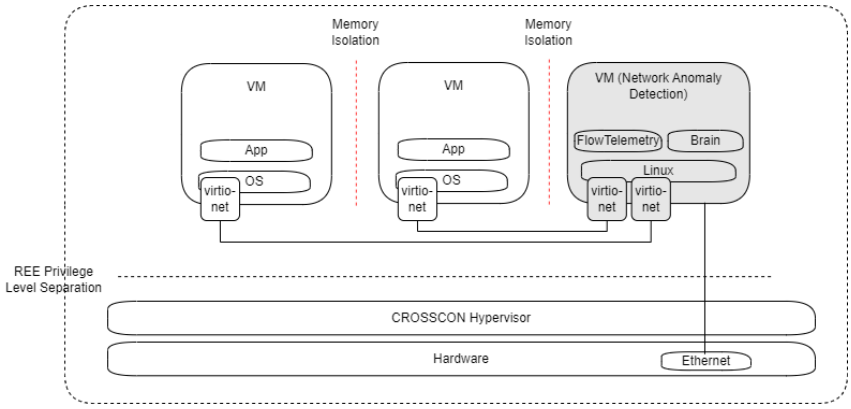


Figure 34: Behavioral-based trusted service provisioning reference architecture based on CROSSCON Hypervisor and VirtIO-networking

Such an architecture based on the VirtIO-net standard is supported by the CROSSCON hypervisor[8]. It ensures all device's traffic is visible to the service, and the anomaly detection and its DL models are separated from the normal application world. While different flavors and variations of the architecture can be considered for the provisioning and operation of the network anomaly detection service, the central role and reference of the service with respect to the VM isolation and virtIO-net standard remain.

3.3.6 Control Flow Integrity Trusted Service

In computer security, one of the strongest security guarantees for an application is control flow integrity. This property can defend the application from various run-time attacks aimed at diverting its control flow, i.e. the sequence of instructions that is executed by it. For instance, an attacker could exploit some memory vulnerabilities of the application to mount a Return Oriented Programming (ROP) attack. This attack can lead to arbitrary code execution by allowing the controlled execution of disjoint pieces of existing code. As a result, an application can be exploited to execute unintended operations.

Control flow integrity is a fundamental security property for the correct functioning of an application. For this reason, several chip manufacturers are working to create hardware solutions that can offer such a guarantee. Unfortunately, relying on hardware is not always an option, especially in the context of IoT

[7] https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html
[8] see for instance the Bao demo at https://github.com/bao-project/bao-demos/tree/feat/virtio-demo

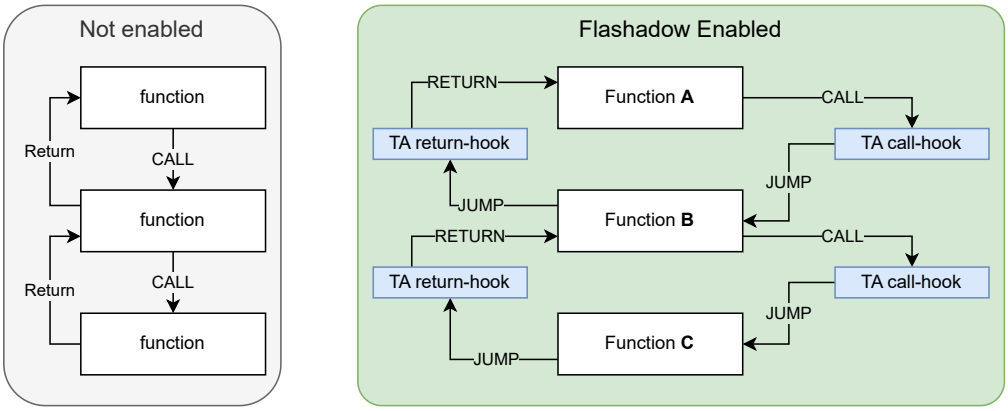


Figure 35: Difference between an insecure application and an application using Flashadow.

that counts a plethora of different devices lacking such specific hardware. With CROSSCON we propose to address this security gap by leveraging the TEE rather than specific hardware.

TEEs are often used to enable TAs: distinct security-oriented applications that offer some security service to untrusted clients. However, these TAs are independent, self-contained, and passive. A passive service (i.e., one that acts only upon request from the client) cannot properly enforce control flow integrity because it requires a more active interaction with the CA. We hereby propose two novel designs for active control flow integrity for the bare-metal TEE (described in 3.5 below). Our contribution is twofold: we propose Flashadow for the non-MPU bare-metal TEE (for Class 0 devices, exemplified by the MSP430 architecture) and uIPS for the MPU bare-metal TEE (for Class 1 devices, exemplified by the ARMv7-M platforms).

Flashadow [187] is focused on protecting the backward edges of the control flow, i.e. it ensures that each function returns to the point in the code where it was called. To accomplish this, Flashadow performs two main operations: (i) upon each function call from the application, it keeps track of the call instruction (our return site) on a separate shadow stack, and (ii) upon each return statement, it makes sure that the return site is indeed the one registered on the shadow stack. Together, these two operations guarantee that the backward edges of the control flow are protected from attacks. Notably, our approach is software based, for it does not require any hardware capability on the device. To achieve so, contrarily to common TAs, Flashadow must be enabled at compile time by using a dedicated toolchain that creates compatible code.

Call instrumentation The first requirement for Flashadow is the instrumentation of the call instructions to keep track of the legitimate return sites. To do so, Flashadow replaces each call instruction in the application code with a new hook that transfers control to the TA. These new instructions allow the TA to read the program counter (PC) to infer the return site and save it in a secure storage (enabled by the TEE).

Return instrumentation Instead of trusting the stack, which could have been compromised by an attacker, the application needs to transfer control to the TA. Having kept track of all of the calls, Flashadow can check whether the application is trying to return to a legitimate point in the code or if an attacker has tried to hijack the control flow. Similarly to the call instrumentation, each return instruction is replaced with a jump to our TA. Once Flashadow has gained control, it can fetch the last return site from the secure storage (saved in the call hook) and check whether it matches the one on the application stack. If the two match, then the application can perform the return statement. Otherwise, the application is interrupted to prevent the attacker from completing the exploit.

We propose a second design, uIPS, for our MPU-enabled bare-metal TEE [188]. Contrarily to Flashadow,

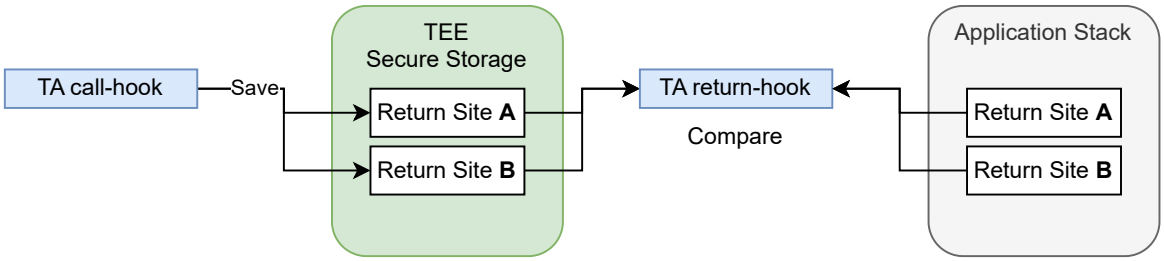


Figure 36: The operations performed in the two Flashadow hooks.

uIPS protects both forward- and backward-edges of the application control flow. The backward edge protection is achieved similarly to Flashadow: each call and each return instructions are instrumented to populate and check a shadow stack, respectively. Once again, the shadow stack is protected through the secure storage of the underlying bare-metal TEE.

As for the forward-edges protection, uIPS provides an index-based check on each indirect branch. Specifically, every time the application needs to jump to an unknown location in the code (dynamic jump/branch), our uIPS TA is invoked. This will then check if the destination address is legitimate by comparing it with a set of allowed destinations. This set, which stems from a previously obtained control flow graph of the application, is stored in the secure storage as well.

3.4 CROSSCON TEE Toolchain

The availability of a secure toolchain for developing software for TEEs is essential, because any vulnerability or compromise in the toolchain can undermine the trust guarantees supposedly provided by the TEE. In this section we describe the main components that were developed in the context of the CROSSCON project for enhancing TEE toolchain security: a new secure update mechanism, the formal foundations for a secure compilation framework, and a proposal for integrating these tools in a DevSecOps flow.

3.4.1 Secure Update

A secure update mechanism is a critical part of a secure TEE toolchain, since it is involved in the provisioning, updating, and management of firmware with security guarantees. Unfortunately, there is no consensus on a secure standard for firmware updates in the IoT world [189]. Many solutions have been proposed, with various use cases and scenarios in mind. In industry practice, different manufacturers follow different approaches depending on their tools, infrastructures, and strategies [190].

For the CROSSCON project, we developed a new secure update scheme [191, 192] that leverages state-of-the-art security and can be deployed even on very constrained IoT devices. The new framework is based on a standard developed by the Software Updates for Internet of Things (SUIT) working group of the Internet Engineering Task Force (IETF), with some specific extension aimed at increasing the transparency of the update process and providing formal guarantees about the functional properties of the update content. The proposed enhancements provide a higher level of assurance for IoT firmware updates, addressing security and safety concerns that are not covered by the original standard.

3.4.1.1 The SUIT standard

The SUIT standard is described in RFC 9019 [193]. It is based on a metadata structure, called the *update manifest*, which is sent to the affected devices when a new update is available. The information contained in the manifest is used for authentication and authorization purposes, to retrieve and validate the

new firmware image, and to fulfill any other requirement that the process might have. The architecture is flexible enough to be applied in many situations (e.g. different device classes, attended or unattended appliances, incremental or monolithic updates), but at the same time can be made robust with regard to functionality and security.

An information model for the SUIT manifest has been described in RFC 9124 [194], and a proposal for a concrete encoding scheme is currently in the draft stage [195]; it is based on the Concise Binary Object Representation (CBOR) binary serialization format (described in RFC 8949) and the associated CBOR Object Signing and Encryption (COSE) packaging mechanism with cryptography support (described in RFC 8152). Moreover, the standard allows for possible extensions to the manifest in order to implement optional capabilities, including firmware encryption, trust domains, update management, inclusion of a file in the MUD format (RFC 8520), and secure methods for an IoT device to report on the firmware update status.

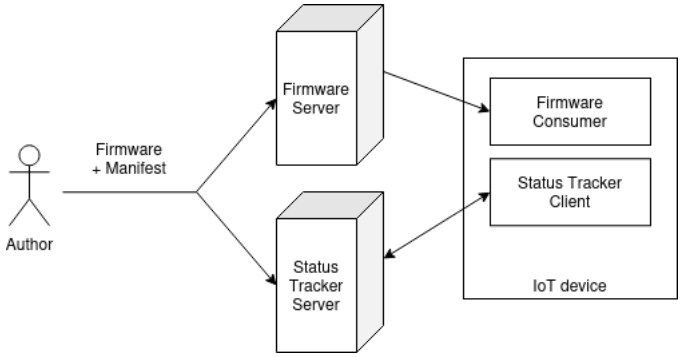


Figure 37: The SUIT secure update architecture.

The architecture of the SUIT update process is schematically depicted in Figure 37 and involves five main components.

- The **Firmware Author** is the agent responsible for creating a new firmware, uploading it to the distribution server and notifying the device management platform.
- The **Firmware Consumer** is the IoT device or, more specifically, one or more software components inside the device that need to be updated;
- The **Firmware Server** is a remote server that can store manifests and firmware images and make them available upon request;
- The **Status Tracker Server** is a remote server that keeps track of software and hardware information about each device on the network and the availability of updates;
- The **Status Tracker Client** is a software component running on the IoT device and communicating with the Status Tracker Server.

The update process can be triggered by the Status Tracker Server (push mode) as soon as a new firmware image is available, or by the Status Tracker Client (pull mode) by periodically querying the server. Hybrid approaches are also possible, and the Status Tracker Client can implement a more complex logic to decide on a time that does not disrupt the device workflow.

Once the update is initiated, the Firmware Consumer receives the manifest and must validate the signature to assert its authenticity. Once the signature is verified, the Firmware Consumer must parse the manifest to check the validity of the update, identify if it applies to the device, and perform the required integrity checks. The manifest can also specify how to perform the update, where to store the firmware, and so on.

Finally, the firmware image is fetched depending on the capabilities of the device; it can be downloaded

using an URI in the manifest, it can be embedded in the manifest itself, or it can be delivered through physical means. The obtained image is then verified and installed according to the instructions contained in the manifest.

3.4.1.2 Novelties of the CROSSCON Secure Update framework

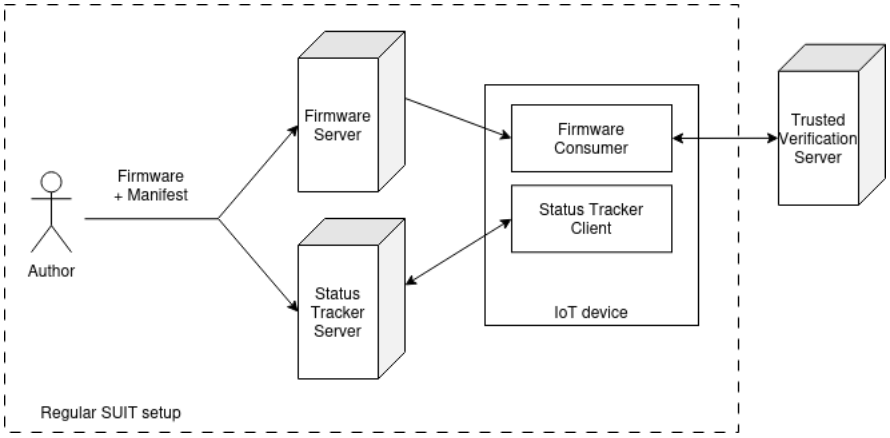


Figure 38: The SUIT secure update architecture, with CROSSCON additions.

The CROSSCON secure update is based on the SUIT standard, with two key enhancements to strengthen the overall process. First, it integrates a *Software Bill Of Materials (SBOM)* to improve the transparency of the process and simplify the tracking of dependencies and vulnerabilities. Second, it extends the SUIT manifest format by adding new fields designed to include formal proofs of selected firmware security properties, serving as a *certification manifest*. Moreover, we supplement the high-level architecture of the SUIT mechanism described in Figure 37 with an additional (trusted) server whose job is to perform the verification of computationally intensive proof steps when needed. The overall scheme is depicted in Figure 38.

We now discuss in more detail the two required additions to the SUIT manifest. The *SBOM* is a structured list of all the software components involved in the construction of the update package (including open-source and third-party components). For each item in the list, the SBOM contains in particular its license, the precise version used in the codebase, and its patch status. When the update is received, this structured list is parsed and for each listed component a check for the existence of possible reported vulnerability related to it is performed. If a vulnerability is found, the update process is aborted and the appropriate authority is notified. In this way, vulnerable software components are identified even before attempting to install them.

The *certification manifest* is a list of formal properties satisfied by the contents of the update package, together with a corresponding proof certificate. The information in the certification manifest enables the device to directly verify the validity of such properties before performing the update. In more detail, each item of the certification manifest is a record consisting of the following fields:

- a unique identifier for the property considered;
- a list of the components targeted by the proof;
- a proof certificate, expressed in a formal language specified by a suitable language identifier;
- a Boolean flag (``locality constraint'') forcing the proof verification step to be performed on the device;
- a list of suggested verification servers, which is only considered when the locality constraint flag is not set.

We now describe in detail the proposed workflow for update generation and delivery.

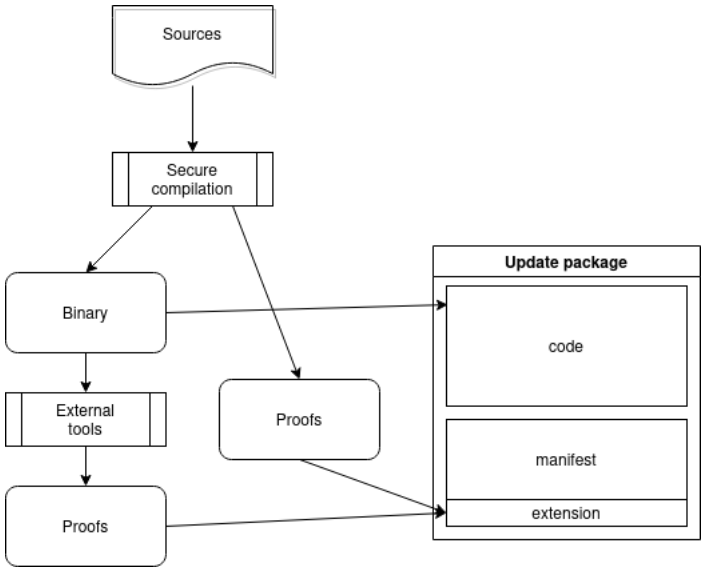


Figure 39: Workflow for update generation.

Update Generation. The workflow for the construction of the update package is summarized in Figure 39. The agent responsible for producing the firmware update (i.e. the *Firmware Author*, in the terminology of subsection 3.4.1.1) starts by compiling the source code of the update, possibly using a certifying compiler, that is a compiler that is able to generate, in addition to the object code, also a proof certificate attesting that the compiled code satisfies some chosen correctness and/or security property [196]. The results of this step are the compiled binary and a set of formal proof certificates for the security properties guaranteed by the compilation process.

In the next stage, the Firmware Author has a chance to run additional static analysis tools on the binary image of the update, thereby obtaining further security proofs to be bundled in the update package. Once this step is completed, the generated proof artifacts are collected in the certification manifest, together with the metadata needed to independently verify them.

Finally, the Firmware Author prepares the update package by bundling together the update image (in case it should be distributed together with the manifest), the regular SUIT manifest, the SBOM and the certification manifest, and signing the result. The resulting package is then sent to the Firmware Server, in order to enable distribution of the update, and the Status Tracker Server is notified of the availability of the update.

Update Installation. The corresponding workflow for the installation of an update package is depicted in Figure 40. At update reception time, the Firmware Consumer starts processing the SUIT envelope by verifying its cryptographic signature. If this basic integrity check passes, the Firmware Consumer can proceed further and extract the SBOM, the manifest itself, and the integrated firmware image (if present). Then the SBOM is checked for the presence of vulnerable components by querying an external (trusted) database of known vulnerabilities. If some vulnerable or otherwise undesirable component (e.g., for licensing reasons) is found, the update is immediately rejected.

If no vulnerable components are found, the Firmware Consumer extracts the proof certificates packaged in the certification manifest and verifies them by feeding each certificate to the appropriate proof checker (as dictated by the corresponding language identifier). This step can either be performed directly on the device (if the available resources allow it) or by a remote procedure call to a trusted external verification server.

