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

D3.1 CROSSCON Open Security Stack Documentation - Draft

Document Identification			
Status	Final	**Due Date**	30/04/2024
Version	1.0	**Submission Date**	30/04/2024

Related WP	WP3	**Document Reference**	D3.1
Related Deliverable(s)	D1.4, D1.5, D2.1 D3.2	**Dissemination Level(*)**	PU
Lead Participant	UMINHO	**Lead Author**	Sandro Pinto (UMINHO)
Contributors	UMINHO, UNITN, UWU, TUD	**Reviewers**	UNITN, UWU

Keywords

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

(*) 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**
Sandro Pinto	UMINHO
David Cerdeira	UMINHO
João Sousa	UMINHO
Luís Cunha	UMINHO
Bruno Crispo	UNITN
Michele Grisafi	UNITN
Marco Roveri	UNITN
Alberto Tacchella	UNITN
Tommaso Zoppi	UNITN
Lukas Petzi	UWU
Peter Ten	UWU
Hristo Koshutanski	ATOS
Shaza Zeitouni	TUD

Document history			
Ver.	**Date**	**Change editors**	**Changes**
0.1	08/01/2024	Sandro Pinto (UMINHO), João Sousa (UMINHO), David Cerdeira (UMINHO)	First draft - Initial Contributions;
0.2	28/02/2024	Sandro Pinto (UMINHO), João Sousa (UMINHO), David Cerdeira (UMINHO), Peter Ten (UWU), Lukas Petzi (UWU), Akos Milánkovich (SLAB), Bruno Crispo (UNITN), Michele Grisafi (UNITN), Alberto Tacchella (UNITN), ommaso Zoppi (UNITN), Shaza Zeitouni (TUD), Hristo Koshutanski (ATOS)	Second draft ready consortium early feedback;
0.3	09/04/2024	Sandro Pinto (UMINHO), João Sousa (UMINHO), Luís Cunha (UMINHO), David Cerdeira (UMINHO), Peter Ten (UWU), Lukas Petzi (UWU), Akos Milánkovich (SLAB), Bruno Crispo (UNITN), Michele Grisafi (UNITN), Alberto Tacchella (UNITN), ommaso Zoppi (UNITN), Shaza Zeitouni (TUD), Hristo Koshutanski (ATOS)	Deliverable ready for review;
0.4	26/04/2024	Sandro Pinto (UMINHO), João Sousa (UMINHO), David Cerdeira (UMINHO), Luís Cunha (UMINHO)	Document After Review of UWU and UNITN;
0.9	29/04/2024	Juan Alonso (ATOS)	QA Review;
1.0	30/04/2024	H. Koshutanski (ATOS)	Final version submitted;

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Sandro Pinto (UMINHO)	29/04/2024
Quality Manager	Juan Alonso (ATOS)	30/04/2024
Project Coordinator	H. Koshutanski (ATOS)	30/04/2024

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation / acronym	Description
ABI	Application Binary Interface
AES	Advanced Encryption Standard
AIA	Advanced Interrupt Architecture
API	Application Programming Interface
APU	Application Processing Unit
ASLR	Address Space Layout Randomization
BOM	Bill of Materials
CA	Client Application
CCA	Confidential Computing Architecture
CI/CD	Continuous Integration / Continuous Development
CoVE	Confidential VM Extension
CPU	Central Processing Unit
CSI	Channel State Information
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DAST	Dynamic Application Security Testing
DMA	Direct Memory Access
DRM	Digital rights management
DRTM	Dynamic RTM
FOSE	Firmware Object Signing and Encryption
FOTA	Firmware OTA
FPGA	Field-Programmable Gate Array
GIC	General Interrupt Controller
GPOS	General Purpose OS
GPU	Graphic Processing Unit
HAL	Hardware Abstraction Layer
HBOM	Hardware BOM
I/O	Input/Output
IAST	Interactive Application Security Testing
IDAU	Implementation-Defined Attribution Unit
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
IPI	Inter-Processor Interrupt
ISA	Instruction Set Architecture
JOSE	JSON Object Signing and Encryption
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
MTT	Memory Tracking Tables
OS	Operating Systems
OTA	Over-the-Air
OTP	One-Time-Programmable
PLIC	Platform Local Interrupt Controller
PMP	Physical Memory Protection
PMU	Performance Monitor Unit
PUF	Physical Unclonable Function
RDC	Resource Domain Controller
REE	Rich Execution Environment
RFF	Radio Frequency Fingerprinting
RFFI	RFF Identification
RNG	Random Number Generator
RSA	Rivest-Shamir-Adleman
RTM	Root-of-Trust Measurement
RTOS	Real-Time OS
RTU	Real-Time Processing Unit
SAST	Static Application Security Testing
SAU	Security Attribution Unit
SBOM	Software BOM
SCA	Software Composition Analysis
SCUBA	Secure Code Update By Attestation
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
SMMU	System Memory-Management Unit
SoC	System on Chip
SPDX	Software Package Data eXchange
SPH	Static Partitioning Hypervisor
sPMP	supervisor Physical Memory Protection
SRTM	Static RTM
SUIT	Software Updates for Internet of Things
SWID	Software Identification
TA	Trusted Application
TCB	Trusted Computing Base
TDX	Trust Domain Extensions
TEE	Trusted Execution Environment
TinyML	Tiny Machine Learning
TPM	Trusted Platform Module
TUF	The Update Framework
UC	Use Case

vFPGA	virtual FPGA
VM	Virtual Machine
VMM	VM Monitor
Wi-Fi	Wireless Fidelity
WP	Work Package

Executive Summary

The Internet of Things (IoT) landscape is highly heterogeneous, consisting of devices with low-power Microcontroller Unit (MCU) with limited security capabilities, all the way to devices boasting advanced Application Processing Unit (APU) with multiple cores leveraging reconfigurable hardware. Moreover, a typical IoT stack encompasses various layers, including hardware, firmware, and Operating Systems (OS), each layer contributing to the system's complexity and increasing the potential attack surface. This heterogeneity introduces numerous challenges, including (i) the lack of interoperability and isolation across different Trusted Execution Environment (TEE) systems, (ii) providing the required dynamicity and per-Virtual Machine (VM) services without enlarging the Trusted Computing Base (TCB) and lowering the isolation guarantees, (iii) the lack of novel trusted services, (iv) firmware updates and secure cross-compilation, and (v) the of lack of security features on bare-metal devices. The CROSSCON Stack, as the core of the Work Package (WP) 3, aims to address these challenges by providing a highly portable IoT security stack that can operate on various edge devices and multiple hardware platforms. This work can be divided into five main areas: (i) TEE isolation and abstraction by decomposing trusted services from trusted kernel, (ii) CROSSCON Hypervisor with dynamic VM and per-VM TEE features, (iii) new Trusted Services such as Physical Unclonable Function (PUF)-based authentication, context-based authentication, (iv) CROSSCON TEE Toolchain, which manages firmware updates and a secure compilation process, and (v) CROSSCON Bare-metal TEE which provides TEE security guarantees to resource constrained devices. This document presents the initial results of our work based on WP2 specifications, including initial analysis, design, and plan for future work.

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, proposing 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. To select the hypervisor to be used as the foundation for the CROSSCON Hypervisor, we compared the most prominent open-source static partitioning hypervisors. This document also emphasizes the significance of creating innovative trusted services as complementary to existing ones, while describing the proposed trusted services' functionality, architecture, and implementation. Furthermore, it examines the current update mechanisms utilized in IoT and conducts a literature review on secure compilation proposals, before establishing the CROSSCON Toolchain. Finally, it discusses the existing solutions for implementing TEEs on low-end devices and delineates the design of the proposed Baremetal TEEs.

1.2 Relation to Other Project Work

This document takes as direct input the initial version of the CROSSCON open specifications in deliverable D2.1, and requirements elicitation final technical specification D1.5. Particularly relevant are the draft specifications laid out in document D2.1 and the device classification and requirements established in deliverable D1.5. Deliverable D3.2 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.

The outcomes presented in this document will serve as input for subsequent documents, notably for the enhancement of the CROSSCON Stack specification in D2.3. This will involve incorporating new insights into the system architecture and component interfaces. Furthermore, the components presented in this document will be subject to validation tests which are specified in deliverable D1.6. Additionally, these results will contribute to the development of the initial version of the integrated CROSSCON security stack in D5.2. 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.

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 D2.1. 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., Memory Protection Unit (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 functions (e.g., secure storage, on-time programmable memories) and 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 components (e.g., Random Number Generator (RNG), PUF, HW-based intrusion detection) and 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 a low core count 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 (D2.1): The CROSSCON stack design intends to cover multiple architectures and vendors in a wide range of implementation scenarios. This heterogeneity has materialized in several CROSSCON instantiation options, each conceptually corresponding to a different

CROSSCON stack configuration deployment. Given the functionalities provided by the CROSSCON stack components and the security capabilities featured by the selected platforms (e.g., virtualization, TEE, secure boot, RNG, or PUF), seven CROSSCON instantiation options emerged. Considered architectures include: ArmV7-A, ArmV8-A (<V8.4), RISC-V for APUs and TI MSP, AVR, ArmV6-M, ArmV7-M, ArmV8-M, Armv8-R and RISC-V for MCUs.