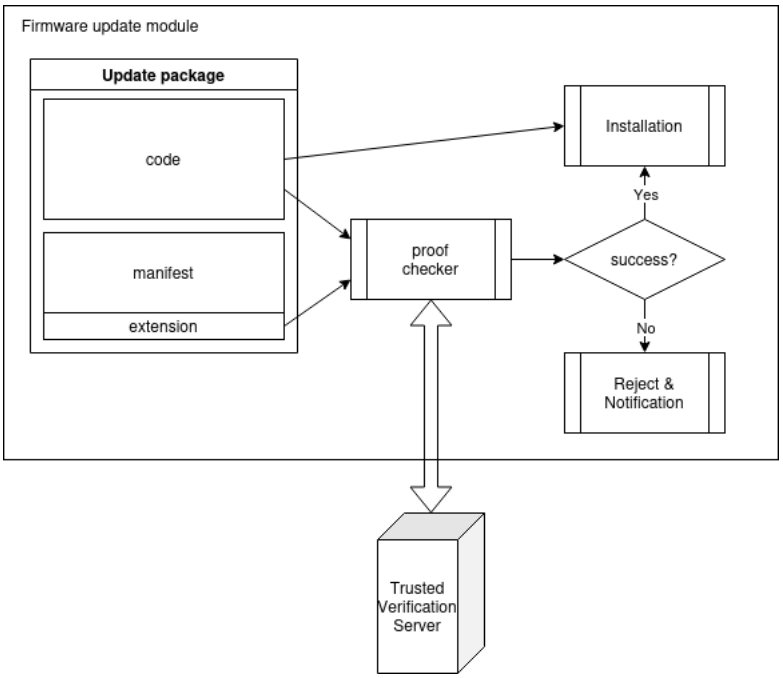


Figure 40: Workflow for update installation.

If any of the proof certificates fails to verify, the update is again rejected and an appropriate notification is sent. If all the proofs are successfully verified, the Firmware Consumer can proceed with the regular SUIT installation workflow, either using the binary image embedded in the update envelope or fetching it from the URI specified in the manifest.

3.4.1.3 Implementation

We implemented a prototype of our framework by extending an existing implementation of the SUIT standard with new code to produce and verify the extended update package. The resulting implementation is available in the CROSSCON Github repositories[9].

For the update production phase, after evaluating the available implementations for the creation of SUIT Manifests and Envelopes, we chose SUIT-Tool[10] by ARM as the target for extension. This Python tool facilitates the generation and signing of manifests starting from a JSON input file structured similarly to the resulting manifest.

Despite the limited documentation available, extending the tool to support the additional fields has been relatively straightforward given the modular structure of the implementation. In particular, the new manifest generation step required changes in two main components:

- the manifest representation, included in the file *manifest.py*, which defines the internal representation of the various components and data types used to represent the manifest. The main modifications in this component are the addition of a field for the SBOM, among the other severable elements of the manifest, and the addition of the Certification Manifest, specified by a set of hierarchical classes and subclasses which implement the structure presented in 3.4.1.2;
- the manifest builder, included in the file *compile.py*, which parses the given input file and encodes its content in the SUIT format. The extension here is as simple as converting both the SBOM and

[9] See https://github.com/crosscon/secure_update_consumer, containing the code for the Firmware Consumer, and https://github.com/crosscon/secure_update_infrastructure, containing the code for the remaining infrastructure, e.g. the trusted verification server.

[10] https://gitlab.arm.com/research/ietf-suit/suit-tool

Certification Manifest objects into the SUIT encoding and inserting them in the output file if they are present in the input in the first place.

For proof generation we leveraged the capabilities of the cvc5 Satisfiability Modulo Theory (SMT) solver [197], which (in its latest versions) is able to produce proof certificates in the Cooperating Proof Calculus (CPC) formal language.

For the update installation phase we extended ARM's reference implementation Suit-Parser[11], written in C, to support the new fields added to the SUIT manifest. The original tool supported only a limited form of parsing (in particular limited to manifests containing a single updated component) and signature verification, so we extended it to handle multiple updated components. In addition to manifest parsing and verification, our implementation supports the extraction and the verification of the SBOM and the checking of the proof certificates contained in the certification manifest. As of this writing, CPC is the only formal language supported for certificate verification; this step is performed by the ETHOS proof checker[12], developed by the same authors of cvc5. However, adding the support for other proof checkers and/or formal languages is straightforward.

3.4.1.4 Proof production

In the course of the project we developed automated proof production scripts for the following classes of safety/security properties.

- **Correctness of static configuration parameters.** In many IoT settings the reliability of a trusted component may depend on the correctness of a set of statically determined *configuration parameters* describing, for instance, memory maps, partitioning of resources, and so on. This correctness requirement can be checked at compile time by defining a formal specification for the allowed values of the configuration parameters as an SMT formula, whose validity is then verified by an SMT solver as part of the compilation pipeline. The resulting proof certificate can be used as a guarantee of the correctness of the configuration deployed by the update.

  This methodology has been applied to the CROSSCON Hypervisor by formalizing the assumptions detailed in the deliverable D2.2 (including for instance the disjointness requirement on the memory regions managed by the separation kernel) and developing an automatic checker for the resulting formal specification. This checker can read a configuration file for the CROSSCON hypervisor, check whether the described configuration is correct (i.e. satisfies the specification), and in the affirmative case produce a proof certificate that witnesses the truth of this requirement.

- **Control flow preservation.** A security policy for software updates may require each new version of a component to retain the same control flow of the original component, without introducing new execution paths. This requirement can be formalized by codifying the Control Flow Graph (CFG) of the original and the updated components as appropriate data structures Γ and Γ', and verify with an SMT solver that the two graphs denoted by Γ and Γ' are isomorphic. The corresponding proof certificate, then, may be used as a formal witness for the fact that the update satisfies the security policy that was set for the software running on the device.

  In the context of the CROSSCON project we developed a Python script, `check_cfp.py`, which is able to extract the control flow graphs of two binary images (interpreted as the original and the updated versions of a software component), establish if the required property holds, and in the affirmative case generate a proof certificate to be bundled with the update, and verified upon deployment.

- **Binary instrumentation.** Many approaches to software hardening use *compile-time instrumentation* techniques to automatically generate binary images that enforce security properties such as memory safety or control flow integrity. It is thus of interest to be able to independently verify the presence

[11] https://gitlab.arm.com/research/ietf-suit/suit-parser
[12] https://github.com/cvc5/ethos

of such instrumentation on a given binary, whether the corresponding source is available (to check the output of the instrumented compiler) or not. In order to verify such a property using an SMT solver, we can encode it into a satisfiability problem in the theory of strings and regular languages, a recent addition to the SMT-LIB standard, as detailed in [191].

In the context of the CROSSCON project we developed a Python script, `check_instr.py`, to analyze the disassembled code of a binary for the presence of a compile-time instrumentation described by the user by means of suitable regular expressions, and automatically generate a proof certificate when the disassembled listing matches the specified instrumentation.

3.4.2 Secure Cross-Compilation for TAs

The main goal of the CROSSCON secure compilation effort is to guarantee memory safety for Trusted Applications (TAs). Memory safety is usually defined as the absence of a certain kind of errors, i.e. *invalid memory accesses*. These errors can be further divided in two classes:

- *spatial* memory errors, where an unauthorised memory location is accessed; typical examples are out-of-bound array accesses, buffer overflows, and null pointer dereferences;
- *temporal* memory errors, where a valid memory area is referenced at an erroneous point in time, for instance when it is not yet, or no longer, valid; typical examples are uninitialized memory reads, use after free, and double free.

The absence of spatial memory errors is called *spatial memory safety*, and analogously the absence of temporal memory errors is called *temporal memory safety*. To ensure complete memory safety, both spatial and temporal errors must be prevented without false negatives.

Memory-safe languages enforce both spatial and temporal memory safety by forbidding direct pointer manipulations, checking object bounds at every array access, and using automatic garbage collection to reclaim heap-allocated objects when it is safe to do so. Unfortunately, TAs are typically written in memory-unsafe languages (e.g. C) and are thus prone to all the pitfalls of manual memory management.

Moreover, TEEs typically do not provide isolation guarantees inside the address space of single TA, but only between different TAs. This means that memory corruption between different TAs are prevented, but invalid memory accesses inside the address space of a single TA are not avoided, which potentially allows an attacker to exploit it.

In order to mitigate this risk, we developed the theoretical foundations for a secure (cross-)compiler that is able to translate a source program expressed in a restricted version of the C programming language to a target program expressed in a generic assembly-level language. The latter is not tied to a single ISA; instead, it is sufficiently generic to be readily adapted to any major existing architecture. The compiler comes with a set of associated security proofs that provide a formal guarantee about the preservation of memory safety properties enjoyed by the source program being compiled.

In the next subsection we describe in broad strokes the overall design of the compiler, whereas in the remaining subsections we briefly describe the main points of the formal models that we used and the main results that we were able to prove. We leave out most of the details, referring the interested reader to the technical report [198] for the full theoretical development.

3.4.2.1 High-level design

We formalize memory safety as a *trace property*, i.e. a property defined on infinite sequences of abstract *memory events*. To this end we define a Finite State Automaton (FSA) $\mathcal{A}$, the *memory safety monitor*, whose input language is a set of abstract memory events. A sequence of memory events is defined to be safe if and only if it is accepted by the automaton $\mathcal{A}$. This way of defining memory safety is

largely independent of the actual language that is being analyzed. This is a crucial requirement in the present setting, as it allows us to establish the *preservation* of this property through translation between different languages.

We then define a formal model of a simple C-like programming language. In fact we will consider two different models, with the same syntax but different semantics. The first model, denoted $\mathsf{L_c}$, allows any kind of operation on pointers and has unchecked memory accesses; this is meant to model the standard, unsafe semantics of the C programming language. This model serves as the source language for the secure compiler. The second model, that we denote $\mathbf{L_{cc}}$, describes a language with the same syntax of the previous one but a different, memory-safe semantics based on an abstract memory coloring scheme. In this language, programs that perform memory unsafe operations (like out of bound accesses or violations of temporal safety) become stuck, i.e. invalid. This model will serve as an intermediate step for the secure compilation process.

Finally, we define a formal model of an assembly-like language, denoted L_{ca}, which abstracts away the architecture-dependent parts of a typical RISC instruction set. This language serves as the target of the secure compiler. The compilation process itself, then, splits naturally into two stages: in the first stage, denoted $C^{\mathsf{c}}_{\mathbf{cc}}(\cdot)$, we reinterpret a $\mathsf{L_c}$ program as the corresponding $\mathbf{L_{cc}}$ program, ensuring that every illegal memory access is caught by the new semantics. In the second stage, the $\mathbf{L_{cc}}$ program is compiled to a L_{ca} program that mirrors more closely the actual code executed by the machine, while preserving the memory safety semantics introduced in the previous step.

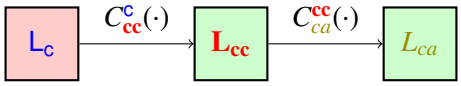


Figure 41: Secure cross-compilation overview.

The overall structure of the process is summarized in Figure 41. Each box corresponds to a formal model of a programming language. A red background corresponds to a memory-unsafe semantics, whereas a green background corresponds to a memory-safe one.

The preservation of the memory safety property from a source $\mathsf{L_c}$ program to the corresponding target L_{ca} program is then proved by formal means, thereby providing a solid guarantee for the security of the (cross-)compilation scheme.

3.4.2.2 The memory safety monitor

The memory safety monitor is based on the notion of *memory coloring* (also known as memory tagging). In a memory coloring scheme, each memory location is paired with an identifier, called a *color* (or tag). Every time a memory region is allocated a new color for it is chosen, and a pointer to that region equipped with the same color is returned. At every subsequent memory access, the underlying hardware ensures that the color contained in the pointer matches the color of the corresponding location. In principle, this ensures that each pointer can only access locations within the memory region from which it was originally derived. However, this scheme only achieves perfect security when no pair of defined memory regions ever share their color, which in general requires an unbounded number of distinct colors.

More abstractly, a general model of memory coloring that hides away as much implementation details as possible (e.g. number of tags available, minimum size of colorable memory regions) can be used to define a language-independent notion of memory safety, along the lines exposed in the previous subsection. Namely, we define a FSA whose goal is to recognize all the safe traces of memory events. This turns memory safety into a trace property, which is thus amenable to verification using standard methods in programming language semantics such as (strong or weak) simulations.

We now briefly describe the definition of the memory safety monitor (MSM). We think of a physical memory as a set of *locations* whose identifiers are drawn from a linearly ordered set $\mathbb{L}$. Since we want a model which is able to cope with any possible memory size, we shall assume that the locations are indexed by natural numbers, i.e. we take $\mathbb{L} = \mathbb{N}$. Each location can be *colored* with an element of a set $\mathbb{C}$. Again, since we want a general model we will take $\mathbb{C}$ to be infinite, and identify it with $\mathbb{N}$. Memory allocation and deallocation events are labeled by a *size* $n \in \mathbb{N}$, $n > 0$. Every allocation event results in a *starting location* $\ell_0 \in \mathbb{L}$, which uniquely identifies the allocated region, and a *color* $c \in \mathbb{C}$ assigned to every location in the half-open interval $[\ell_0, \ell_0 + n)$. A deallocation event is valid if and only if it refers to a previous allocation event which has not been deallocated since. Memory read and write events are valid if and only if they refer to an allocated region and their color coincides with the color of the location they refer to.

To formalize this intuitive picture we introduce two (partial) functions: a *memory tagging function* $T : \mathbb{L} \rightharpoonup_{\mathrm{f}} \mathbb{C}$ mapping each allocated location to its color, and a *region function* $S : \mathbb{L} \rightharpoonup_{\mathrm{f}} \mathbb{N} \times \mathbb{C}$ mapping the starting location of every active region to its size and assigned color. A state of the MSM is a triple composed by a memory tagging function, a region function and a color $\hat{c} \in \mathbb{C}$.

The set of **abstract memory actions** is the inductive set $MemAct$ whose generic element is defined by the following grammar:

$$\alpha := \epsilon \mid read\ \ell^c \mid write\ \ell^c \mid alloc\ n\ \ell^c \mid free\ n\ \ell^c$$

where $n \in \mathbb{N}$, $\ell \in \mathbb{L}$, $c \in \mathbb{C}$. An **abstract memory trace** $\overline{\alpha}$ is any finite sequence of abstract memory actions.

Each transition is labeled with an abstract memory action α. The possible transitions are defined by the following set of inference rules:

$$\text{MS-Read}\ \frac{T(\ell) = c}{(T, S, \hat{c}) \xrightarrow{read\ \ell^c} (T, S, \hat{c})} \qquad \text{MS-Write}\ \frac{T(\ell) = c}{(T, S, \hat{c}) \xrightarrow{write\ \ell^c} (T, S, \hat{c})}$$

$$\text{MS-Alloc}\ \frac{n > 0 \qquad c = \hat{c} \qquad T(\ell + i) = \bot\ \forall i \in [0, n)}{(T, S, \hat{c}) \xrightarrow{alloc\ n\ \ell^c} (T[\ell + i \mapsto c]_{0 \le i < n}, S[\ell \mapsto (n, c)], \hat{c} + 1)}$$

$$\text{MS-Free}\ \frac{S(\ell) = (n, c) \qquad T(\ell + i) = c\ \forall i \in [0, n)}{(T, S, \hat{c}) \xrightarrow{free\ n\ \ell^c} (T[\ell + i \mapsto \bot]_{0 \le i < n}, S[\ell \mapsto \bot], \hat{c})}$$

The *initial state* of the automaton is the state $M_0 := (\emptyset, \emptyset, 0)$. A state M is called *safe* if it is reachable from M_0. An abstract memory trace $\overline{\alpha}$ is safe if and only if there exists a safe execution of the automaton that produces the trace $\overline{\alpha}$.

Using the above formalization we can prove the following key results:

- *Uniqueness of abstract colors*: if $M = (T, S, \hat{c})$ is a safe state then for every $\ell_0, \ell_1 \in \mathbb{L}$ such that $S(\ell_0) = (n_0, c_0)$ and $S(\ell_1) = (n_1, c_1)$ we have that $c_0 = c_1$ implies $\ell_0 = \ell_1$.
- *Region membership implies coloring*: if $M = (T, S, \hat{c})$ is a safe state then for every $\ell_0 \in \mathbb{L}$, if $S(\ell_0) = (n_0, c_0)$ then $T(\ell_0 + k) = c_0$ for every $k \in [0, n_0)$.
- *Region identification*: if $M = (T, S, \hat{c})$ is a safe state then for every $\ell_0 \in \mathbb{L}$, if $T(\ell_0) = c_0$ then there exist and are unique $n_0, i \in \mathbb{N}$ such that $i < n_0$, $T(\ell_0 - i) = c_0$ and $S(\ell_0 - i) = (n_0, c_0)$.

3.4.2.3 The C-like language

The source language is inspired by the formalizations of the C language used by CheckedC [199] and MSWasm [200]. Since we do not consider *intra-object* memory safety in this work, we do not explicitly model structs; instead, we model every C compound type as a single contiguous region of memory (``array'').

$$
\begin{array}{rl}
\text{Simple type:} & \mathsf{w} := \mathsf{int} \mid \mathsf{ptr}\ \tau \\
\text{Value type:} & \tau := \mathsf{w} \mid \mathsf{array}\ \mathsf{w} \\
\text{Function type:} & \sigma := \mathsf{w} \to \mathsf{w} \\
\text{Variable declaration:} & \mathsf{V} := \mathsf{w}\ \mathsf{x} \\
\text{Function declaration:} & \mathsf{D} := \mathsf{w}\ \mathsf{f}(\mathsf{V}) \\
\text{Function definition:} & \mathsf{F} := \mathsf{D}\ \{\ (\mathsf{V}^*)\ \mathsf{e}\ \} \\
\text{Module:} & \mathsf{M} := \mathsf{F}^* \\
\text{Expression:} & \mathsf{e} := \mathsf{z} \mid \mathsf{x} \mid \mathsf{e}; \mathsf{e} \mid \mathsf{e} \odot \mathsf{e} \mid \mathsf{x} := \mathsf{e} \mid \mathsf{x} := \mathsf{f}(\mathsf{e}) \mid *\mathsf{e} \mid *\mathsf{e} := \mathsf{e} \\
 & \quad \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}\ \mathsf{else}\ \mathsf{e} \mid \mathsf{malloc}\ \mathsf{e} \mid \mathsf{free}\ \mathsf{e} \\
\text{Binary operator:} & \odot := + \mid - \mid * \mid == \mid < \mid \&\& \mid \| \\
\text{Runtime value:} & \mathsf{v} := \mathsf{z} \mid \mathsf{n}^{\mathsf{s}} \\
\text{Allocation slot:} & \mathsf{s} := \mathsf{n}\ \mathsf{n}\ \mathsf{n}
\end{array}
$$

Figure 42: Syntax of $\mathsf{L_c}$.

The grammar defining the syntax of $\mathsf{L_c}$ is presented in fig. 42. The terminal symbols are *positive integers* (n), *integers* (z), *variable identifiers* (x) and *function identifiers* (f). Among C types we only model integers, pointers and arrays. Moreover, we consider only arrays of simple types; multi-dimensional arrays must be recast in terms of arrays of pointers. For simplicity, our integer type is unbounded, which lets us sidestep all issues related to integer overflow. We model only unary functions; functions of multiple arguments can be coded as functions taking (a pointer to) an array of arguments. We allow only simple types in function signatures and local variables; this implies that every array must be allocated dynamically, and can be accessed only through pointers. The only static values in the language are integer values; explicit memory addresses are treated like integers. Pointer values, which consist of a positive integer address together with an associated allocation slot, are runtime values and can be obtained only through calls to malloc. An *allocation slot* is a triple of positive integers $\mathsf{s} = (\mathsf{b}, \mathsf{n}, \mathsf{i})$ where b is the *base* (or starting address) of the slot, n is the *size*, and i is an (unique) *identifier*. A pointer value a^{s} is a pair consisting of an address a and an allocation slot s.

For simplicity we do not introduce a separate statement class; loops and other control flow statements must be recast in terms of conditionals and function calls. The body of a function is taken to be a single expression whose evaluation gives the return value of the function. The usual array access syntax $\mathsf{e}_1[\mathsf{e}_2]$ can be desugared to $*(\mathsf{e}_1 + \mathsf{e}_2)$.

Unsafe semantics. We define a small-step labeled operational semantics for $\mathsf{L_c}$ whose transition relation is denoted $\Theta \xrightarrow{\alpha} \Theta'$. The $\mathsf{L_c}$ memory actions are defined by the following grammar:

$$\alpha := \epsilon \mid \mathsf{pread}(\mathsf{a}^{\mathsf{s}}) \mid \mathsf{pwrite}(\mathsf{a}^{\mathsf{s}}) \mid \mathsf{iread}(\mathsf{z}) \mid \mathsf{iwrite}(\mathsf{z}) \mid \mathsf{palloc}(\mathsf{a}^{\mathsf{s}}) \mid \mathsf{pfree}(\mathsf{a}^{\mathsf{s}}) \mid \mathsf{ifree}(\mathsf{z})$$

We consider read, write and free operations accepting either a pointer value (which may be valid or invalid) or an integer (i.e., a bare memory address).

A *local configuration* Θ is a quadruple $(\theta, \mathsf{H}, \mathsf{A}, \mathsf{e})$ where:

- θ (*local environment*) is a finite partial map $\theta\colon \mathsf{Ident} \rightharpoonup_{\mathrm{f}} \mathsf{Val}$;
- H (*heap*) is a total function $\mathsf{H}\colon [0, h_s) \to \mathsf{Val}$, where $h_s \in \mathbb{N}$ is a fixed heap size;
- A (*allocator state*) is a triple of the form $(\overline{\mathsf{s}}_{\mathsf{A}}, \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max})$, where $\overline{\mathsf{s}}_{\mathsf{A}}$ is the list of current allocated slots, $\overline{\mathsf{s}}_{\mathsf{F}}$ is the list of current free slots, and $\mathsf{i}_{\max}$ is the first unused slot identifier;
- e is the expression to be evaluated.

The rules for the evaluation of assignments, conditionals, etc. are standard. The evaluation of expressions which involve a memory operation proceeds according to the following rules:

$$\mathsf{L_c}\text{-E-PRead}\ \frac{\mathsf{H}(\mathsf{a}) = \mathsf{v}}{\vdash (\theta, \mathsf{H}, \mathsf{A}, *\mathsf{a}^{\mathsf{s}}) \xrightarrow{\mathsf{pread}(\mathsf{a}^{\mathsf{s}})} (\theta, \mathsf{H}, \mathsf{A}, \mathsf{v})} \qquad \mathsf{L_c}\text{-E-PWrite}\ \frac{}{\vdash (\theta, \mathsf{H}, \mathsf{A}, *\mathsf{a}^{\mathsf{s}} := \mathsf{v}) \xrightarrow{\mathsf{pwrite}(\mathsf{a}^{\mathsf{s}})} (\theta, \mathsf{H}[\mathsf{a} \mapsto \mathsf{v}], \mathsf{A}, \mathsf{v})}$$

$$\mathsf{L_c}\text{-E-IRead}\ \frac{\mathsf{H}(\mathsf{z}) = \mathsf{v}}{\vdash (\theta, \mathsf{H}, \mathsf{A}, *\mathsf{z}) \xrightarrow{\mathsf{iread}(\mathsf{z})} (\theta, \mathsf{H}, \mathsf{A}, \mathsf{v})} \qquad \mathsf{L_c}\text{-E-IWrite}\ \frac{}{\vdash (\theta, \mathsf{H}, \mathsf{A}, *\mathsf{z} := \mathsf{v}) \xrightarrow{\mathsf{iwrite}(\mathsf{z})} (\theta, \mathsf{H}[\mathsf{z} \mapsto \mathsf{v}], \mathsf{A}, \mathsf{v})}$$

$$\mathsf{L_c}\text{-E-Malloc}\ \frac{\mathsf{s} = (\mathsf{a}, \mathsf{n}, \mathsf{i}) \qquad \vdash \mathsf{A} \xhookrightarrow{\mathsf{palloc}(\mathsf{a}^{\mathsf{s}})} \mathsf{A}'}{\vdash (\theta, \mathsf{H}, \mathsf{A}, \mathsf{malloc\ n}) \xrightarrow{\mathsf{palloc}(\mathsf{a}^{\mathsf{s}})} (\theta, \mathsf{H}, \mathsf{A}', \mathsf{a}^{\mathsf{s}})}$$

$$\mathsf{L_c}\text{-E-PFree}\ \frac{\mathsf{s} = (\mathsf{a}, \mathsf{n}, \mathsf{i}) \qquad \vdash \mathsf{A} \xhookrightarrow{\mathsf{pfree}(\mathsf{a}^{\mathsf{s}})} \mathsf{A}'}{\vdash (\theta, \mathsf{H}, \mathsf{A}, \mathsf{free\ a}^{\mathsf{s}}) \xrightarrow{\mathsf{pfree}(\mathsf{a}^{\mathsf{s}})} (\theta, \mathsf{H}, \mathsf{A}', 0)} \qquad \mathsf{L_c}\text{-E-IFree}\ \frac{\vdash \mathsf{A} \xhookrightarrow{\mathsf{ifree}(\mathsf{z})} \mathsf{A}'}{\vdash (\theta, \mathsf{H}, \mathsf{A}, \mathsf{free\ z}) \xrightarrow{\mathsf{ifree}(\mathsf{z})} (\theta, \mathsf{H}, \mathsf{A}', 0)}$$

where $\mathsf{A} \xhookrightarrow{\alpha} \mathsf{A}'$ is a transition relation between allocator states labeled by $\mathsf{L_c}$ (de)allocation actions, defined by the following set of rules:

$$\mathsf{L_c}\text{-Alloc}\ \frac{\mathsf{A} = (\overline{\mathsf{s}}_{\mathsf{A}}, \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max}) \qquad \mathsf{s} = (\mathsf{a}, \mathsf{n}, \mathsf{i}) \qquad \mathsf{i} = \mathsf{i}_{\max} \\ \overline{\mathsf{s}}_{\mathsf{F}} = \overline{\mathsf{s}}_1 +\!\!+ (\mathsf{a}, \mathsf{n}', \mathsf{i}') :: \overline{\mathsf{s}}_2 \qquad 0 < \mathsf{n} \le \mathsf{n}' \qquad \nexists \mathsf{s}' \in \overline{\mathsf{s}}_1.\mathsf{s}'_{.2} \ge \mathsf{n}}{\vdash \mathsf{A} \xhookrightarrow{\mathsf{palloc}(\mathsf{a}^{\mathsf{s}})} (\mathsf{s} :: \overline{\mathsf{s}}_{\mathsf{A}}, \overline{\mathsf{s}}_1 +\!\!+ (\mathsf{a} + \mathsf{n}, \mathsf{n}' - \mathsf{n}, \mathsf{i}') :: \overline{\mathsf{s}}_2, \mathsf{i}_{\max} + 1)}$$

$$\mathsf{L_c}\text{-Free}\ \frac{\mathsf{A} = (\overline{\mathsf{s}}_{\mathsf{A}}, \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max}) \qquad \mathsf{s} = (\mathsf{a}, \mathsf{n}, \mathsf{i}) \qquad \overline{\mathsf{s}}_{\mathsf{A}} = \overline{\mathsf{s}}_1 +\!\!+ \mathsf{s} :: \overline{\mathsf{s}}_2}{\vdash \mathsf{A} \xhookrightarrow{\mathsf{pfree}(\mathsf{a}^{\mathsf{s}})} (\overline{\mathsf{s}}_1 +\!\!+ \overline{\mathsf{s}}_2, \mathsf{s} :: \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max})}$$

$$\mathsf{L_c}\text{-FreeInv}\ \frac{\mathsf{A} = (\overline{\mathsf{s}}_{\mathsf{A}}, \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max}) \qquad \mathsf{s} = (\mathsf{a}, \mathsf{n}, \mathsf{i}) \qquad \mathsf{s} \notin \overline{\mathsf{s}}_{\mathsf{A}}}{\vdash \mathsf{A} \xhookrightarrow{\mathsf{pfree}(\mathsf{a}^{\mathsf{s}})} \mathsf{A}}$$

$$\mathsf{L_c}\text{-FreeInt}\ \frac{\mathsf{A} = (\overline{\mathsf{s}}_{\mathsf{A}}, \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max}) \qquad \overline{\mathsf{s}}_{\mathsf{A}} = \overline{\mathsf{s}}_1 +\!\!+ (\mathsf{a}, \mathsf{n}, \mathsf{i}) :: \overline{\mathsf{s}}_2}{\vdash \mathsf{A} \xhookrightarrow{\mathsf{ifree}(\mathsf{a})} (\overline{\mathsf{s}}_1 +\!\!+ \overline{\mathsf{s}}_2, (\mathsf{a}, \mathsf{n}, \mathsf{i}) :: \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max})}$$

$$\mathsf{L_c}\text{-FreeIntInv}\ \frac{\mathsf{A} = (\overline{\mathsf{s}}_{\mathsf{A}}, \overline{\mathsf{s}}_{\mathsf{F}}, \mathsf{i}_{\max}) \qquad (\mathsf{a}, *, *) \notin \overline{\mathsf{s}}_{\mathsf{A}}}{\vdash \mathsf{A} \xhookrightarrow{\mathsf{ifree}(\mathsf{a})} \mathsf{A}}$$

We define a $\mathsf{L_c}$ program as *memory safe* when all its executions can be simulated by a (safe) execution of the memory safety monitor. In order to do this we need:

1. a way to match abstract MSM locations with concrete addresses in the $\mathsf{L_c}$ heap;
2. a way to match MSM states with $\mathsf{L_c}$ configurations;
3. a relation between abstract memory actions α and $\mathsf{L_c}$ memory actions α.

To solve the first issue, we define an $\mathsf{L_c}$ *address matching function* to be a finite partial bijection

$$\delta \colon [0, h_s) \times \mathbb{N} \rightharpoonup_{\mathrm{f}} \mathbb{L} \times \mathbb{C}$$

subject to some validity conditions (that will not be detailed here). We interpret the equality $\delta(\mathsf{a}, \mathsf{i}) = (\ell, c)$ as the assertion that the address a in the allocation slot with identifier i corresponds to the abstract location ℓ with abstract color c.

To solve the second issue we introduce a notion of agreement between a MSM state M and a configuration Θ, parameterized by an address matching function δ. Given a $\mathsf{L_c}$ allocator state A, we write $\mathsf{A} \sim_{\delta} M$ if the following condition holds: for every $\mathsf{L_c}$ allocation slot $(\mathsf{b}, \mathsf{n}, \mathsf{i})$,

$$\mathsf{s} \in \overline{\mathsf{s}}_{\mathsf{A}} \text{ if and only if } \delta(\mathsf{b}, \mathsf{i}) = (\ell_0, c) \text{ and } S(\ell_0) = (n, c)\,. \tag{3.1}$$

Finally, to fulfill point 3 we define a binary relation between MSM and $\mathsf{L_c}$ memory actions, also parameterized by a valid address matching function δ, that we will denote by $\alpha \sim_\delta \alpha$. This relation is defined by the following set of rules:

$$\mathsf{L_c}\text{-TR-Read}\ \frac{\mathsf{s} = (\mathsf{b}, \mathsf{n}, \mathsf{i}) \qquad \delta(\mathsf{a}, \mathsf{i}) = (\ell, c)}{\mathsf{pread}(\mathsf{a}^{\mathsf{s}}) \sim_\delta \mathit{read}\ \ell^c} \qquad \mathsf{L_c}\text{-TR-Write}\ \frac{\mathsf{s} = (\mathsf{b}, \mathsf{n}, \mathsf{i}) \qquad \delta(\mathsf{a}, \mathsf{i}) = (\ell, c)}{\mathsf{pwrite}(\mathsf{a}^{\mathsf{s}}, \mathsf{v}) \sim_\delta \mathit{write}\ \ell^c}$$

$$\mathsf{L_c}\text{-TR-Alloc}\ \frac{\mathsf{s} = (\mathsf{a}, \mathsf{n}, \mathsf{i}) \qquad \delta(\mathsf{a}, \mathsf{i}) = (\ell, c) \qquad [\mathsf{n}] = n}{\mathsf{palloc}(\mathsf{a}^{\mathsf{s}}) \sim_\delta \mathit{alloc}\ n\ \ell^c}$$

$$\mathsf{L_c}\text{-TR-Free}\ \frac{\mathsf{s} = (\mathsf{a}, \mathsf{n}, \mathsf{i}) \qquad \delta(\mathsf{a}, \mathsf{i}) = (\ell, c) \qquad [\mathsf{n}] = n}{\mathsf{pfree}(\mathsf{a}^{\mathsf{s}}) \sim_\delta \mathit{free}\ n\ \ell^c}$$

We can now define a notion of memory safety for the evaluation of $\mathsf{L_c}$ functions and programs as follows:

- A local configuration $\Theta = (\theta, \mathsf{H}, \mathsf{A}, \mathsf{e})$ is *memory safe* if there exist a safe MSM state M and a valid address matching function δ such that $\mathsf{A} \sim_\delta M$;
- Given another configuration $\Theta' = (\theta', \mathsf{H}', \mathsf{A}', \mathsf{e}')$, the transition $\Theta \xrightarrow{\alpha} \Theta'$ is *memory safe* if there exist a MSM state M', a valid address matching function δ' extending δ and an abstract memory action α such that $\Theta'.\mathsf{A} \sim_{\delta'} M'$, $\alpha \sim_{\delta'} \alpha$ and $M \xrightarrow{\alpha} M'$.

Safe semantics. We now define a second formal model for the same C-like language by adding (concrete) colors to the allocation slots. On the syntactic level, we add a new terminal symbol $\mathbf{c}$ for *concrete colors* (not to be confused with the abstract colors, i.e. elements of $\mathbb{C}$, used in the memory safety monitor). The set of all possible concrete colors will be denoted by $\mathbf{Col}$. Here we have two possible choices:

- We can require $\mathbf{Col}$ to be equipped with a map $\mathbf{next} \colon \mathcal{P}(\mathbf{Col}) \to \mathbf{Col}$ that sends each subset $\mathbf{C} \subseteq \mathbf{Col}$ to a ``fresh'' color (i.e., $\mathbf{c} \in \mathbf{Col}$ such that $\mathbf{c} \notin \mathbf{C}$); this forces the set $\mathbf{Col}$ to be infinite.
- Alternatively, we can only require the existence of a *partial* map $\mathbf{next} \colon \mathcal{P}(\mathbf{Col}) \rightharpoonup_{\mathrm{f}} \mathbf{Col}$ that sends each subset $\mathbf{C} \subseteq \mathbf{Col}$ to a fresh color *when one is available* (and is undefined otherwise). This allows a set $\mathbf{Col}$ of every finite cardinality.

The model resulting from the first choice (i.e., infinite concrete colors) will be denoted by $\mathbf{L_{cc}}$. We only consider this model in what follows, which corresponds to not committing ourselves to a fixed number of concrete colors. It is however very easy to adapt the formalism to the case of a finite number of concrete colors, at the price of a more conservative semantics (that is, a semantics that rejects some memory-safe programs due to the exhaustion of available concrete colors during execution). We also require the existence of a distinguished color $\mathbf{c_0} \in \mathbf{Col}$, used to mark unallocated memory regions.

The only difference with respect to the syntax shown in fig. 42 is that $\mathbf{L_{cc}}$ allocation slots are now quadruples $\mathbf{s} = (\mathbf{b}, \mathbf{n}, \mathbf{i}, \mathbf{c})$ involving also a concrete color. The semantics is again specified by a transition relation $\mathbf{\Theta} \xrightarrow{\alpha} \mathbf{\Theta}'$ labeled by memory actions. The set of $\mathbf{L_{cc}}$ memory actions is defined as follows:

$$\alpha := \epsilon \mid \mathbf{pread}(\mathbf{a}^{\mathbf{s}}) \mid \mathbf{pwrite}(\mathbf{a}^{\mathbf{s}}) \mid \mathbf{palloc}(\mathbf{a}^{\mathbf{s}}) \mid \mathbf{pfree}(\mathbf{a}^{\mathbf{s}}) \mid \mathbf{trap}$$

No memory actions using bare integers are allowed; in their place we have a $\mathbf{trap}$ action, which signals that an illegal memory operation has been performed.

A local configuration $\mathbf{\Theta}$ is either a quadruple $(\theta, \mathbf{H}, \mathbf{A}, \mathbf{e})$, where

- $\mathbf{H}$ (*colored heap*) is a total function $\mathbf{H} \colon [0, h_s) \to \mathbf{Val} \times \mathbf{Col}$;
- $\mathbf{A}$ (*colored allocator state*) is a quadruple of the form $(\bar{\mathbf{s}}_{\mathbf{A}}, \bar{\mathbf{s}}_{\mathbf{F}}, \mathbf{i}_{\max}, \mathbf{C})$, where $\mathbf{C}$ is the set of concrete colors currently in use;

- θ, $\mathbf{e}$ are as in $\mathsf{L_c}$,

or the special configuration $\mathbf{err}$ (error state).