We redesigned the CROSSCON instantiation options from the initial options presented in D2.1. In the new version of the instantiation options, we only consider virtualization support in platforms featuring virtualization hardware capabilities, such as the second-stage MMU. Additionally, we divided the instantiation options previously categorized as *TEE- and Virtualization-less environment* into two distinct options: the *SW-only isolation environment* and the *Basic memory isolation environment*. This separation was necessary to distinguish between platforms lacking any security capabilities and those with only basic security capabilities.

To map the selected platforms into the instantiation options, we considered all CROSSCON components (e.g., CROSSCON Hypervisor, Trusted Application (TA), Trusted OSes) and the capabilities provided by the selected platforms that can support them. The operation of CROSSCON components may depend on the hardware they are running on. For example, a platform without specialized virtualization hardware (e.g., 2nd stage MMU) does not contain capabilities to run hypervisor functionalities, but could feature TEE support. Considering this, CROSSCON instantiation options encompass the following scenarios:

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 a software-based methodology to ensure isolation between normal applications and trusted applications.

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 trusted applications.

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 component operates above the firmware, 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 features TEE support but does not include a hypervisor. However, different architectures may adopt different TEE technologies. CROSSCON addresses this by supporting varied TEE implementations across multiple platforms, including Arm, RISC-V, and potentially others.

Environment with TEE and Virtualization (v): This CROSSCON instantiation option combines the flexibility the hypervisor provides with TEE security guarantees. This option encompasses all isolation capabilities.

Environment with Virtualization, TEE, and Field-Programmable Gate Array (FPGA) (vi): A platform could contain accelerators deployed in FPGA fabric. Through this instantiation option, CROSSCON demonstrates an awareness of the interface with these components, and carefully considers the security implications of FPGA-enabled devices.

In addition to these CROSSCON instantiation options, in some contexts (e.g., when the platform lacks dedicated hardware for running multiple trusted components), TEE components could be moved to the Rich Execution Environment (REE) by leveraging virtualization (i.e., using a novel per-VM feature of CROSSCON Hypervisor described in Section 3.2.5.2) or TEE dedicated hardware (e.g., using Trustzone-M).

2.1 Platform Analysis and Selection

The specification of the CROSSCON stack is general and does not depend on any specific manufacturer or device. However, its implementation interacts directly with the hardware of the target device. Therefore, it was important to select instances of representative platforms on which the CROSSCON stack is going to be implemented.

Considering CROSSCON's device classification, CROSSCON instantiation options, and selected UCs, we selected a set of platforms to cover at least one platform per class and one platform per CROSSCON instantiation option and architecture (RISC-V or Arm architectures). The selected platforms are MSP430F5529LP, NUCLEO-G0B1RE, NUCLEO-H743ZI2, nRF52840 Dongle, ESP32-C3-AWS-ExpressLink-DevKit, NXP LPC55S6x, Arty-100T (BA5x), Beagle-V, Raspberry PI 4B, Beagle-bone AI-64, ZCU 102 and Genesys 2.

Table 1 summarizes the security features of the selected platforms. These features include the presence of TEE technology, memory and I/O isolation, cryptography accelerators, On-The-Fly Encryption/Decryption (OTF), RNG, PUF, Universally Unique Identifier (UUID), Secure Element, Secure Boot and One-Time-Programmable (OTP) Memories. Among these platform capabilities, we also distinguish them regarding their target applications, i.e., APU, MCU, or FPGA.

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	-	-	-	-	-	-	✓	✓
	nRF52840 Dongle	-	-	MPU	✓	-	✓	-	-	-	-	-
	ESP32-C3-AWS-ExpressLink-DevKit	-	-	PMP	✓	✓	✓	-	-	-	✓	✓
	NXP LPC55S6x (LPC55S6x)	TF-M	✓	MPU	✓	✓	✓	✓	✓	✓	✓	✓
	Arty-100T (BA5x)	PMP	-	-	-	-	-	-	-	-	-	-
APU	**Beagle-V**	PMP	-	-	-	-	✓	-	-	-	-	✓
	Raspberry PI 4B	-	-	MMU(2nstage)	-	-	✓	-	-	-	✓	✓
	Beaglebone AI-64	TZ-A	✓	MMU(2nstage)	✓	-	✓	-	-	-	✓	✓
FPGA	**Genesys 2**	PMP	-	-	-	-	-	-	-	-	-	✓
	ZCU 102	TZ-A	✓	MMU(2nstage)	✓	-	-	✓	-	✓	✓	✓

With a clear understanding of the security capabilities of each platform, we can carry out the classification according to their Class (0---3) and corresponding 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; the *NUCLEO-H743ZI2*, equipped with MPU, RNG, and PUF; and the *nRF52840 Dongle*, providing MPU, RNG, and cryptography accelerators. 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.