The new evaluation rules for memory operations are as follows:

$$\mathbf{L_{cc}}\text{-E-PRead}\ \frac{\mathbf{s}_{.4} = \mathbf{c} \qquad \mathbf{H}(\mathbf{a}) = (\mathbf{v}, \mathbf{c}') \qquad \mathbf{c} = \mathbf{c}'}{\vdash (\theta, \mathbf{H}, \mathbf{A}, *\mathbf{a}^{\mathbf{s}}) \xrightarrow{\mathbf{pread}(\mathbf{a}^{\mathbf{s}})} (\theta, \mathbf{H}, \mathbf{A}, \mathbf{v})}$$

$$\mathbf{L_{cc}}\text{-E-PReadInv}\ \frac{\mathbf{s}_{.4} = \mathbf{c} \qquad \mathbf{H}(\mathbf{a}) = (\mathbf{v}, \mathbf{c}') \qquad \mathbf{c} \neq \mathbf{c}'}{\vdash (\theta, \mathbf{H}, \mathbf{A}, *\mathbf{a}^{\mathbf{s}}) \xrightarrow{\mathbf{trap}} \mathbf{err}}$$

$$\mathbf{L_{cc}}\text{-E-PWrite}\ \frac{\mathbf{s}_{.4} = \mathbf{c} \qquad \mathbf{H}(\mathbf{a}) = (\mathbf{v}', \mathbf{c}') \qquad \mathbf{c} = \mathbf{c}'}{\vdash (\theta, \mathbf{H}, \mathbf{A}, *\mathbf{a}^{\mathbf{s}} := \mathbf{v}) \xrightarrow{\mathbf{pwrite}(\mathbf{a}^{\mathbf{s}})} (\theta, \mathbf{H}[\mathbf{a} \mapsto \mathbf{v}, \mathbf{c}], \mathbf{A}, \mathbf{v})}$$

$$\mathbf{L_{cc}}\text{-E-PWriteInv}\ \frac{\mathbf{s}_{.4} = \mathbf{c} \qquad \mathbf{H}(\mathbf{a}) = (\mathbf{v}', \mathbf{c}') \qquad \mathbf{c} \neq \mathbf{c}'}{\vdash (\theta, \mathbf{H}, \mathbf{A}, *\mathbf{a}^{\mathbf{s}} := \mathbf{v}) \xrightarrow{\mathbf{trap}} \mathbf{err}}$$

$$\mathbf{L_{cc}}\text{-E-Malloc}\ \frac{\mathbf{s} = (\mathbf{a}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \vdash \mathbf{A} \xhookrightarrow{\mathbf{palloc}(\mathbf{a}^{\mathbf{s}})} \mathbf{A}'}{\vdash (\theta, \mathbf{H}, \mathbf{A}, \mathbf{malloc\ n}) \xrightarrow{\mathbf{palloc}(\mathbf{a}^{\mathbf{s}})} (\theta, \mathbf{H}, \mathbf{A}', \mathbf{a}^{\mathbf{s}})}$$

$$\mathbf{L_{cc}}\text{-E-PFree}\ \frac{\mathbf{s} = (\mathbf{a}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \vdash \mathbf{A} \xhookrightarrow{\mathbf{pfree}(\mathbf{a}^{\mathbf{s}})} \mathbf{A}'}{\vdash (\theta, \mathbf{H}, \mathbf{A}, \mathbf{free\ a}^{\mathbf{s}}) \xrightarrow{\mathbf{pfree}(\mathbf{a}^{\mathbf{s}})} \left(\theta, \mathbf{H}[\mathbf{a}+\mathbf{j} \mapsto (*, \mathbf{c_0})]_{\mathbf{j} \in [\mathbf{0},\mathbf{n})}, \mathbf{A}', \mathbf{0}\right)}$$

$$\mathbf{L_{cc}}\text{-E-PFreeInv}\ \frac{\mathbf{s} = (\mathbf{a}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \nvdash \mathbf{A} \xhookrightarrow{\mathbf{pfree}(\mathbf{a}^{\mathbf{s}})} \mathbf{A}'}{\vdash (\theta, \mathbf{H}, \mathbf{A}, \mathbf{free\ a}^{\mathbf{s}}) \xrightarrow{\mathbf{trap}} \mathbf{err}} \qquad \mathbf{L_{cc}}\text{-E-IFree}\ \frac{}{\vdash (\theta, \mathbf{H}, \mathbf{A}, \mathbf{free\ z}) \xrightarrow{\mathbf{trap}} \mathbf{err}}$$

The transition rules for the $\mathbf{L_{cc}}$ allocator state are:

$$\mathbf{L_{cc}}\text{-Alloc}\ \frac{\begin{array}{c}\mathbf{A} = (\bar{\mathbf{s}}_{\mathbf{A}}, \bar{\mathbf{s}}_{\mathbf{F}}, \mathbf{i}_{\max}, \mathbf{C}) \qquad \mathbf{s} = (\mathbf{a}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \mathbf{i} = \mathbf{i}_{\max} \qquad \mathbf{c} = \mathbf{next}(\mathbf{C}) \\ \bar{\mathbf{s}}_{\mathbf{F}} = \bar{\mathbf{s}}_1 +\!\!+ (\mathbf{a}, \mathbf{n}', \mathbf{i}', \mathbf{c}') :: \bar{\mathbf{s}}_2 \qquad \mathbf{0} < \mathbf{n} \leq \mathbf{n}' \qquad \nexists \mathbf{s}' \in \bar{\mathbf{s}}_1.\mathbf{s}'_{.2} \geq \mathbf{n}\end{array}}{\vdash \mathbf{A} \xhookrightarrow{\mathbf{palloc}(\mathbf{a}^{\mathbf{s}})} (\mathbf{s} :: \bar{\mathbf{s}}_{\mathbf{A}}, \bar{\mathbf{s}}_1 +\!\!+ (\mathbf{a}+\mathbf{n}, \mathbf{n}'-\mathbf{n}, \tau', \mathbf{i}', \mathbf{c}') :: \bar{\mathbf{s}}_2, \mathbf{i}_{\max}+\mathbf{1}, \mathbf{C} \cup \{\mathbf{c}\})}$$

$$\mathbf{L_{cc}}\text{-Free}\ \frac{\mathbf{A} = (\bar{\mathbf{s}}_{\mathbf{A}}, \bar{\mathbf{s}}_{\mathbf{F}}, \mathbf{i}_{\max}, \mathbf{C}) \qquad \mathbf{s} = (\mathbf{a}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \bar{\mathbf{s}}_{\mathbf{A}} = \bar{\mathbf{s}}_1 +\!\!+ \mathbf{s} :: \bar{\mathbf{s}}_2}{\vdash \mathbf{A} \xhookrightarrow{\mathbf{pfree}(\mathbf{a}^{\mathbf{s}})} (\bar{\mathbf{s}}_1 +\!\!+ \bar{\mathbf{s}}_2, \mathbf{s} :: \bar{\mathbf{s}}_{\mathbf{F}}, \mathbf{i}_{\max}, \mathbf{C})}$$

The definition of memory safety for $\mathbf{L_{cc}}$ programs proceeds along the same guidelines used for $\mathsf{L_c}$:

- we require the existence of a $\mathbf{L_{cc}}$ address matching function, that is a finite partial bijection

$$\delta : [0, h_s) \times \mathbb{N} \times \mathbf{Col} \rightharpoonup_{\mathrm{f}} \mathbb{L} \times \mathbb{C}$$

  (with appropriate validity conditions) that matches the addresses in the $\mathbf{L_{cc}}$ heap to the MSM locations;

- we define an associated agreement relation $\mathbf{A} \sim_\delta M$ between colored allocator states and abstract memory states;

- we define a memory action relation $\alpha \sim_\delta \alpha$ according to the following set of rules:

$$\mathbf{L_{cc}}\text{-TR-Read}\ \frac{\mathbf{s} = (\mathbf{b}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \delta(\mathbf{a}, \mathbf{i}, \mathbf{c}) = (\ell, c)}{\mathbf{pread}(\mathbf{a}^{\mathbf{s}}) \sim_\delta \mathit{read}\ \ell^c}$$

$$\mathbf{L_{cc}}\text{-TR-Write}\ \frac{\mathbf{s} = (\mathbf{b}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \delta(\mathbf{a}, \mathbf{i}, \mathbf{c}) = (\ell, c)}{\mathbf{pwrite}(\mathbf{a}^{\mathbf{s}}, \mathbf{v}) \sim_\delta \mathit{write}\ \ell^c}$$

$$\mathbf{L_{cc}}\text{-TR-Alloc}\ \frac{\mathbf{s} = (\mathbf{a}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \delta(\mathbf{a}, \mathbf{i}, \mathbf{c}) = (\ell, c) \qquad [\mathbf{n}] = n}{\mathbf{palloc}(\mathbf{a}^{\mathbf{s}}) \sim_\delta \mathit{alloc}\ n\ \ell^c}$$

$$
\begin{array}{rl}
\text{Program:} & M := I^* \\
\text{Instruction:} & I := I_a \mid I_b \mid I_j \mid I_t \mid I_m \mid I_c \\
\text{Arithmetic instr.:} & I_a := O_a\ r\ r\ r \\
\text{Arithmetic opcode:} & O_a := add \mid sub \mid mul \\
\text{Branching instr.:} & I_b := O_b\ r\ i \\
\text{Branching opcode:} & O_b := jz \mid jn \\
\text{Jumping instr.:} & I_j := jmp\ i \mid call\ i \mid ret \\
\text{Transfer instr.:} & I_t := movi\ r\ i \mid mov\ r\ r \\
\text{Memory instr.:} & I_m := ld\ r\ r \mid st\ r\ r \mid alloc\ r\ r \mid free\ r \mid size\ r\ r \\
\text{Coloring instr.:} & I_c := newc\ r \mid stc\ r\ c \mid setc\ r\ r \\
\text{Allocation slot:} & s := n\ n\ n
\end{array}
$$

Figure 43: Syntax of L_{ca}.

$$
\mathbf{L_{cc}}\text{-TR-Free}\ \frac{\mathbf{s} = (\mathbf{a}, \mathbf{n}, \mathbf{i}, \mathbf{c}) \qquad \delta(\mathbf{a}, \mathbf{i}, \mathbf{c}) = (\ell, c) \qquad [\mathbf{n}] = n}{\mathbf{pfree}(\mathbf{a^s}) \sim_\delta free\ n\ \ell^c}
$$

We then define memory safety for local configurations and transitions in the same way as for $\mathsf{L_c}$.

3.4.2.4 The assembly-like language

The target language is a simple idealized assembly-like language with infinitely many registers holding (unbounded) integers, a finite memory of size $N > 0$ words (where each word again can hold any integer), and a reserved memory region for code, indexed by natural numbers. We denote the corresponding formal model with L_{ca}.

The grammar defining the syntax of L_{ca} is presented in fig. 43. The terminal symbols are: *immediates* $i \in \mathbb{Z}$, *registers* $r \in Reg$ (we take $Reg = \mathbb{N}$ for simplicity), and *concrete colors* $c \in Col$, with a distinguished color $c_0 \in Col$.

A L_{ca} program is simply a list of instructions; each instruction can be referred to by its index, so there is no need for explicit labels. Branching instructions use pc-relative addressing, whereas jump instructions use absolute indexes which are generated either at compilation time (in the case of the entry point of a function) or dynamically in case of procedure calls, using a separate call stack which is handled by the *call* and *ret* instructions.

The intended semantics of the memory and coloring instructions is as follows:

- $ld\ r_d\ r_a$ loads into register r_d the value at the colored memory address contained in register r_a;
- $st\ r_a\ r_s$ stores at the colored memory address contained in register r_a the value contained in register r_s;
- $alloc\ r_b\ r_n$ allocates a block of size specified by register r_n in memory and stores its starting (uncolored) address into register r_b;
- $free\ r_b$ frees a previously allocated block starting at the colored address specified by register r_b;
- $size\ r_d\ r_b$ returns the size of a previously allocated block starting at the colored address specified by register r_b in register r_d;
- $newc\ r_d$ adds to the address contained in register r_d a new concrete color not previously used;
- $stc\ r_d\ c$ adds to the address contained in register r_d the concrete color c;

- $setc\ r_b\ r_n$ sets the color of every memory cell in the block starting at the address specified by register r_b and with size specified by register r_n to the color contained in register r_b.

Bare memory addresses are represented as (positive) integers. Colored memory addresses are represented by means of a pairing function whose purpose is to code a value-color pair in a single L_{ca} value. To this end, we assume that the following three (total) maps are defined:

$$pair : Addr \times Col \to Val \tag{3.2}$$
$$addr : Val \to Addr \tag{3.3}$$
$$col : Val \to Col \tag{3.4}$$

They must be such that the following equalities hold:

$$addr(pair(a, c)) = a \quad \forall a \in Addr, c \in Col \tag{3.5}$$
$$col(pair(a, c)) = c \quad \forall a \in Addr, c \in Col \tag{3.6}$$
$$pair(addr(v), col(v)) = v \quad \forall v \in Val \tag{3.7}$$

We now define a labeled small-step labeled operational semantics for L_{ca}. The memory actions are defined by the following grammar:

$$\alpha := \epsilon \mid read(x) \mid write(x) \mid alloc(s) \mid free(s)$$

where $x \in Val$, $s \in Slots$.

A L_{ca} machine state Ω is a tuple $\Omega = (pc, R, S, M, A, C)$ where:

- $pc \in \mathbb{N}$ is the program counter, that is the index of the instruction currently being executed;
- $R : \mathbb{N} \to_{\mathrm{f}} Val$ is a partial map sending each pseudoregister to the L_{ca} value stored in it;
- $S \in \mathrm{List}\,\mathbb{N}$ is the call stack;
- $M : [0, N) \to Val \times Col$ is a total function representing a memory of some fixed size N;
- A is a triple of the form $(\bar{s}_A, \bar{s}_F, i_{\max})$, with the usual meaning;
- $C \subseteq Col$ is a finite subset of concrete colors.

The operational semantics for a L_{ca} program $\bar{I}$ is given in terms of a transition relation between machine states, denoted $\bar{I} \vdash \Omega \xrightarrow{\alpha} \Omega'$. Again, most of the rules are standard; the ones involving a memory operation are as follows.

$$L_{ca}\text{-E-Load}\ \frac{\bar{I}(pc) = ld\ r_d\ r_a \quad addr(R(r_a)) \in Addr \quad col(R(r_a)) = M(addr(R(r_a)))_{.2} \quad I(pc+1)\ def}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{read(R(r_a))} (pc+1, R[r_d \mapsto M(addr(R(r_a)))_{.1}], S, M, A, C)}$$

$$L_{ca}\text{-E-Store}\ \frac{\bar{I}(pc) = st\ r_a\ r_s \quad addr(R(r_a)) \in Addr \quad col(R(r_a)) = M(addr(R(r_a)))_{.2} \quad I(pc+1)\ def}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{write(R(r_a))} (pc+1, R, S, M[addr(R(r_a)) \mapsto (R(r_s), *)], A, C)}$$

$$L_{ca}\text{-E-Alloc}\ \frac{\bar{I}(pc) = alloc\ r_b\ r_n \quad s := (b, R(r_n), i) \quad A \xrightarrow{alloc(s)} A' \quad I(pc+1)\ def}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{alloc(s)} (pc+1, R[r_b \mapsto b], S, M, A', C)}$$

$$L_{ca}\text{-E-Free}\ \frac{\bar{I}(pc) = free\ r_b \quad s := (R(r_b), n, i) \quad A \xrightarrow{free(s)} A' \quad I(pc+1)\ def}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{free(s)} (pc+1, R, S, M, A', C)}$$

$$L_{ca}\text{-E-Size}\ \frac{\bar{I}(pc) = size\ r_d\ r_b \quad s := (R(r_b), n, i) \quad A = (\bar{s}_A, \bar{s}_F, i_{\max}) \quad s \in \bar{s}_A \quad I(pc+1)\ def}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{\epsilon} (pc+1, R[r_d \mapsto n], S, M, A, C)}$$

$$L_{ca}\text{-E-Newc}\ \frac{\bar{I}(pc) = newc\ r_d \qquad c = next(C) \qquad a := addr(R(r_d))) \qquad I(pc+1)\ def}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{\epsilon} (pc+1, R[r_d \mapsto pair(a, c)], S, M, A, C \cup \{c\})}$$

$$L_{ca}\text{-E-Storec}\ \frac{\bar{I}(pc) = stc\ r_d\ c \qquad a := addr(R(r_d))) \qquad I(pc+1)\ def}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{\epsilon} (pc+1, R[r_d \mapsto pair(a, c)], S, M, A, C \cup \{c\})}$$

$$L_{ca}\text{-E-Setc}\ \frac{\begin{array}{c}\bar{I}(pc) = setc\ r_b\ r_n \qquad \forall j \in [0, R(r_n)).R(r_b)+j \in Addr \\ M' := M[R(r_b)+j \mapsto (*, col(R(r_b)))]_{j \in [0, R(r_n))} \qquad I(pc+1)\ def\end{array}}{\bar{I} \vdash (pc, R, S, M, A, C) \xrightarrow{\epsilon} (pc+1, R, S, M', A, C)}$$

We do not introduce a *trap* action because our goal is to catch every incorrect execution already at the $\mathbf{L_{cc}}$ level. So we tailor our target language to implement correct executions only; memory-unsafe programs never get compiled to L_{ca}.

The definition of memory safety for L_{ca} programs follows the usual route: we require the existence of an address matching function $\delta\colon [0, N) \times \mathbb{N} \rightharpoonup_{\mathrm{f}} \mathbb{L} \times \mathbb{C}$, with suitable validity conditions, and define the associated agreement relation $A \sim_\delta M$ between L_{ca} allocator states and abstract memory states. We also require the existence of a function $id\colon Col \rightharpoonup_{\mathrm{f}} \mathbb{N}$ that maps each concrete color to the corresponding slot identifier. Then we define the memory action relation $\alpha \sim_\delta \alpha$ by using the following rules:

$$L_{ca}\text{-TR-Read}\ \frac{i := id(col(x)) \qquad \delta(addr(x), i) = (\ell, c)}{read(x) \sim_\delta read\ \ell^c}$$

$$L_{ca}\text{-TR-Write}\ \frac{i := id(col(x)) \qquad \delta(addr(x), i) = (\ell, c)}{write(x) \sim_\delta write\ \ell^c}$$

$$L_{ca}\text{-TR-Alloc}\ \frac{s = (a, n, i) \qquad \delta(a, i) = (\ell, c) \qquad [n] = n}{alloc(s) \sim_\delta alloc\ n\ \ell^c}$$

$$L_{ca}\text{-TR-Free}\ \frac{s = (a, n, i) \qquad \delta(a, i) = (\ell, c) \qquad [n] = n}{free(s) \sim_\delta free\ n\ \ell^c}$$

Finally, we define memory safety for L_{ca} machine states and transitions in the usual way.