We now match the platforms with the respective instantiation options, considering all CROSSCON components (e.g., CROSSCON Hypervisor, Trusted Application (TA)) and the platform's security capabilities.

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 chosen for instantiation option ***ii*** include the *NUCLEO-G0B1RE*, *NUCLEO-H743ZI2*, *nRF52840* Dongle, *ESP32-C3-AWS-ExpressLink-DevKit*, and the *Beagle-V*. It's worth mentioning that despite the *Beagle-V* featuring RISC-V architecture with a three-privileged-level system (M, S, and U-mode) and employing memory protection using Physical Memory Protection (PMP) and MMU, it is still considered constrained in terms of security support, hence its inclusion in this option.

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, TEE support, 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 levels, UC provider partners selected the platforms that best match their UC: The *nRF52840* Dongle and *NXP LPC55S6* are 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.

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	UNIT	*(i)*	
	NUCLEO-G0B1RE	1	Armv6-M	UNIT	*(ii)*	
	NUCLEO-H743ZI2	1	Armv7-M	UNIT / UM	*(ii)*	
	nRF52840 Dongle	1	Armv7-M	3MDEB	*(ii)*	UC1
	ESP32-C3-AWS -ExpressLink-DevKit	2	RISC-V	UNIT / 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 w/ H + IOMMU)	3	RISC-V	UM	*(v), (vi)*	
FPGA	ZCU 102	3	FPGA	TUD / UM	*(vi)*	UC5
	Genesys 2	3	FPGA	BEYOND	*(vi)*	

Summary. 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.

3 Research Results

This section outlines the research and innovation efforts involved in designing and implementing the components for an initial validation of the research of the CROSSCON stack. The section begins focusing on TEE Isolation and Abstraction, covering TEE technologies, TEE vulnerabilities, TEE isolation, and TEE abstraction discussions. Following this, it focuses on the CROSSCON Hypervisor, examining virtualization and microarchitecture isolation techniques, hypervisor feature analysis, and selection of the hypervisor that serves as the starting point for CROSSCON Hypervisor development. Here the development progress of two features of the CROSSCON Hypervisor is reported: dynamic VM creation and per VM TEE service support. Next, this document discusses the development of new trusted services: PUF-based authentication, remote attestation, FPGA-related services, behavioral-based services, and control flow integrity services. After, it delves into the CROSSCON TEE Toolchain, detailing existing IoT update mechanisms and standards, and requirements for integration with Dev-Sec-Ops platforms. It also provides a literature review on secure compilation and the design of CROSSCON Secure Update. Lastly, the document details the CROSSCON Bare-Metal TEE, reviewing requirements and platforms, state-of-the-art approaches, and the implementation of Bare-Metal TEE, including both MPU and non-MPU variants.

3.1 TEE Isolation and Abstraction

This section concentrates on two key aspects of integrating CROSSCON with TEEs: TEE Isolation and Abstraction.

Regarding TEE isolation, in a previous work, we've demonstrated that TEE systems, TrustZone TEE system specifically, suffered from more than 200 TEE vulnerabilities in TEE systems from 2013 to 2018 [1]. Because new TEE vulnerabilities are constantly being discovered, we explore if existing TEEs continue to suffer from the same security issues. To do this we once again investigate the root causes of the security vulnerabilities affecting TEEs, analyzing TEE vulnerabilities from 2019 until the current date. The insights obtained from this study help to achieve the goal of developing additional isolation capabilities to decompose a single TEE domain into multiple TEE domains, an approach that has been shown to mitigate the impact of TEE vulnerabilities [2, 3].

Regarding TEE abstraction, the main concern is to offer a solution that ensures the interoperability of trusted services between TEEs. Initially, our goal was to map TA-level interfaces between different TEEs; however, given the widespread adoption of Global Platform TEE specifications [4], interoperability is not a significant concern at the TA Application Programming Interface (API) level. Consequently, we have redirected our efforts, in two directions. First, we aim to identify potentially useful APIs not covered by the Global Platform specification, which could benefit the development of trusted services within CROSSCON. Second, we aim to reduce interoperability issues between TEE technologies. Because of 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 prevents software reuse between TEE technologies, especially for legacy code.

3.1.1 TEEs and TEE Technologies

As a definition, a TEE is a secure area within a computing device, typically a processor or a separate chip, where sensitive operations can be performed securely. Widely utilized across various computing spectrums, from mobile to server platforms, TEEs are designed to protect sensitive operations against unauthorized access or modification. These systems operate alongside a REE, which typically hosts a general-purpose OS utilizing a platform's hardware capabilities to implement rich functionalities. On the other hand, through dedicated hardware, TEEs support the execution of security-sensitive applications within isolated secure domains (i.e, the TA), ensuring separation from components executing in the REE. Their utilization extends to various applications such as mobile banking, Digital rights management (DRM), and secure key storage. Figure 1 presents an overview of a system incorporating a TEE, showcasing dedicated hardware components (e.g., trusted cores, trusted RAM, trusted ROM, etc) and trusted areas within shared memory and storage.

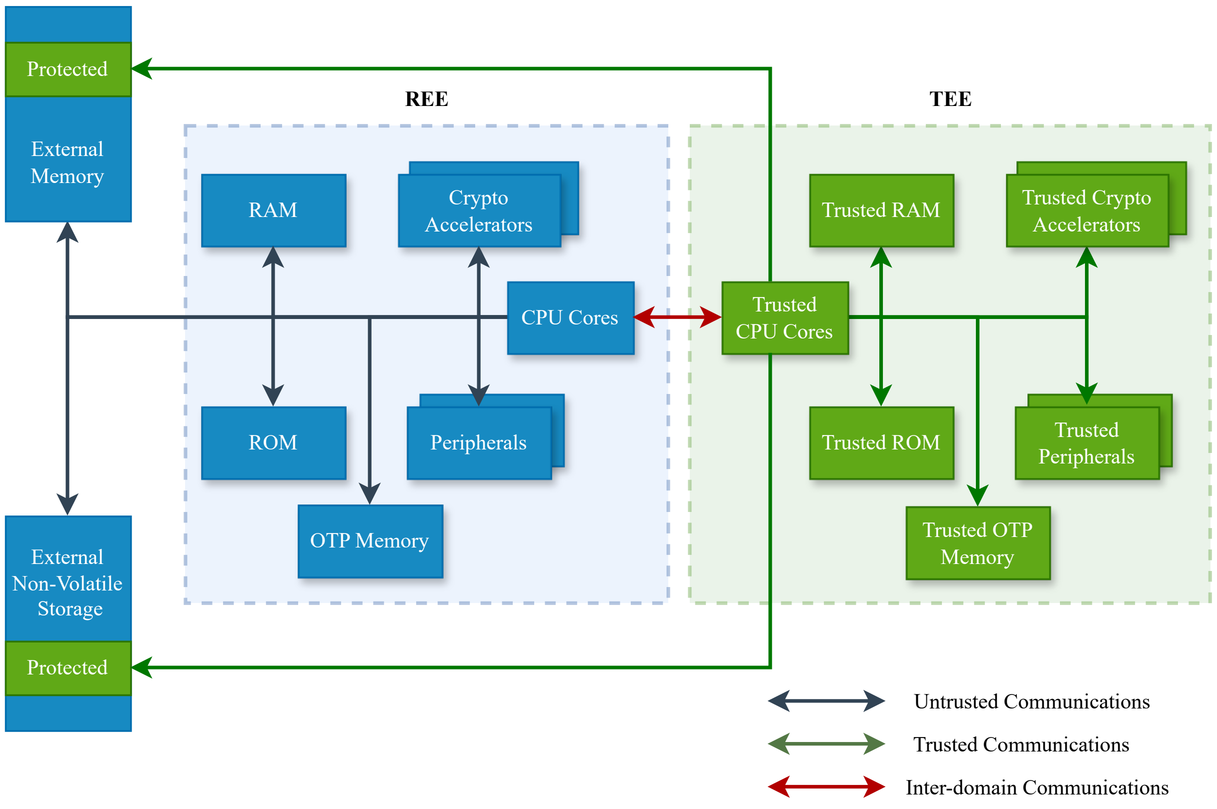


Figure 1: General representation of TEE components and their interactions.

According to 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.

In existing literature, several works survey the numerous commercially available TEE solutions [6, 1], aiming to find the design decision similarities and systematize them in terms of overall security goals. In the following sections we follow a TEE analysis that systematizes TEEs according to their architectural support of four security properties: secure boot, run-time isolation, trusted Input/Output (I/O), and secure storage.

3.1.1.1 TEE Secure Boot

Secure boot ensures that the execution environment will be configured correctly and the initial state of a TA will act as expected. It refers to proving the correctness of the initial state of the TEE. The verifiable launch process involves three main processes, (i) measurement, (ii) attestation, and (optionally) (iii) secure storage.