3.4.2.5 Secure compilation

First stage. As the two languages $\mathsf{L_c}$ and $\mathbf{L_{cc}}$ are syntactically the same, the compiler from $\mathsf{L_c}$ to $\mathbf{L_{cc}}$ is essentially a homomorphic translation. The compilation of expressions proceeds according to the following rules:

$$\text{C1-Val}\ \frac{\mathsf{e} = \mathsf{z} \qquad [\mathsf{z}] = [\mathbf{z}]}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{z}} \qquad \text{C1-Var}\ \frac{\mathsf{e} = \mathsf{x} \qquad [\mathsf{x}] = [\mathbf{x}]}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{x}}$$

$$\text{C1-Seq}\ \frac{\mathsf{e} = \mathsf{e}_1; \mathsf{e}_2 \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_1) = \mathbf{e_1} \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_2) = \mathbf{e_2}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{e_1}; \mathbf{e_2}}$$

$$\text{C1-BinOp}\ \frac{\mathsf{e} = \mathsf{e}_1 \odot \mathsf{e}_2 \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_1) = \mathbf{e_1} \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_2) = \mathbf{e_2}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{e_1} \odot \mathbf{e_2}}$$

$$\text{C1-Asgn}\ \frac{\mathsf{e} = \mathsf{x} := \mathsf{e}_1 \qquad [\mathsf{x}] = [\mathbf{x}] \qquad C^{\mathsf{c}}_{\mathbf{cc}}(e_1) = \mathbf{e_1}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{x} := \mathbf{e_1}}$$

$$\text{C1-Call}\ \frac{\mathsf{e} = \mathsf{x} := \mathsf{f}(\mathsf{e}_1 \qquad [\mathsf{x}] = [\mathbf{x}] \qquad [\mathsf{f}] = [\mathbf{f}] \qquad C^{\mathsf{c}}_{\mathbf{cc}}(e_1) = \mathbf{e_1}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{x} := \mathbf{f}(\mathbf{e_1})}$$

$$\text{C1-Cond}\ \frac{\mathsf{e} = \mathsf{if}\ \mathsf{e}_0\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_0) = \mathbf{e_0} \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_1) = \mathbf{e_1} \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_2) = \mathbf{e_2}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{if}\ \mathbf{e_0}\ \mathbf{then}\ \mathbf{e_1}\ \mathbf{else}\ \mathbf{e_2}}$$

$$\text{C1-PRead}\ \frac{\mathsf{e} = *\mathsf{e}_1 \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_1) = \mathbf{e_1}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := *\mathbf{e_1}}$$

$$\text{C1-PWrite}\ \frac{\mathsf{e} = *\mathsf{e}_1 := \mathsf{e}_2 \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_1) = \mathbf{e_1} \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_2) = \mathbf{e_2}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := *\mathbf{e_1} := \mathbf{e_2}}$$

$$\text{C1-Alloc}\ \frac{\mathsf{e} = \mathsf{malloc}\ \mathsf{e}_1 \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_1) = \mathbf{e_1}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{malloc}\ \mathbf{e_1}}$$

$$\text{C1-Free}\ \frac{\mathsf{e} = \mathsf{free}\ \mathsf{e}_1 \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}_1) = \mathbf{e_1}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e}) := \mathbf{free}\ \mathbf{e_1}}$$

In other words, the map $C^{\mathsf{c}}_{\mathbf{cc}}(\cdot)$ is defined by the obvious structural induction on $\mathsf{L_c}$ expressions; note that there is no need to define the compiler on pointer values, since such values are never found in a $\mathsf{L_c}$ source program (they exist only at runtime).

The compilation of a function definition is also straightforward, as described by the following rule:

$$\text{C1-FDef}\ \frac{\mathsf{F} = \mathsf{w_r}\ \mathsf{f}(\mathsf{w_p}\ \mathsf{x_p})\{(\overline{\mathsf{w_v}\ \mathsf{x_v}})\ \mathsf{e_b}\} \qquad [\mathsf{f}] = [\mathbf{f}] \qquad [\mathsf{x_p}] = [\mathbf{x_p}] \qquad [\mathsf{x_v}] = [\mathbf{x_v}] \qquad C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{e_b}) = \mathbf{e_b}}{C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{F}) := \mathbf{w_r}\ \mathbf{f}(\mathbf{w_p}\ \mathbf{x_p})\{(\overline{\mathbf{w_v}\ \mathbf{x_v}})\ \mathbf{e_b}\}}$$

which is then extended to modules (lists of function definitions) by a simple structural induction:

$$\text{C1-Nil}\ \frac{}{C^{\mathsf{c}}_{\mathbf{cc}}([]) := []} \qquad \text{C1-App}\ \frac{}{C^{\mathsf{c}}_{\mathbf{cc}}(\overline{\mathsf{F}_0} :: \mathsf{F}) := C^{\mathsf{c}}_{\mathbf{cc}}(\overline{\mathsf{F}_0}) :: C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{F})}$$

In order to show the desired security properties for $C^{\mathsf{c}}_{\mathbf{cc}}(\cdot)$, we need to be able to directly relate the two languages $\mathsf{L_c}$ and $\mathbf{L_{cc}}$ independently from the memory safety monitor. In order to do this, we will need the following additional constructions:

- a notion of $\mathsf{L_c}$--$\mathbf{L_{cc}}$ address matching function $\delta\colon [0, h_s)\times\mathbb{N} \rightharpoonup_{\mathrm{f}} [0, h_s)\times\mathbb{N}\times\mathbf{Col}$, with suitable validity conditions;
- a set of *cross-language relations*, relating equivalent values ($\mathsf{v} \sim_\delta \mathbf{v}$), expressions ($\mathsf{e} \sim_\delta \mathbf{e}$), local configurations ($\Theta \sim_\delta \boldsymbol{\Theta}$) and so on;
- a *memory action relation* $\alpha \sim_\delta \alpha$, which is then extended to traces of memory events in the obvious way;
- some lemmas relating all these ingredients to the memory safety properties of each language.

The fundamental lemma for $C^{\mathsf{c}}_{\mathbf{cc}}(\cdot)$ (``functional correctness'') may be formulated as follows.

Theorem 1. *Let $\Theta = (\theta, \mathsf{H}, \mathsf{A}, \mathsf{e})$ and $\Theta' = (\theta', \mathsf{H}', \mathsf{A}', \mathsf{e}')$ be $\mathsf{L_c}$ local configurations. Suppose that:*

1. *we have the single-step reduction $\vdash \Theta \xrightarrow{\alpha} \Theta'$;*
2. *$\Theta \sim_\delta \boldsymbol{\Theta}$ for some $\mathbf{L_{cc}}$ local configuration $\boldsymbol{\Theta} = (\theta, \mathbf{H}, \mathbf{A}, \mathbf{e})$ and valid $\mathsf{L_c}$-$\mathbf{L_{cc}}$ address matching function δ;*
3. *Θ is memory safe;*
4. *the reduction $\Theta \xrightarrow{\alpha} \Theta'$ is memory safe.*

Then there exists a valid $\mathsf{L_c}$-$\mathbf{L_{cc}}$ address matching function δ' extending δ, an $\mathbf{L_{cc}}$ memory action α and a $\mathbf{L_{cc}}$ local configuration $\boldsymbol{\Theta}'$ such that:

1. *$\alpha \sim_{\delta'} \alpha$,*
2. *$\vdash \boldsymbol{\Theta} \xrightarrow{\alpha} \boldsymbol{\Theta}'$,*
3. *$\Theta' \sim_{\delta'} \boldsymbol{\Theta}'$,*

4. Θ *is memory safe,*
5. *the reduction* $\Theta \xrightarrow{\alpha} \Theta'$ *is memory safe.*

The proof of this result proceeds by case analysis on the proof of hypothesis (4), the only non-trivial cases being the ones which involve a non-trivial memory action.

From the above theorem we can deduce the following secure compilation results for the first stage translation.

Theorem 2 (Memory safety preservation for first stage compiler)**.** *If* M *is a memory safe* $\mathsf{L_c}$ *program then* $C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{M})$ *is a memory safe* $\mathbf{L_{cc}}$ *program.*

Theorem 3 (Memory violation detection)**.** *Let* M *be a* $\mathsf{L_c}$ *program. Suppose that the execution of* M *produces a memory trace* $\overline{\alpha}$ *that is not memory safe. Then the execution of the* $\mathbf{L_{cc}}$ *program* $C^{\mathsf{c}}_{\mathbf{cc}}(\mathsf{M})$ *produces a memory trace* $\overline{\alpha}$ *that ends with* **trap**.

Second stage. The definition of the compiler from $\mathbf{L_{cc}}$ to L_{ca} is necessarily more involved, since the syntax and the execution model of the two languages are quite different. The compiler on $\mathbf{L_{cc}}$ expressions will be defined as a map

$$C_e : \mathbf{Expr} \times \mathbb{N} \to (\mathrm{List}\ Ins) \times \mathbb{N}$$

mapping a $\mathbf{L_{cc}}$ expression to a corresponding list of L_{ca} instructions. The second argument of C_e is the index of the first available pseudoregister, which is updated on every call to the compiler and returned as the second element of the resulting pair. The definition must guarantee that this parameter increments monotonically, i.e. that $C_e(\mathbf{e}, n)_{.2} \geq n$ for every $\mathbf{e} \in \mathbf{Expr}$.

We define the map C_e by structural induction on its first argument $\mathbf{e} \in \mathbf{Expr}$ by means of the following set of rules.

$$\text{C2-Val}\ \frac{\mathbf{e} = \mathbf{v} \qquad [v] = [\mathbf{v}]}{C_e(\mathbf{e}, n) = ([movi\ n\ v], n+1)}$$

$$\text{C2-Var}\ \frac{\mathbf{e} = \mathbf{x}}{C_e(\mathbf{x}, n) = ([mov\ n\ V(\mathbf{x})], n+1)}$$

$$\text{C2-Seq}\ \frac{\mathbf{e} = \mathbf{e_1}; \mathbf{e_2} \qquad C_e(\mathbf{e_1}, n) = (t_1, n_1) \qquad C_e(\mathbf{e_2}, n_1) = (t_2, n_2)}{C_e(\mathbf{e}, n) = (t_1 +\!\!+ t_2, n_2)}$$

$$\text{C2-Add}\ \frac{\mathbf{e} = \mathbf{e_1} + \mathbf{e_2} \qquad C_e(\mathbf{e_1}, n) = (t_1, n_1) \qquad C_e(\mathbf{e_2}, n_1) = (t_2, n_2)}{C_e(\mathbf{e}, n) := (t_1 +\!\!+ t_2 +\!\!+ [add\ n_2\ n_1 - 1\ n_2 - 1], n_2 + 1)}$$

$$\text{C2-Sub}\ \frac{\mathbf{e} = \mathbf{e_1} - \mathbf{e_2} \qquad C_e(\mathbf{e_1}, n) = (t_1, n_1) \qquad C_e(\mathbf{e_2}, n_1) = (t_2, n_2)}{C_e(\mathbf{e}, n) := (t_1 +\!\!+ t_2 +\!\!+ [sub\ n_2\ n_1 - 1\ n_2 - 1], n_2 + 1)}$$

$$\text{C2-Mul}\ \frac{\mathbf{e} = \mathbf{e_1} * \mathbf{e_2} \qquad C_e(\mathbf{e_1}, n) = (t_1, n_1) \qquad C_e(\mathbf{e_2}, n_1) = (t_2, n_2)}{C_e(\mathbf{e}, n) := (t_1 +\!\!+ t_2 +\!\!+ [mul\ n_2\ n_1 - 1\ n_2 - 1], n_2 + 1)}$$

$$\text{C2-Asgn}\ \frac{\mathbf{e} = \mathbf{x} := \mathbf{e_1} \qquad C_e(\mathbf{e_1}, n) = (t, n_1)}{C_e(\mathbf{e}, n) := (t +\!\!+ [mov\ V(\mathbf{x})\ n_1 - 1], n_1)}$$

$$\text{C2-Call}\ \frac{\mathbf{e} = \mathbf{x} := \mathbf{g}(\mathbf{e_1}) \qquad C_e(\mathbf{e_1}, n) = (t, n_1)}{C_e(\mathbf{e}, n) := (t +\!\!+ [mov\ 0\ n_1 - 1, call\ F(\mathbf{g}), mov\ V(\mathbf{x})\ 0], n_1)}$$

$$\text{C2-If}\ \frac{\begin{array}{c}\mathbf{e} = \textbf{if } \mathbf{e_0} \textbf{ then } \mathbf{e_1} \textbf{ else } \mathbf{e_2} \qquad\qquad C_e(\mathbf{e_0}, n) = (t_0, n_0) \\ C_e(\mathbf{e_1}, n_0) = (t_1, n_1) \qquad \ell_1 := \mathrm{len}(t_1) \qquad C_e(\mathbf{e_2}, n_1) = (t_2, n_2) \qquad \ell_2 := \mathrm{len}(t_2)\end{array}}{C_e(\mathbf{e}, n) := (t_0 +\!\!+ [jz\ n_0 - 1\ \ell_1 + 1] +\!\!+ t_1 +\!\!+ [jmp\ \ell_2] +\!\!+ t_2, n_2)}$$

$$\text{C2-PRead}\ \frac{\mathbf{e} = *\mathbf{e_1} \qquad C_e(\mathbf{e_1}, n) = (t, n_1)}{C_e(\mathbf{e}, n) := (t +\!\!+ [ld\ n_1\ n_1 - 1], n_1 + 1)}$$

$$\text{C2-PWrite}\ \frac{\mathbf{e} = *\mathbf{e_1} := \mathbf{e_2} \qquad C_e(\mathbf{e_1}, n) = (t_1, n_1) \qquad C_e(\mathbf{e_2}, n_1) = (t_2, n_2)}{C_e(\mathbf{e}, n) := (t_1 +\!\!+ t_2 +\!\!+ [st\ n_1 - 1\ n_2 - 1], n_2)}$$

$$\text{C2-Malloc}\ \frac{\mathbf{e} = \mathbf{malloc\ e_1} \qquad C_e(\mathbf{e_1}, n) = (t, n_1)}{C_e(\mathbf{e}, n) := (t +\!\!+ [alloc\ n_1\ n_1 - 1, newc\ n_1, setc\ n_1\ n_1 - 1], n_1 + 1)}$$

$$\text{C2-Free}\ \frac{\mathbf{e} = \mathbf{free\ e_1} \qquad C_e(\mathbf{e_1}, n) = (t, n_1)}{C_e(\mathbf{e}, n) := (t +\!\!+ [size\ n_1\ n_1 - 1, free\ n_1 - 1, stc\ n_1 - 1\ c_0, setc\ n_1 - 1\ n_1], n_1 + 1)}$$

In this definition some auxiliary functions appear:

- for each function identifier $\mathbf{f}$, a finite partial map $V_{\mathbf{f}} : \mathbf{Ident} \rightharpoonup_{\text{f}} \mathbb{N}$ whose purpose is to map each declared local variable in $\mathbf{f}$ to the index of the corresponding pseudoregister;
- a finite partial map $F : \mathbf{Ident} \rightharpoonup_{\text{f}} \mathbb{N}$ whose purpose is to map each function name in $\overline{\mathbf{F}}$ to its entry point in the corresponding compiled L_{ca} program.

These functions are defined by the module compilation process. Specifically, let $\mathbf{M}$ be an $\mathbf{L_{cc}}$ module defining the list of functions $\overline{\mathbf{F}}$. For each $\mathbf{f} \in \overline{\mathbf{F}}$, we define the helper function $V_{\mathbf{f}} : \mathbf{Ident} \rightharpoonup_{\text{f}} \mathbb{N}$ by reserving pseudoregister 0 for the function argument, and pseudoregisters $1, \ldots, n$ for the n local variables declared by the functions. Then we compile the body of the function (with first available pseudoregister $n+1$), and we append an epilogue that puts the result in register zero and calls the *ret* instruction.

$$\text{C2-FDef}\ \frac{\mathbf{F} = \mathbf{w_r\ f(w_p\ x_p)\{(\overline{w_v\ x_v})\ e_b\}} \qquad n := \text{len}(\overline{\mathbf{x_v}}) \qquad V_{\mathbf{f}} := \emptyset[\mathbf{x_p} \mapsto 0, \overline{\mathbf{x_v}} \mapsto (1, \ldots, n)] \qquad C_e(\mathbf{e_b}, n+1) = (t, m)}{C_{ca}^{\mathbf{cc}}(\mathbf{F}) := t +\!\!+ [mov\ 0\ m - 1, ret]}$$

Finally, we define the two maps $C_{ca}^{\mathbf{cc}}(\cdot)$ (compilation of $\mathbf{L_{cc}}$ modules) and $F(\cdot)$ by simultaneous induction, as follows:

$$\text{C2-Nil}\ \frac{}{C_{ca}^{\mathbf{cc}}([]) := []} \qquad \text{C2-App}\ \frac{}{C_{ca}^{\mathbf{cc}}(\overline{\mathbf{F_0}} :: \mathbf{F}) := C_{ca}^{\mathbf{cc}}(\overline{\mathbf{F_0}}) :: C_{ca}^{\mathbf{cc}}(\mathbf{F})}$$

$$\text{F-Nil}\ \frac{}{F([]) := \emptyset} \qquad \text{F-App}\ \frac{\mathbf{F} = \mathbf{w_r\ f(w_p\ x_p)\{(\overline{w_v\ x_v})\ e_b\}} \qquad F(\overline{\mathbf{F_0}}) = F_0 \qquad C_{ca}^{\mathbf{cc}}(\overline{\mathbf{F_0}}) = (t, n)}{F(\overline{\mathbf{F_0}} :: \mathbf{F}) := F_0[\mathbf{f} \mapsto \text{len}(t)]}$$

This concludes the definition of the compiler from $\mathbf{L_{cc}}$ to L_{ca}.

To relate the semantic structures of the two languages $\mathbf{L_{cc}}$ and L_{ca} we need again the usual additional constructions:

- a notion of $\mathbf{L_{cc}}$--L_{ca} address matching function $\delta : [0, h_s) \times \mathbb{N} \rightharpoonup_{\text{f}} [0, N) \times \mathbb{N}$, with suitable validity conditions;
- a suitable set of *cross-language relations* $\mathbf{v} \sim_\delta v$, $\mathbf{\Theta} \sim_\delta \Omega$;
- a *memory action relation* $\alpha \sim_\delta \alpha$.