Measurement: It is the first process before an enclave TA executes. Typically, it is assumed to be a fingerprint of TA 's initial state. Involving two steps: (i) mapping the binary executable (of the component to be measured) into memory, and (ii) computing cryptographic hashes over it. The measurement process begins at the Root-of-Trust Measurement (RTM) and goes through a chain-of-trust measurement until measuring the TA itself. There is Static RTM (SRTM) and Dynamic RTM (DRTM). SRTM is performed at system reset before untrusted components have been started, until the enclave execution itself. It could be implemented entirely in hardware or in immutable software (including all necessary components to boot the enclave securely). On the other hand, DRTM can be performed after untrusted code initialization. Since some untrusted components could be active prior to this process, DRTM must detect the presence of adversaries present in already active untrusted components. After finishing the measurement process, the measurement report needs to be stored and signed before being sent to the verifier for the attestation process. The TEE uses the measurement as part of the attestation process to prove its authenticity and integrity to a remote entity.

Attestation: It is the process of proving the system's identity and configuration to another party. It involves a verifier to check that the system has been launched correctly and to ensure the execution of an expected initial state. In the case of a TEE, the primary goal of attestation is to establish trust in the TEE by providing evidence to a verifying party, either locally or remotely, to demonstrate the trustworthiness and integrity of the trusted entity. Local attestation is applicable when a verifier is co-located on the same platform. In contrast, remote attestation is meant for use by a remote verifier that is not on the same platform as the TEE being attested.

Secure storage: Provisioning secrets into enclaves is often the last optional step during its launch. Secrets can include sensitive data, encryption keys, authentication credentials, or any other information that needs to be protected and securely accessed by the enclave.

3.1.1.2 TEE Run-Time Isolation

TEE Run-Time Isolation ensures the protection of critical resources like the CPU and memory from potential threats, using protection mechanisms. This ensures the secure operation of TEE components, preventing unauthorized access to security-critical data.

Protection mechanisms aim to achieve isolation of TEE resources in three main ways, i.e., spatial partitioning, temporal partitioning, and spatio-temporal partitioning.

Spatial isolation involves the separation of memory, resources, or execution environments, 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 isolation ensures that the memory and resources allocated to one TA are inaccessible to another, enhancing the security and confidentiality of the TA .

Temporal isolation is the separation of execution timelines between different components of the system stack. This ensures that different components can share resources during their execution without interference. For example, if multiple applications run within the TEE, temporal isolation ensures that the execution of one application does not impact the execution of another.

Lastly, spatio-temporal isolation refers to a mix of both principles (temporal and spatial isolation), preventing interference not only across different spatial domains (e.g., memory spaces) but also across different temporal domains (e.g., time periods). For example, this approach is particularly relevant in cloud computing environments where multiple entities share the same underlying TEE infrastructure while requiring strong isolation and controlled access to resources.

Apart from resource partitioning, TEEs must also follow logical and cryptographic isolation.

Logical isolation refers to the mechanisms used to prevent the TEE adversaries from gaining access to protected data by intercepting data accesses. Since logical isolation involves the management of secure-sensitive data, this process is managed by a trusted component in the security system stack (e.g., the firmware, or hypervisors), which could modify permissions in run time by re-allocating resources through system security primitives.

On the other hand, cryptographic isolation refers to the cryptography used for only allowing authorized entities to access/decrypt the correct content. Unlike logical isolation, where protected data is entirely inaccessible to unauthorized parties (ensuring integrity), in cryptographic isolation, unauthorized entities may be able to read the ciphertext but are unable to retrieve the plaintext (ensuring confidentiality). Typically, TEEs include a cryptographic engine, which can be used to enforce this type of isolation.

Another crucial aspect in run-time isolation of TEEs is in process of context-switch between trusted and non-trusted computing, i.e., transitions from TEE to REE and vice-versa. 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.

3.1.1.3 TEE Trusted I/O

Establishing Trusted I/O in TEEs is critical for safeguarding the integrity and confidentiality of TAs when utilizing peripherals, e.g., FPGA deployed accelerator. 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 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 for authentication and attestation. Some TEE implementations utilize a combination of cryptography and logic isolation for robust MMIO access protection.

However, ensuring TA data integrity and confidentiality under devices requires more than just a trusted path. Accelerators like FPGAs or GPUs can leak TA information, necessitating a trusted device architecture. This can be achieved through spatial or temporal partitioning, ensuring exclusive resource usage and secure context switching. Spatio-temporal partitioning is useful for scenarios requiring flexibility, like multi-tenancy. While cryptography can enforce isolation, its performance overhead and cost implications make it less preferable for hardware implementation.

3.1.1.4 TEE Secure Storage

To protect security-critical data TEEs require a secure storage mechanism. The most straightforward way to ensure the security of stored data is through a sealing and unsealing process. The process of sealing involves the protection of data through an encryption process. Contrarily, the process of unsealing involves the process of decryption of this persistent data, i.e., through cryptography is possible to ensure that any data is not available to unauthorized parties, and that any data tampering is detected.