We can then prove a functional correctness result for the second-stage compiler $C_{ca}^{\mathbf{cc}}(\cdot)$. The formulation is similar to the one used in lemma 1, the only difference being the fact that now a single $\mathbf{L_{cc}}$ transition may be related to multiple L_{ca} transitions.

Theorem 4. *Let* $\mathbf{\Theta} = (\theta, \mathbf{H}, \mathbf{A}, \mathbf{e})$ *and* $\mathbf{\Theta'} = (\theta', \mathbf{H'}, \mathbf{A'}, \mathbf{e'})$ *be* $\mathbf{L_{cc}}$ *local configurations. Suppose that:*

1. *we have the single-step reduction* $\vdash \mathbf{\Theta} \xrightarrow{\alpha} \mathbf{\Theta'}$;
2. $\mathbf{\Theta} \sim_\delta \Omega$ *for some* L_{ca} *machine state* $\mathbf{\Omega}$ *and valid* $\mathbf{L_{cc}}$-L_{ca} *address matching function* δ;
3. $\mathbf{\Theta}$ *is memory safe;*
4. *the reduction* $\mathbf{\Theta} \xrightarrow{\alpha} \mathbf{\Theta'}$ *is memory safe.*

Then there exists a valid $\mathbf{L_{cc}}$-L_{ca} *address matching function* δ' *extending* δ*, a sequence of* L_{ca} *memory actions* $\overline{\alpha}$ *and a* L_{ca} *machine state* $\mathbf{\Omega}'$ *such that:*

1. $\alpha \sim_{\delta'} \overline{\alpha}$,
2. $\vdash \Omega \xrightarrow{\overline{\alpha}}{}^{*} \Omega'$,
3. $\mathbf{\Theta}' \sim_{\delta'} \Omega'$,
4. Ω *is memory safe,*
5. *the reduction* $\Omega \xrightarrow{\alpha}{}^{*} \Omega'$ *is memory safe.*

The proof proceeds again by case analysis on the possible proofs of hypothesis (4).

Finally, we use the above lemma to prove the following memory safety preservation result.
Theorem 5 (Memory safety preservation for second stage compiler)**.** *If* $\mathbf{M}$ *is a memory safe* $\mathbf{L_{cc}}$ *program then* $C_{ca}^{\mathbf{cc}}(\mathbf{M})$ *is a memory safe* L_{ca} *program.*

Main result. Define $C(\mathsf{M}) := C_{ca}^{\mathbf{cc}}(C_{\mathbf{cc}}^{\mathsf{c}}(\mathsf{M}))$. Putting together theorems 2 and 5, we obtain the main result of our formalization.
Theorem 6 (Memory safety preservation)**.** *For every* $\mathsf{L_c}$ *program* M*, if* M *is memory safe then* $C(\mathsf{M})$ *is memory safe.*

3.4.3 DevSecOps Toolchain Integration: Implementation Phase

Following the initial planning and requirements analysis for the DevSecOps integration in the CROSSCON toolchain, this phase focused on establishing the technical foundations required for secure and automated software development workflows. The primary objective was to deploy, configure, and prepare the necessary DevSecOps infrastructure on the CROSSCON testbed, covering both continuous integration (CI) and security evaluation capabilities.

3.4.3.1 Deployment of DevSecOps Components

During this phase, the essential tools for enabling a DevSecOps workflow were installed and made available on the CROSSCON testbed server:

- **Jenkins (v2.452.2)**: Jenkins was deployed on the CROSSCON testbed to provide the core CI/CD engine for building, managing, and (in the future) automating the deployment of CROSSCON firmware and component updates. Access to Jenkins is secured via SSH port forwarding, allowing authorized developers to connect remotely and initiate jobs for artifact creation or (test) deployment.
- **DeltAICert**: The DeltAICert tool was made available on the same testbed, offering targeted delta security evaluation as described in previous deliverables. This tool supports automated assessment of security requirements, with a focus on analyzing changes (delta) since the last baseline and verifying critical assets, such as the SBOM of firmware update packages.

For both tools, secure access is provided to consortium members through SSH tunnels, ensuring that sensitive development and testing environments remain protected.

3.4.3.2 State of Integration

While the core components necessary for DevSecOps (Jenkins for CI/CD and DeltAICert for change-driven security assessment) were successfully deployed on the CROSSCON testbed, the full integration and automation of workflows was not completed in this phase. This was due to two key factors:

1. **Toolchain Maturity (TRL Level)**: The CROSSCON toolchain components and associated automation scripts remain at a low Technology Readiness Level (TRL), with many features still in late development. As a result, stable, end-to-end CI/CD automation and validation could not be finalized within the timeframe of this deliverable.
2. **Late Delivery and Maturity of Validation Components**: Several validation and automation modules necessary for integrated DevSecOps workflows became available only late in the reporting period or are still undergoing testing and adaptation for the specific embedded testbed setup.

Consequently, the technical groundwork for a robust DevSecOps pipeline has been laid, but the actual integration and execution of automated workflows, including seamless coordination between Jenkins, DeltAICert, and hardware-in-the-loop testing, remains pending.

3.5 CROSSCON Bare-Metal TEE

In computer science, the term bare-metal commonly refers to systems in which applications run directly on the hardware, without the support of an underlying OS. While this approach can improve performance on high-end systems, it is often a necessity on very low-end devices such as MCUs that lack the resources needed to support a fully fledged OS. The IoT ecosystem comprises a vast number of such constrained devices for which bare-metal execution is the only viable option. These systems constitute a distinct class of computing platforms designed to run applications directly on hardware, without the mediation of a kernel or OS. Applications on bare-metal devices are typically responsible for both hardware initialization and the execution of core functionalities.

Bare-metal devices are widely adopted in both industrial and research settings for many reasons including power efficiency and cost-effectiveness. However, they typically offer very limited functionalities, often foregoing architectural features like memory virtualization and caching. These limitations have an impact on the security capabilities of these devices: bare-metal systems generally lack advanced hardware-based security features found in higher-end devices, such as Trusted Platform Module (TPM), MMU and hardware TEEs. These features often play a crucial role for the establishment of a secure computing environment, but incur sensible cost and increase the complexity of the system. Consequently, the responsibility for ensuring system security in bare-metal environments largely falls to the applications themselves, which must implement appropriate safeguards.

One of the goals of the CROSSCON project is to increase the security guarantees for a wide spectrum of devices, comprising both high- and low-end systems. Even though the CROSSCON Hypervisor can be deployed on some low-end devices to instantiate common trusted OSes, its architecture is not compatible with the restrictions commonly found on low-end bare-metal devices. Therefore, we propose a new TEE specifically designed for bare-metal devices to allow such systems to interact securely with the rest of the CROSSCON stack.

Bare-metal devices can span across a wide variety of architectures, each with different hardware features and capabilities. To demonstrate the flexibility of the CROSSCON bare-metal TEE, we propose a prototype for three different architectures: MSP430, Armv7-M and RISC-V. Notably, these architectures cover two different classes, with a distinct set of security features available. Specifically, Armv7-M offers: (i) two privilege levels, (ii) an MPU, (iii) an amount of memory in the order of MB. Similarly, we chose a RISC-V low-end architectures with a PMP and only two privilege levels. On the other hand, the MSP430 offers no MPU, no privilege levels and only hundreds of KB of memory.

3.5.1 Security requirements for bare-metal platforms

In order to offer adequate security guarantees and functionalities, there are some key features that must be supported by any TEE. Some of these features can be mapped to security primitives that must

be enabled by the hardware or the software itself. Given the constrained hardware on the bare-metal devices, we propose to replace such primitives with software-based solutions.

We consider the following set of requirements to be essential for a bare-metal TEE in order to be considered secure.

1. *Memory Isolation*: this property ensures that the entire untrusted portion of the software cannot compromise nor leak the memory of the trusted part. This requirement is crucial since its absence would allow a compromised CA to corrupt the software TEE, thus breaking its security.
2. *Privilege Separation*: the untrusted software cannot execute arbitrary operations with the privilege level enjoyed by the TEE. Without such a guarantee, the untrusted software could re-configure the TEE at its pleasure and void the security guarantees it provides.
3. *Inter-Domain Communication*: The TEE and the untrusted software must be able to communicate. In practice, this means that the TEE provides a set of APIs that the untrusted software can invoke. Conversely, the TEE typically has full control over the untrusted software and can interact with it as needed.

3.5.2 CROSSCON Baremetal TEEs

In order to meet the security requirements of CROSSCON, we propose two different bare-metal TEEs: BareTEE-noMPU and BareTEE-MPU. From a high-level perspective, both TEEs act as a middle ground between the features offered by the CROSSCON Hypervisor and the bare-metal devices. Given the constrained resources of the latter, running the full hypervisor on them is not feasible. For this reason, the two BareTEEs strive to provide a reduced, yet comprehensive set of security features to satisfy the above-mentioned minimal requirements of memory isolation, privilege separation, and cross-domain intercommunication. The BareTEE-MPU can be deployed in the presence of an MPU or a PMP, whereas in the absence of such hardware modules we can use the BareTEE-noMPU version.

As part of the CROSSCON project we provide three prototypes: we implement the MPU version for the Armv7-M and RISC-V architectures and noMPU version for the MSP430 architecture. Although both solutions provide a similar set of security guarantees to the applications, they do so using a slightly different isolation model. Figure 44 shows the difference between the two versions, which will be highlighted in the following paragraphs.

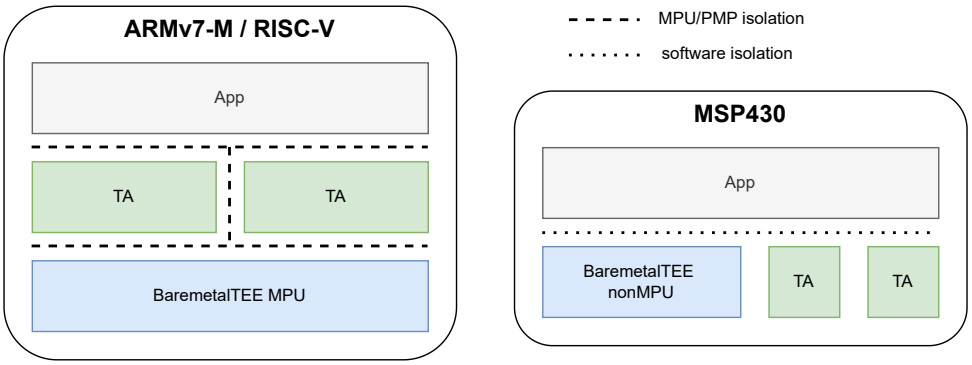


Figure 44: High-level comparison between the memory isolation provided by the two different bare-metal TEEs.

3.5.2.1 BareTEE-MPU

The MPU and the PMP are a hardware component that allows basic security features for memory protection. If, on the one hand, such a components are not as flexible as the MMU, on the other hand they still enable a good degree of security on controlling access to memory. In particular, the MPU and the PMP define a limited number of memory regions onto which some memory access privileges (R/W/X) can be enforced. Regions can overlap, and the same memory address can have different access rights depending on the execution privilege. Notably, contrarily to the MMU that assigns memory areas to specific processes/applications, the MPU and PMP do not enforce the concept of memory ownership. Consequently, defining different access privileges for different entities sharing the same system, e.g. the application and the TEE, is also challenging.

Nevertheless, we propose a fully fledged TEE for bare-metal devices with either an MPU or a PMP, providing an implementation for the ARMv7-M and RISC-V architecture [201]. Notably, our design and implementation are built around the GlobalPlatform (GP) API, with the TEE supporting a subset of the Core API and all of the Client API. This enables interoperability of TAs and CAs across different architectures and devices.

Memory isolation. We define three different entities: Client Application (CA), Trusted Application (TA) and the bare-metal TEE itself. The bare-metal TEE comprises the software that manages both the TAs and the CA, as well as the core that handles the memory isolation between the three entities (as can be seen in Figure 44). Notably, the memory isolation goes only one direction, with the TA that can access the memory of the CA, and the bare-metal TEE that can access the memory of the entire system, but not vice versa. Furthermore, on top of the vertical isolation, the bare-metal TEE enforces an horizontal separation between the different TAs, preventing them from accessing each others' memory. The memory isolation is enabled by the MPU or the PMP, which need to be configured dynamically on each domain switch. Specifically, upon execution of any of the three entities, the bare-metal TEE configures these hardware security primitives accordingly to enforce our isolation constraints. In more detail:

- **CA Execution**: whenever the user application is being executed, the MPU/PMP is configured to only allow access to the application memory. Accesses to the memory of the TAs and of the bare-metal TEE is prevented.
- **TA execution**: whenever a security service from a TA is invoked, the MPU/PMP is configured to allow access to the memory reserved to the specific TA and to the entire application memory. Accesses to the memory regions belonging to the other TAs or to the TEE itself are disabled.
- **Bare-metal TEE execution**: whenever the bare-metal TEE is run, the MPU/PMP is configured to allow access to the entire memory.

Figure 45 summarises the memory access rights of each component.

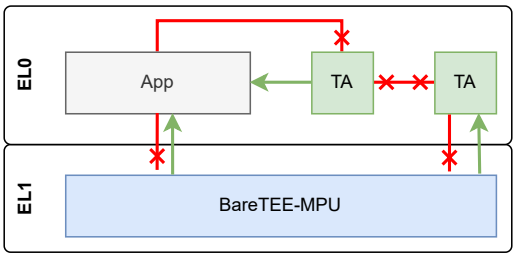


Figure 45: Memory isolation enforced between the three entities on a BareTEE-MPU system: Application, TA and bare-metal TEE. Each arrow symbolises the access capabilities of an entity over the memory of the pointed entity.

Privilege separation. ARMv7-M and our chosen RISC-V platform support two privilege levels: unprivileged and privileged, technically referred to on ARMv7-M as EL0 and EL1, and on RISC-V as User (U) and Machine (M). On a high-end system equipped with more than 2 privilege levels, each software entity would run at a different level. In this case, we must reserve the privileged level for our bare-metal TEE and share the unprivileged between TAs and CA. This ensures that no entity other than the bare-metal TEE can perform privileged operations without its consent. By managing the MPU/PMP appropriately and by offering a proper set of APIs, following the TEE Abstraction level proposed in Section 3.1.5, the TAs and CA can coexist at the unprivileged level. The bare-metal TEE also carefully manages the interrupts and software exceptions to prevent any privilege escalation from TAs and CA.

Inter-domain communication. Communication between the three domains is modeled with our TEE Abstraction model presented in Section 3.1.5. This set of APIs, which facilitate the interaction between the three software entities, is managed by the bare-metal TEE. While the bare-metal TEE can freely call and jump to the TA and CA, lowering the execution privilege, both TA and CA need to trigger an exception with each API request. This is handled transparently by the bare-metal TEE, that parses the exception and routes the request to the desired functionality.

3.5.2.2 BareTEE-noMPU

In the absence of basic hardware security support, e.g. the MPU, we must resort to an extremely compact and stripped down TEE providing only the most basic security services. The goal is to have a solution that is compatible with the use cases of such constrained devices while allowing some degree of protection in line with CROSSCON requirements [202].

Memory isolation. The fundamental security primitive that is required by a TEE is memory isolation, i.e. the separation of the memory used by the TEE itself from the memory used by the untrusted application. This ensures that the security services running in the TEE can be protected from attackers compromising the untrusted application, whose malicious behaviour is confined in the unsecured part of the memory.

Without an MPU, memory isolation must be implemented purely in software through software instrumentation and instruction virtualization. The instrumentation consists in adding, modifying and removing instructions from the original application in order to make it adhere to a certain security policy. Notably, the instrumentation alters the semantics of the code but not its functionalities, unless these are explicitly and deterministically malicious (in which case they are simply blocked). However, there are cases in which the outcome of an instruction cannot be known at compile time, but only at run-time where it could turn out malicious. In order to prevent the exploitation of these dynamic instructions we resort to a virtualization technique. In particular, we replace these dynamic instructions with calls to specific TEE functions that emulate their functions. As part of this emulation, these functions also make sure that the outcome does not violate the security policy. Notably, since the application is considered untrusted once deployed, these functions are located and executed in the TEE (exposed via APIs) so that the security checks cannot be circumvented. Figure 47 shows the instrumentation/virtualization technique.

As a result, we can deploy a secure binary, i.e. properly instrumented, alongside the TEE without allowing attackers to compromise the security services deployed within it. Interestingly, there are several challenges in maintaining this isolation in real-world scenarios. The most important among them is the handling of interrupts, a common feature that preempts the execution of the currently executing function to execute a priority task. Interrupts are widely used in embedded systems and can be used to disrupt the functionalities of the TEE when triggered during its execution. To prevent that, we supervise the handling of interrupts by making sure that the TEE is ready to give up control to the application upon

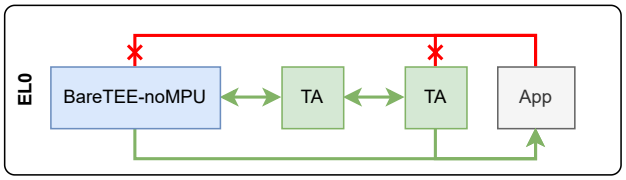


Figure 46: Isolation enforced between the three entities on a BareTEE-noMPU system: Application, TA and bare-metal TEE. The arrows show the access capabilities of the starting entity over the memory of the pointed entity.

the triggering of an interrupt. We achieve this by instrumenting the Interrupt Service RoutineInterrupt Service Routines (ISRs) with dedicated routines that clear any sensitive data from memory.

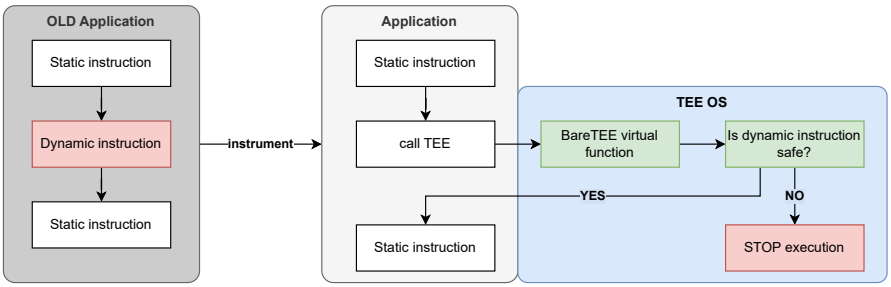


Figure 47: Representation of the instrumentation/virtualization technique of BareTEE-noMPU.