Encryption can be implemented in software, by using software modules to encrypt information, or through dedicated hardware, involving higher costs but improved performance. The process of encryption follows a symmetric or an asymmetric technique, where a symmetric maintains the same key for the encryption and decryption process, and an asymmetric process uses different keys for different processes. Advanced Encryption Standard (AES) is the most common symmetric encryption algorithm, while Rivest-Shamir-Adleman (RSA) is the most common asymmetric encryption algorithm [7].

3.1.2 TEE Implementations

This section explores various advancements in hardware-based security technologies, focusing on solutions offered by Arm, Intel, AMD, and RISC-V architectures.

Arm TrustZone. 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.'' 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/FFA. 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 EL2 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. 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 FF-A (Firmware Framework for Arm) [10]. 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 OSs and hypervisors, making it easier to reuse certified Trusted Firmware and hypervisor configurations across various setups.

Arm CCA. 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 [11]. 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 the Realm, a secure, isolated environment. 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.

TrustZone-M. Arm TrustZone for Cortex-M[12], TrustZone-M, is integrated into select Armv8-M architecture microcontrollers and offers a hardware-based security framework through two orthogonal states: secure and non-secure. Unlike its TrustZone-A counterpart, TrustZone-M utilizes a memory map approach; 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, the latter allowing secure entry points within a specific non-secure callable region. To perform access control attribution units, filling a similar role to the existing TrustZone controllers are used.

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 (TDs) [14]. Similar to SGX, TDX's primary objective is to safeguard its isolated environment, specifically TDs, 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 SEAM (Secure-Arbitration Mode). 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.

Physical Memory Protection. The RISC-V architecture features the PMP, a mechanism designed for performing access control over system resources [16]. 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 access permissions. Although the number of entries is implementation-specific, it will necessarily be limited when compared to virtual memory-based solutions. 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 permissions for each core, and if needed, it can dynamically reconfigure the access control policies.

Confidential Virtual Machine Extensions. The RISC-V Confidential VM Extension (CoVE) represents RISC-V's response to the confidential computing UC [17]. 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 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 Supervisor (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.

Figure 2 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 left side, TrustZone establishes communication between the OS in the normal world and the TEE in the secure world via a secure monitor. FFA introduces a trusted hypervisor to oversee trusted VMs. CCA expands TrustZone to incorporate the Realm world, housing the Realm Manager (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 VMs, granting the option for full protection at startup or selective memory protection. RISC-V Instruction Set Architecture (ISA) employs Physical Memory Protection (PMP) to establish security domains, with SBI managing context switching.

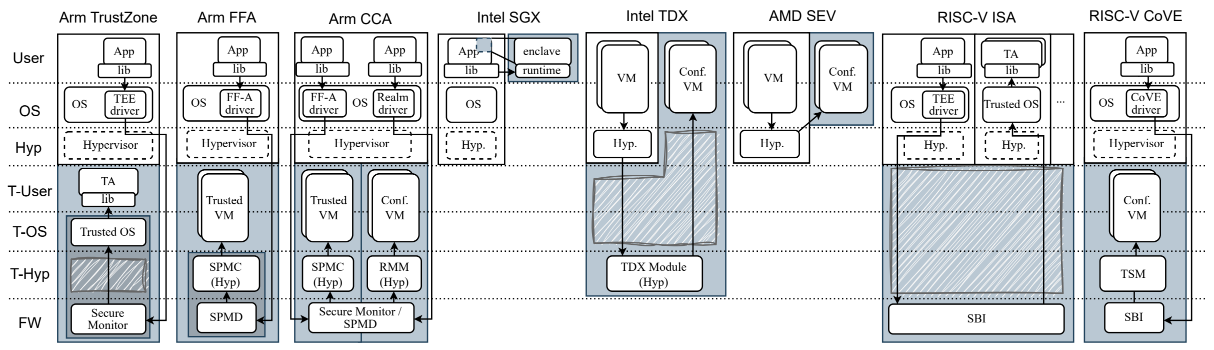


Figure 2: TEE models representation.

3.1.3 TEE Vulnerabilities

As mentioned in 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. In the future, TEEs are expected to be integrated into trillions of IoT devices. Unfortunately, studies have continually demonstrated vulnerabilities in TEE systems across vendors. For example, in [1], we analyse 207 TEE bug reports on Arm-based devices, from 2013 until mid-2018. In this section, we continue this work by collecting and analyzing numerous vulnerabilities and limitations affecting TEEs from 2019 onwards. Despite the widespread utilization of this technology across hundreds of millions of devices, TEEs have encountered numerous successful attacks in recent years [1]. Our analysis centers on Common Vulnerabilities and Exposures (CVE) reports for TEE systems developed by various vendors, applicable to both APUs and MCUs classes of devices. Through a detailed examination of publicly documented exploits and vulnerabilities dating back to 2019, we have identified critical weaknesses within existing TEE implementations.