Privilege separation. The MSP430 architecture does not provide different privilege levels, thus making the separation of privileges a challenging task. To compensate for the lack of privilege levels, we leverage a code verification technique. Specifically, the bare-metal TEE inspect the binary of the deployed CA to detect any attempt to execute privileged operations. This check is only performed at boot and prevents the application from starting if any such instruction is found.

Inter-domain communication. Similarly to the MPU version, the BareTEE-noMPU employs a set of APIs to allow communication between the TAs and the CA. However, due to the much more constrained resources of the MSP430 devices, we reduced the set of available APIs to a bare minimum and simplified them. This allows us to offer strong security guarantees without increasing the complexity of the code. The TEE Abstraction Model can still be mapped to the APIs of this implementation, although with some limitations.

4 Conclusions

The IoT ecosystem is inherently heterogeneous, encompassing devices that range from low-power MCUs with minimal security features to high-performance, multi-core APUs equipped with reconfigurable hardware. A typical IoT system spans multiple layers—hardware, firmware, and the OS—each contributing to complexity and expanding the attack surface. This diversity brings significant security challenges, including: the lack of interoperability and isolation across different TEE implementations; the difficulty of providing dynamic, per-VM services without enlarging the TCB; the absence of novel trusted services; the challenges of secure firmware updates and cross-compilation; and the limited security capabilities of bare-metal devices.

Within WP3 of the CROSSCON project, we addressed these challenges through the development of the CROSSCON Open Security Stack. This document has presented the research and development outcomes of this work.

The process began with a platform selection study (Section 2), resulting in the choice of *Bao* as the hypervisor foundation for the CROSSCON Hypervisor. TEE isolation and abstraction were explored in Section 3.1, while Section 3.2 detailed the CROSSCON Hypervisor design and its key features, including dynamic virtual machine creation and per-VM TEE service support.

Novel trusted services, described in Section 3.3, were also developed, including PUF-based authentication, remote attestation, FPGA-related services, behavioral analysis services, and control flow integrity mechanisms. Furthermore, the CROSSCON TEE Toolchain (Section 3.4) was designed to support secure update mechanisms and integration with DevSecOps platforms. Finally, Section 3.5 presented the CROSSCON Bare-Metal TEE, covering requirements, platform evaluation, and implementation, including both MPU and non-MPU variants.

Overall, this deliverable consolidates the research and development results of WP3, presenting the components of the CROSSCON Open Security Stack: the CROSSCON Hypervisor, novel trusted services, the TEE Toolchain, and the Bare-Metal TEE.

References

[1] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. ``"SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems''. In: *Proc. of S&P*. 2020.

[2] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. ``ReZone: Disarming TrustZone with TEE Privilege Reduction''. In: *Proc. of USENIX Security*. 2022.

[3] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang. ``TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms''. In: *Proc. of VEE*. 2019.

[4] Matthieu Bettinger, Etienne Rivière, Sonia Ben Mokhtar, and Anthony Simonet-Boulogne. ``COoL-TEE: Client-TEE Collaboration for Resilient Distributed Search''. In: *2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2025.

[5] CC Consortium et al. ``Confidential Computing: Hardware-Based Trusted Execution for Applications and Data''. In: *A Publication of The Confidential Computing Consortium* (2021).

[6] Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. ``SoK: Hardware-supported Trusted Execution Environments''. Working Paper. 2022.

[7] Ozgu Can, Fursan Thabit, Asia Othman Aljahdali, Sharaf Al-Homdy, and Hoda A Alkhzaimi. ``A comprehensive literature of genetics cryptographic algorithms for data security in cloud computing''. In: *Cybernetics and Systems* (2025).

[8] Sandro Pinto and Nuno Santos. ``Demystifying Arm TrustZone: A Comprehensive Survey''. In: *ACM Comput. Surv.* (2019).

[9] Arm. *Arm Architecture Reference Manual for A-profile architecture*.

[10] TrustedFirmware. *Hafnium Hypervisor*.

[11] Arm. *Arm Firmware Framework for Arm A-profile*.

[12] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. ``Design and Verification of the Arm Confidential Compute Architecture''. In: *Proc. of USENIX OSDI*. 2022.

[13] V. Costan and S. Devadas. ``Intel SGX Explained''. In: *IACR Cryptology ePrint Archive* (2016).

[14] Intel. *Intel Trust Domain Extensions (Intel TDX)*. www.www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. 2021.

[15] AMD. *AMD Secure Encrypted Virtualization*. www.developer.amd.com/sev/. 2019.

[16] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. ``CoVE: Towards Confidential Computing on RISC-V Platforms''. In: *Proc. of ACM International Conference on Computing Frontiers*. 2023.

[17] RISC-V Foundation. *Privileged Architecture v1.12, Ratified*. www.github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12. 2021.

[18] Arm. *TrustZone for Cortex-M.*

[19] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. ``The ANDIX research OS—ARM TrustZone meets industrial control systems security''. In: *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*. 2015.

[20] Xi Tan, Zheyuan Ma, Sandro Pinto, Le Guan, Ning Zhang, Jun Xu, Zhiqiang Lin, Hongxin Hu, and Ziming Zhao. ``SoK: Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems''. In: *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*. USENIX Association, 2024.

[21] Android. *"Full-disk encryption"*. [online] Available at: https://source.android.com/docs/security/features/encryption/full-disk.

[22] FreeRTOS. *"Real-time operating system for microcontrollers and small microprocessors"*. [online] Available at: https://www.freertos.org/.

[23] Zephyr. *"Zephyr Project Documentation"*. [online] Available at: https://docs.zephyrproject.org/latest/index.html.
[24] A. Muñoz, R. Ríos, R. Román, and J. López. ``"A survey on the (in)security of trusted execution environments"''. In: *Computers & Security*. 2023.
[25] M. Busch, J. Westphal, and T. Mueller. ``"Unearthing the TrustedCore: A Critical Review on Huawei's Trusted Execution Environment"''. In: *Proc. of USENIX WOOT*. 2020.
[26] F. Khalid and A. Masood. ``"Vulnerability analysis of Qualcomm Secure Execution Environment (QSEE)"''. In: *Computers & Security*. 2022.
[27] Samsung. *"Samsung TEEGRIS"*. [online] Available at: https://developer.samsung.com/teegris/overview.html.
[28] T. Drozdovskyi and O. Moliavko. ``"mTower: Trusted Execution Environment for MCU-based devices"''. In: *Journal of Open Source Software*. 2019.
[29] Google. *"Trusty"*. [online] Available at: https://source.android.com/docs/security/features/trusty.
[30] TrustedFirmware. *"Trusted Firmware Projects"*. [online] Available at: https://www.trustedfirmware.org/.
[31] TrustedFirmware. *"OP-TEE"*. [online] Available at: https://www.trustedfirmware.org/projects/op-tee/.
[32] TrustedFirmware. *"Trusted Firmware-M Documentation"*. [online] Available at: https://www.trustedfirmware.org/projects/tf-m/.
[33] Trustonic. *"Kinibi-520a: The latest Trustonic Trusted Execution Environment (TEE)"*. [online] Available at: https://www.trustonic.com/technical-articles/kinibi-520a-the-latest-trusted-execution-environment-tee/. 2021.
[34] Trustonic. *"Kinibi-M"*. [online] Available at: https://www.trustonic.com/opinion/not-just-droning-rise-kinibi-m/.
[35] NVIDIA. *"Understanding Security"*. [online] Available at: https://docs.nvidia.com/drive/drive-os-5.2.0.0L/drive-os/index.html\#page/DRIVE_OS_Linux_SDK_Development_Guide/Security.
[36] TrustedFirmware. *"Trusted Little Kernel (TLK) Dispatcher"*. [online] Available at: https://trustedfirmware-a.readthedocs.io/en/v2.11/components/spd/tlk-dispatcher.html.
[37] AMD. *"AMD-TEE (AMD's Trusted Execution Environment)"*. [online] Available at: https://docs.kernel.org/tee/amd-tee.html.
[38] TsinglinkCloud. *"Qinglian Cloud's TinyTEE"*. [online] Available at: https://www.qinglianyun.com/Front/Secure/safe.
[39] GlobalPlatform. *"TEE internal core API specification"*. [online] Available at: https://www.qinglianyun.com/Front/Secure/safe.
[40] M. Busch, P. Mao, and M. Payer. ``"GlobalConfusion: TrustZone Trusted Application 0-Days by Design"''. In: *Proc. of USENIX Security*. 2024.
[41] CVE. *"Common Vulnerabilities and Exposures"*. [online] Available at: https://cve.mitre.org/about/.
[42] CVE. *"CVE Numbering Authorities"*. [online] Available at: https://www.cve.org/ProgramOrganization/CNAs.
[43] A. Adamski, J. Guilbon, and M. Peterlin. *"A Deep Dive Into Samsung's TrustZone"*. [online] Available at: https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-1.html. 2019.
[44] I hate software. *"Reverse-engineering Samsung Exynos 9820 bootloader and TZ"*. [online] Available at: https://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html. 2019.
[45] Blue Frost Security. *"TEE Exploitation on Samsung Exynos devices"*. [online] Available at: https://labs.bluefrostsecurity.de/blog/2019/05/27/tee-exploitation-on-samsung-exynos-devices-introduction/. 2019.

[46] F. Menarini. *"Breaking TEE Security Part 2: Exploiting Trusted Applications (TAs)"*. [online] Available at: https://www.riscure.com/tee-security-samsung-teegris-part-2/. 2019.
[47] S. Makkaveev. *"The Road to Qualcomm TrustZone Apps Fuzzing"*. [online] Available at: https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/. 2019.
[48] Samsung Mobility Security. *"Security Updates"*. [online] Available at: https://security.samsungmobile.com/securityUpdate.smsb.
[49] Nvidia. *"Product Security"*. [online] Available at: https://www.nvidia.com/en-us/product-security/.
[50] Qualcomm. *"Security Bulletin"*. [online] Available at: https://docs.qualcomm.com/product/publicresources/securitybulletin.
[51] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel, and M. Payer. ``"TEEzz: Fuzzing Trusted Applications on COTS Android Devices"''. In: *Proc. S&P)*. 2023.
[52] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace. ``"PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation"''. In: *Proc. of USENIX Security*. 2020.
[53] G. Duan, Y. Fu, B. Zhang, P. Deng, J. Sun, H. Chen, and Z. Chen. ``"TEEFuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation"''. In: *Future Gener. Comput. Syst.* 2023.
[54] Q. Wang, B. Chang, S. Ji, Y. Tian, X. Zhang, B. Zhao, G. Pan, C. Lyu, M. Payer, W. Wang, and R. Beyah. ``"SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices"''. In: *Proc. of IEEE S&P*. 2024.
[55] T. Roth. *"TrustZone-M(eh): Breaking ARMv8-M's security"*. [online] Available at: https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security. 2018.
[56] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. ``BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect''. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024.
[57] X. Saß, R. Mitev, and A. Sadeghi. ``"Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M"''. In: *Proc. of USENIX Security*. 2023.
[58] K. Ryan. ``"Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone"''. In: *Proc. of ACM CCS*. 2019.
[59] FIRST.org. *"Common Vulnerability Scoring System version 3.0 and 3.1: Specification Document"*. [online] Available at: https://www.first.org/cvss/specification-document.
[60] F. Menarini. *"Breaking TEE Security Part 1: TEEs, TrustZone and TEEGRIS"*. [online] Available at: https://www.riscure.com/tee-security-samsung-teegris-part-1/. 2019.
[61] Scopelliti G. *"Secure Integration of Trusted Execution Environments into Distributed Applications"*. [online] Available at: https://research.kuleuven.be/portal/en/project/3E200948. 2025.
[62] M. Gross, N. Jacob, A. Zankl, and G. Sigl. ``"Breaking TrustZone Memory Isolation through Malicious Hardware on a Modern FPGA-SoC"''. In: *Proc. of ASHES*. 2019.
[63] S. Pinto and C. Rodrigues. ``"What the TrustZone-M Doesn't See, the MCU Does Grieve Over: Lessons Learned"''. In: *Proc. of Black Hat Asia*. [online] Available at: https://www.youtube.com/watch?v=o_-a-_oqCgU. 2024.
[64] A. Barbosa, C. Rodrigues, T. Gomes, and S. Pinto. ``"WiP Paper: BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices"''. In: *Proc. of EWSN*. 2024.
[65] Jubayer Mahmod and Matthew Hicks. ``"UnTrustZone: Systematic Accelerated Aging to Expose On-chip Secrets"''. In: *Proc. of IEEE S&P*. 2024.
[66] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. ``vTZ: Virtualizing ARM TrustZone''. In: *Proc. of USENIX Security*. 2017.
[67] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek. ``PrOS: Light-weight Privatized Secure OSes in ARM TrustZone''. In: *TMC* (2019).
[68] Seung-Kyun Han and Jinsoo Jang. ``MyTEE: Own the Trusted Execution Environment on Embedded Devices.'' In: *Proc. of NDSS*. 2023.

[69] Borna Blazevic, Michael Peter, Mohammad Hamad, and Sebastian Steinhorst. ``TEEVseL4: Trusted Execution Environment for Virtualized seL4-based Systems''. In: *Proc. of RTCSA*. 2023.
[70] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. ``uTango: an open-source TEE for IoT devices''. In: *IEEE Access* (2022).
[71] GlobalPlatform. *TEE Internal Core API Specification v1.3.1*. 2021. URL: https://globalplatform.org/specs-library/tee-internal-core-api-specification/.
[72] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. ``Komodo: Using verification to disentangle secure-enclave hardware from software''. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2017.
[73] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. ``HyperEnclave: An Open and Cross-platform Trusted Execution Environment''. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, 2022.
[74] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. ``Keystone: an open framework for architecting trusted execution environments''. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. Association for Computing Machinery, 2020.
[75] Arm. *Arm Holdings plc FYE24-Q2 Results Presentation*. Slide 18, https://investors.arm.com/static-files/b43123f6-9236-4e3c-bd7f-05c3aa2b26c1.
[76] Arm Ltd. *Armv8-A virtualization*. 2019.
[77] Arm Ltd. *Arm Generic Interrupt Controller v2*. 2013.
[78] Arm Ltd. *Arm Generic Interrupt Controller v3 and v4 - Virtualization*. 2022.
[79] Arm Ltd. *Arm System Memory Management Unit Architecture Specification SMMU architecture version 2*. 2016.
[80] Arm Ltd. *Arm System Memory Management Unit Architecture Specification SMMU architecture version 3*. 2023.
[81] RISC-V Foundation. *RISC-V*. https://www.riscv.org/.
[82] Andrew Waterman1, Krste Asanovi, John Hauser. *The RISC-V Instruction Set Manual - Volume II: Privileged Architecture*. [online] Available at: https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view. 2025.
[83] Bruno Sá, José Martins, and Sandro Pinto. ``A First Look at RISC-V Virtualization From an Embedded Systems Perspective''. In: *IEEE Transactions on Computers* (2022).
[84] Bruno Sá, Luca Valente, José Martins, Davide Rossi, Luca Benini, and Sandro Pinto. ``CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration''. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023).
[85] RISC-V Task Group. *RISC-V Platform-Level Interrupt Controller Specification*. 2023.
[86] João Sousa, José Martins, Tiago Gomes, and Sandro Pinto. ``HSP-V: Hypervisor-Less Static Partitioning for RISC-V COTS Platforms''. In: *IEEE Access* (2024).
[87] John Hauser. *The RISC-V Advanced Interrupt Architecture*. 2023.
[88] Francisco Marques, Manuel Rodríguez, Bruno Sá, and Sandro Pinto. ``“Interrupting” the Status Quo: A First Glance at the RISC-V Advanced Interrupt Architecture (AIA)''. In: *IEEE Access* (2024).
[89] IOMMU Task Group. *RISC-V IOMMU Architecture Specification*. 2023.
[90] Manuel Rodríguez, Francisco Costa, Bruno Vilaça Sá, and Sandro Pinto. ``Open-source RISC-V Input/Output Memory Management Unit (IOMMU) IP''. In: (2023).
[91] Arm Ltd. *Armv8-R virtualization*. 2022.
[92] Dong Du, RISC-V SPMP Task Group. *RISC-V S-mode Physical Memory Protection (SPMP)*. 2023.
[93] Luís Cunha. *Open-source RISC-V Input/Output Physical Memory Protection (IOPMP) IP*.
[94] Francesco Paci, Davide Brunelli, and Luca Benini. ``Lightweight IO virtualization on MPU enabled microcontrollers''. In: *ACM SIGBED Review* (2018).
[95] Felix Bruns, Dirk Kuschnerus, and Attila Bilgic. ``Virtualization for safety-critical, deeply-embedded devices''. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013.

[96] Sanndro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. ``Virtualization on trustzone-enabled microcontrollers? voilà!'' In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019.
[97] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. ``Predictable virtualization on memory protection unit-based microcontrollers''. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2018.
[98] Daniel Gruss. ``Software-based microarchitectural attacks''. In: *it-Information Technology* (2018).
[99] Jakub Szefer. ``Survey of microarchitectural side and covert channels, attacks, and defenses''. In: *Journal of Hardware and Systems Security* (2019).
[100] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. ``A survey of microarchitectural timing attacks and countermeasures on contemporary hardware''. In: *Journal of Cryptographic Engineering* (2018).
[101] Chao Su and Qingkai Zeng. ``Survey of CPU cache-based side-channel attacks: systematic analysis, security models, and countermeasures''. In: *Security and Communication Networks* (2021).
[102] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. ``Spectre attacks: Exploiting speculative execution''. In: *Communications of the ACM* (2020).
[103] Nael Abu-Ghazaleh, Dmitry Ponomarev, and Dmitry Evtyushkin. ``How the spectre and meltdown hacks really worked''. In: *IEEE Spectrum* (2019).
[104] Abel Gordon et al. ``ELI: Bare-Metal Performance for I/O Virtualization''. In: *SIGPLAN Notices* (2012).
[105] Giovani Gracioli et al. ``A Survey on Cache Management Mechanisms for Real-Time Embedded Systems''. In: *ACM Computing Surveys* (2015).
[106] Arm. *Software Delegated Exception Interface (SDEI)*. https : / / developer . arm . com / documentation/den0054/latest. 2021.
[107] R. Ramsauer et al. ``Look Mum, no VM Exits!(Almost)''. In: *Proc. of Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. 2017.
[108] Ralf Ramsauer et al. ``A Novel Software Architecture for Mixed Criticality Systems''. In: *Digital Transformation in Semiconductor Manufacturing*. 2020.
[109] Ralf Ramsauer et al. ``Static Hardware Partitioning on RISC-V -- Shortcomings, Limitations, and Prospects''. In: *Proc. of IEEE World Forum on Internet of Things*. 2022.
[110] T. Kloda et al. ``Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems''. In: *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019.
[111] Parul Sohal et al. ``E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management''. In: *Proc. of Real-Time Systems Symposium (RTSS)*. 2020.
[112] *jailhouse-RT: Bu-maintained version of the jailhouse partitioning hypervisor with real-time features*.
[113] Alessandro Biondi et al. ``SPHERE: A Multi-SoC Architecture for Next-Generation Cyber-Physical Systems Based on Heterogeneous Platforms''. In: *IEEE Access* (2021).
[114] J. Hwang et al. ``Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones''. In: *Proc. of Consumer Communications and Networking Conference*. 2008.
[115] Giulio Corradi. ``Xen on Arm: Real-Time Virtualization with Cache Coloring''. In: *Proc. of Embedded World Conference*. 2020.
[116] *Zephyr project*. 2023.
[117] Jose Martins et al. ``Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems''. In: *Proc. of Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*. 2020.
[118] Bruno Sa et al. ``A First Look at RISC-V Virtualization from an Embedded Systems Perspective''. In: *IEEE Transactions on Computers* (2021).

[119] Gerwin Klein et al. ``SeL4: Formal Verification of an OS Kernel''. In: *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*. 2009.
[120] Gernot Heiser. *The seL4 Microkernel: An Introduction*. The seL4 Foundation. 2020.
[121] Gerwin Klein et al. ``Formally Verified Software in the Real World''. In: *Communications of the ACM* (2018).
[122] Jesse Millwood et al. ``Performance Impacts from the seL4 Hypervisor''. In: *Proc. of the Ground Vehicle Systems Engineering and Technology Symposium*. 2020.
[123] Anna Lyons et al. ``Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time''. In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2018.
[124] Qian Ge et al. ``Time Protection: The Missing OS Abstraction''. In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2019.
[125] Toby Murray et al. ``seL4: From General Purpose to a Proof of Information Flow Enforcement''. In: *Proc. of IEEE Symposium on Security and Privacy (S&P)*. 2013.
[126] Gerwin Klein et al. ``Comprehensive Formal Verification of an OS Microkernel''. In: *ACM Transactions on Computer Systems* (2014).
[127] Gernot Heiser et al. ``Towards Provable Timing-Channel Prevention''. In: *ACM SIGOPS Operating Systems Review* (2020).
[128] Gernot Heiser et al. ``Can We Put the "S" Into IoT?'' In: *Proc. of IEEE World Forum on Internet of Things*. 2022.
[129] José Martins and Sandro Pinto. ``Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems''. In: *Proc. of RTAS*. 2023.
[130] Alvin Che-Chia Chang, Andy Chiu, Andy Dellow, Anup Patel, Dean Liberty, Deepak Gupta, Eckhard Delfs, Gregor Haas, Guerney Hunt, Jack Andrew, Krste Asanovic, Mark Hill, Nick Wood, Osman Koyuncu, Paul Elliott, Ricardo Ramirez, Ravi Sahita (Editor), Samuel Holland, Samuel Ortiz, Ved Shanbhogue, Wojchiech Ozga. *Supervisor Domain Access Protection - RISC-V privileged architecture extension*. [online] Available at: https://github.com/riscv/riscv-smmtt. 2025.
[131] Thomas Aird, Hesham Almatary, Andres Amaya Garcia, John Baldwin, Paul Buxton, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Franz A. Fuchs, Timothy Hutt, Alexandre Joannou, Martin Kaiser, Tariq Kurd, Ben Laurie, Marno van der Maas, Maja Malenko, A. Theodore Markettos, David McKay, Jamie Melling, Stuart Menefy, Simon W. Moore, Peter G. Neumann, Robert Norton, Alexander Richardson, Michael Roe, Peter Rugg, Peter Sewell, Carl Shaw, Ricki Tura, Robert N. M. Watson, Toby Wenman, Jonathan Woodruff, Jason Zhijingcheng Yu. *RISC-V Specification for CHERI Extensions*. [online] Available at: https://github.com/riscv/riscv-cheri. 2025.
[132] An Braeken. ``PUF based authentication protocol for IoT''. In: *Symmetry* (2018).
[133] Wenjie Che, Fareena Saqib, and Jim Plusquellic. ``PUF-based authentication''. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015.
[134] Ünal Kocabaş, Andreas Peter, Stefan Katzenbeisser, and Ahmad-Reza Sadeghi. ``Converse PUF-based authentication''. In: *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*. 2012.
[135] Seungyong Yoon, Byoungkoo Kim, Yousung Kang, and Dooho Choi. ``Puf-based authentication scheme for iot devices''. In: *2020 international conference on information and communication technology convergence (ICTC)*. 2020.
[136] G Edward Suh and Srinivas Devadas. ``Physical unclonable functions for device authentication and secret key generation''. In: *Proceedings of the 44th annual design automation conference*. 2007.
[137] Anthony Van Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. ``Reverse fuzzy extractors: Enabling lightweight mutual authentication for PUF-enabled RFIDs''. In: *Financial Cryptography and Data Security:*

16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers 16. 2012.

[138] Donglan Liu, Xin Liu, Hao Zhang, Hao Yu, Wenting Wang, Lei Ma, Jianfei Chen, and Dong Li. ``Research on end-to-end security authentication protocol of NB-IoT for smart grid based on physical unclonable function''. In: *2019 IEEE 11th International Conference on Communication Software and Networks (ICCSN)*. 2019.

[139] Urbi Chatterjee, Rajat Sadhukhan, Vidya Govindan, Debdeep Mukhopadhyay, Rajat Subhra Chakraborty, Sweta Pati, Debashis Mahata, and Mukesh M Prabhu. ``PUFSSL: an OpenSSL extension for PUF based authentication''. In: *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*. 2018.

[140] Jin Wook Byun. ``An efficient multi-factor authenticated key exchange with physically unclonable function''. In: *2019 International Conference on Electronics, Information, and Communication (ICEIC)*. 2019.

[141] SD Suganthi, RVSS Anitha, Venkatasamy Sureshkumar, S Harish, and S Agalya. ``End to end light weight mutual authentication scheme in IoT-based healthcare environment''. In: *Journal of Reliable Intelligent Environments* (2020).

[142] Raffaele Pugliese, Stefano Regondi, and Riccardo Marini. ``Machine learning-based approach: Global trends, research directions, and regulatory standpoints''. In: *Data Science and Management* (2021).

[143] Wei Liang, Songyou Xie, Dafang Zhang, Xiong Li, and Kuan-ching Li. ``A mutual security authentication method for RFID-PUF circuit based on deep learning''. In: *ACM Transactions on Internet Technology (TOIT)* (2021).

[144] Vlastimil Clupek and Vaclav Zeman. ``Robust mutual authentication and secure transmission of information on low-cost devices using physical unclonable functions and hash functions''. In: *2016 39th International Conference on Telecommunications and Signal Processing (TSP)*. 2016.

[145] Yun-Hsin Chuang and Chin-Laung Lei. ``PUF based authenticated key exchange protocol for IoT without verifiers and explicit CRPs''. In: *IEEE Access* (2021).

[146] Mario Barbareschi, Alessandra De Benedictis, Erasmo La Montagna, Antonino Mazzeo, and Nicola Mazzocca. ``PUF-enabled authentication-as-a-service in fog-IoT systems''. In: *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2019.

[147] Zhao Huang and Quan Wang. ``A PUF-based unified identity verification framework for secure IoT hardware via device authentication''. In: *World Wide Web* (2020).

[148] Karim Lounis and Mohammad Zulkernine. ``T2T-MAP: A PUF-based thing-to-thing mutual authentication protocol for IoT''. In: *IEEE Access* (2021).

[149] Urbi Chatterjee, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. ``A PUF-based secure communication protocol for IoT''. In: *ACM Transactions on Embedded Computing Systems (TECS)* (2017).

[150] Gurjot Singh Gaba, Mustapha Hedabou, Pardeep Kumar, An Braeken, Madhusanka Liyanage, and Mamoun Alazab. ``Zero knowledge proofs based authenticated key agreement protocol for sustainable healthcare''. In: *Sustainable Cities and Society* (2022).

[151] Muhammad Arif Muhal, Xiong Luo, Zahid Mahmood, and Ata Ullah. ``Physical unclonable function based authentication scheme for smart devices in Internet of Things''. In: *2018 IEEE International Conference on Smart Internet of Things (SmartIoT)*. 2018.

[152] JoonYoung Lee, JiHyeon Oh, DeokKyu Kwon, MyeongHyun Kim, SungJin Yu, Nam-Su Jho, and Youngho Park. ``PUFTAP-IoT: PUF-Based Three-Factor Authentication Protocol in IoT Environment Focused on Sensing Devices''. In: *Sensors* (2022).

[153] Qi Jiang, Xin Zhang, Ning Zhang, Youliang Tian, Xindi Ma, and Jianfeng Ma. ``Two-factor authentication protocol using physical unclonable function for IoV''. In: *2019 IEEE/CIC International Conference on Communications in China (ICCC)*. 2019.

[154] Muhammad Naveed Aman, Kee Chaing Chua, and Biplab Sikdar. ``Physically secure mutual authentication for IoT''. In: *2017 IEEE Conference on Dependable and Secure Computing*. 2017.
[155] Carmelo Felicetti, Marco Lanuzza, Antonino Rullo, Domenico Saccà, and Felice Crupi. ``Exploiting Silicon Fingerprint for Device Authentication Using CMOS-PUF and ECC''. In: *2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*. 2021.
[156] Pim Tuyls and Lejla Batina. ``RFID-tags for anti-counterfeiting''. In: *Cryptographers' track at the RSA conference*. 2006.
[157] Carmelo Felicetti, Antonella Guzzo, Giuseppe Manco, Francesco Pasqua, Ettore Ritacco, Antonino Rullo, and Domenico Saccà. ``Deep Learning/PUF-based Item Identification for Supply Chain Management in a Distributed Ledger Framework''. In: *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)*. 2023.
[158] Mohd Shariq, Karan Singh, Mohd Yazid Bajuri, Athanasios A Pantelous, Ali Ahmadian, and Mehdi Salimi. ``A secure and reliable RFID authentication protocol using digital schnorr cryptosystem for IoT-enabled healthcare in COVID-19 scenario''. In: *Sustainable Cities and Society* (2021).
[159] Mahshid Delavar, Sattar Mirzakuchaki, Mohammad Hassan Ameri, and Javad Mohajeri. ``PUF-based solutions for secure communications in Advanced Metering Infrastructure (AMI)''. In: *International Journal of Communication Systems* (2017).
[160] Nathan Beckmann and Miodrag Potkonjak. ``Hardware-based public-key cryptography with public physically unclonable functions''. In: *Information Hiding: 11th International Workshop, IH 2009, Darmstadt, Germany, June 8-10, 2009, Revised Selected Papers 11*. 2009.
[161] Hala Hamadeh and Akhilesh Tyagi. ``Privacy preserving data provenance model based on PUF for secure Internet of Things''. In: *2019 IEEE International Symposium on Smart Electronic Systems (iSES)(Formerly iNiS)*. 2019.
[162] Torben Pryds Pedersen. ``Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing''. In: *Advances in Cryptology --- CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140. ISBN: 978-3-540-46766-3.
[163] Feng Hao. *Schnorr non-interactive zero-knowledge proof*. Tech. rep. 2017.
[164] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.
[165] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2014. URL: https://ethereum.org/en/whitepaper/.
[166] Vikram Dhillon, David Metcalf, Max Hooper, Vikram Dhillon, David Metcalf, and Max Hooper. ``The Hyperledger Project''. In: *Blockchain Enabled Applications* (2017).
[167] Juan Benet. ``IPFS - Content Addressed, Versioned, P2P File System''. In: *arXiv preprint arXiv:1407.3561* (2014).
[168] Lukas Petzi, Torsten Krauß, Alexandra Dmitrienko, and Gene Tsudik. ``AuthentiSafe: Lightweight and Future-Proof Device-to-Device Authentication for IoT''. In: *to appear in the 20th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2025)* (2025).
[169] Jie Ding, Mahyar Nemati, Chathurika Ranaweera, and Jinho Choi. ``IoT Connectivity Technologies and Applications: A Survey''. In: *IEEE Access* (2020).
[170] Jeyanthi Hall, Michel Barbeau, and Evangelos Kranakis. ``Detection Of Transient In Radio Frequency Fingerprinting Using Signal Phase''. In: (2003).
[171] Junqing Zhang, Guanxiong Shen, Walid Saad, and Kaushik Chowdhury. ``Radio Frequency Fingerprint Identification for Device Authentication in the Internet of Things''. In: *IEEE Communications Magazine* (2023).
[172] Anu Jagannath, Jithin Jagannath, and Prem Sagar Pattanshetty Vasanth Kumar. ``A comprehensive survey on radio frequency (RF) fingerprinting: Traditional approaches, deep learning, and open challenges''. In: *Computer Networks* (2022).
[173] Benjamin W. Ramsey, Barry E. Mullins, Michael A. Temple, and Michael R. Grimaila. ``Wireless Intrusion Detection and Device Fingerprinting through Preamble Manipulation''. In: *IEEE Transactions on Dependable and Secure Computing* (2015).

[174] Jingyu Hua, Hongyi Sun, Zhenyu Shen, Zhiyun Qian, and Sheng Zhong. ``Accurate and Efficient Wireless Device Fingerprinting Using Channel State Information''. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018.

[175] Shabir Abdul Samadh, Qianyu Liu, Xue Liu, Negar Ghourchian, and Michel Allegue. ``Indoor Localization Based on Channel State Information''. In: *2019 IEEE Topical Conference on Wireless Sensors and Sensor Networks (WiSNet)*. 2019.

[176] ``IEEE Standard for Information Technology--Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks--Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications''. In: *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)* (2021).

[177] Yongsen Ma, Gang Zhou, and Shuangquan Wang. ``WiFi Sensing with Channel State Information: A Survey''. In: *ACM Comput. Surv.* (2019).

[178] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. ``Signature Verification Using a "Siamese" Time Delay Neural Network''. In: *Proceedings of the 6th International Conference on Neural Information Processing Systems*. Denver, Colorado: Morgan Kaufmann Publishers Inc., 1993.

[179] Hovav Shacham. ``The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)''. In: *Proceedings of the 14th ACM conference on Computer and communications security*. 2007.

[180] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. ``Jump-oriented programming: a new class of code-reuse attack''. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 2011.

[181] Yawei Yue, Shancang Li, Phil Legg, and Fuzhong Li. ``Deep Learning-Based Security Behaviour Analysis in IoT Environments: A Survey''. In: *Security and Communication Networks* (2021).

[182] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2020.

[183] Taiwo Samuel Ajani, Agbotiname Lucky Imoize, and Aderemi A. Atayero. ``An Overview of Machine Learning within Embedded and Mobile Devices–Optimizations and Applications''. In: *Sensors* (2021).

[184] Georgios Kornaros. ``Hardware-Assisted Machine Learning in Resource-Constrained IoT Environments for Security: Review and Future Prospective''. In: *IEEE Access* (2022).

[185] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. ``Survey of intrusion detection systems: techniques, datasets and challenges''. In: *Cybersecurity* (2019).

[186] Khaled A. Alaghbari, Heng-Siong Lim, Mohamad Hanif Md Saad, and Yik Seng Yong. ``Deep Autoencoder-Based Integrated Model for Anomaly Detection and Efficient Feature Extraction in IoT Networks''. In: *IoT* (3 2023).

[187] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. ``FLAShadow: A Flash-based Shadow Stack for Low-end Embedded Systems''. In: *ACM Trans. Internet Things* 5.3 (2024), 19:1–19:29. DOI: 10.1145/3670413.

[188] Luca Degani, Majid Salehi, Fabio Martinelli, and Bruno Crispo. ``μIPS: Software-Based Intrusion Prevention for Bare-Metal Embedded Systems''. In: *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part IV*. Ed. by Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis. Vol. 14347. Lecture Notes in Computer Science. Springer, 2023, pp. 311–331. DOI: 10.1007/978-3-031-51482-1_16. URL: https://doi.org/10.1007/978-3-031-51482-1%5C_16.

[189] Conner Bradley and David Barrera. ``Towards Characterizing IoT Software Update Practices''. In: *Foundations and Practice of Security*. Springer Nature Switzerland, 2023.

[190] Saad El Jaouhari and Eric Bouvet. ``Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions''. In: *Internet of Things* (2022).

[191] Alberto Tacchella, Emanuele Beozzo, Bruno Crispo, and Marco Roveri. ``Certified Secure Updates for IoT Devices''. In: *ICT Systems Security and Privacy Protection - 40th IFIP International Conference, SEC 2025, Maribor, Slovenia, May 21-23, 2025, Proceedings, Part I*. Springer, 2025.

[192] Alberto Tacchella, Emanuele Beozzo, Bruno Crispo, and Marco Roveri. ``Firmware Secure Updates Meet Formal Verification''. In: *ACM Transactions on Cyber-Physical Systems* (2025), p. 3754455. ISSN: 2378-962X, 2378-9638. DOI: 10.1145/3754455. URL: https://dl.acm.org/doi/10.1145/3754455.

[193] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. *A Firmware Update Architecture for Internet of Things*. RFC 9019. 2021.

[194] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. *A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices*. RFC 9124. 2022.

[195] Brendan Moran, Hannes Tschofenig, Henk Birkholz, Koen Zandberg, and Øyvind Rønningstad. *A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest*. Internet-Draft. Work in Progress. Internet Engineering Task Force, 2024. 101 pp.

[196] George C. Necula and Peter Lee. ``The Design and Implementation of a Certifying Compiler''. In: *SIGPLAN Not.* (1998).

[197] Haniel Barbosa, Clark W. Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. ``Generating and Exploiting Automated Reasoning Proof Certificates''. In: *Commun. ACM* (2023).

[198] Alberto Tacchella. ``An approach to memory safety through secure compilation''. Unpublished manuscript available upon request to the author. 2025.

[199] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. ``Achieving safety incrementally with Checked C''. In: *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Flemming Nielson and David Sands. Vol. 11426. Lecture Notes in Computer Science. Springer, 2019, pp. 76–98. DOI: 10.1007/978-3-030-17138-4_4. URL: https://doi.org/10.1007/978-3-030-17138-4%5C_4 (visited on 05/13/2024).

[200] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. ``MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code''. In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 425–454. DOI: 10.1145/3571208.

[201] Sandro Pinto, Luis Cunha, Daniel Oliveira, Michele Grisafi, Emanuele Beozzo, and Bruno Crispo. *Bridging the interoperability gaps among trusted architectures in MCUs*. Accepted at the ICICS 2025 conference. 2025.

[202] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. ``PISTIS: Trusted Computing Architecture for Low-end Embedded Systems''. In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 2022.