Before delving into the analysis of TEE vulnerabilities, we establish certain assumptions aligned with our project goals. Our analysis focuses exclusively on vulnerabilities within TEE components, not including any vulnerability present on the REE side. Specifically, our analysis considers vulnerabilities where attackers could acquire secrets from the TEE, access secrets from the REE, or otherwise escalate privileges to the TEE.

In the analysis of the TrustZone TEE systems, we've considered those developed by Qualcomm, AMD, Samsung, Trustonic, Nvidia, Linaro, and Arm. Qualcomm's TEE solution is known as Qualcomm Trusted Execution Environment (QTEE)[18]; AMD PSP, a key component in AMD SEV, leverages TrustZone technology [15]; Samsung maintains mTower[19] and TEEGRIS[20]; Trustonic maintains kinibi, Nvidia uses Trusty, TZVault, and Trusted Little Kernel (TLK) trusted OSes accross its products; Linaro maintains OP-TEE, an open-source TEE software widely popular for TrustZone development; and Arm maintains Trusted Firmware-M (TF-M) [**Armtfm**], which implements a Secure Processing Environment (SPE) for Armv8-M. These systems are actively maintained, widely adopted for commercial purposes, and offer a substantial amount of information for analysis. Henceforth, for clarity, we will reference each analyzed TEE by the respective company name rather than the software denomination (e.g., "Linaro TEE" for OP-TEE).

Table 3: Sources of reports: CVE (CVE databases), SP (scientific publications) and SB (security bulletins).

TEE System		**CVE**	**SP**	**SB**	**Total**	
					MCU	**APU**
Qualcomm		5	0	43	0	48
AMD		3	0	2	0	5
Samsung		26	13	2	13	28
Trustonic		1	0	0	0	1
Nvidia		2	0	21	0	23
Linaro		13	0	0	0	13
Arm		6	0	0	6	0
Total	**MCU**	6	13	0	19	■
	APU	50	0	68	■	118

We investigate various sources categorized into three main areas, as detailed in Table 3. Our analysis encompassed bug reports sourced from: the CVE database [21] pertaining to the TEE systems under investigation; scientific publications; and CVE reports officially published by Qualcomm [22], AMD [23], Samsung [24] and Nvidia [25], documented in the respective security bulletins. The CVE reports were gathered through keyword searches for terms such as the names of TEEs, "TEE," "Trusted," " TA ," "TrustZone," etc. Additionally, we also distinguish the CVE reports from TEEs implemented for MCU and APU classes of devices.

We collect a total of 136 CVE vulnerabilities. Qualcomm TEE stands out as the TEE with the highest number of vulnerabilities among the categorized TEEs, totaling 48 CVEs. Meanwhile, recent MCU TEE designs like Samsung TEE (mTower) and Arm TEE (TF-M) present a total of 19 vulnerabilities.

After collecting the vulnerability reports, we manually analyzed and categorized them. For the vulnerabilities assigned with a CVSS score [26], we adopted a classification metric based on the attribute score, comprising it into four categories: critical (CVSS ≥ 9), severe (CVSS [7,9[), medium (CVSS [5,7[), and low (CVSS [0,5[). The severity of a specific vulnerability may have different security implications. A critical vulnerability is normally one that can lead to a complete compromise of confidentiality or integrity in the TEE, in the REE, or both.

Table 4 quantifies the number of disclosed vulnerabilities associated with each system according to their severity. The results highlight that:

- **Samsung's TEE system** has the highest number of critical vulnerabilities (20) and the second-highest number of severe vulnerabilities (19). Most of the critical issues come from out-of-bounds read/write accesses (e.g., CVE-2019-20537) and from the use of incompatible types on TEEGRIS (e.g., CVE-2019-20571).
- **Qualcomm's TEE system** has the highest number of severe vulnerabilities with a total of 34, 2 CVEs

Table 4: Number of disclosed CVEs per system from 2019 to 2024.

TEE System	Critical	Severe	Medium	Low	Total
Qualcomm	2	34	12	0	48
AMD	0	2	2	1	5
Samsung	20	19	1	1	41
Trustonic	1	0	0	0	1
Nvidia	0	8	11	4	23
Linaro	7	5	1	0	13
Arm	0	4	2	0	6
Total	30	72	29	6	137

critical vulnerabilities and 12 CVEs classified as medium. Concurrency and memory management issues (e.g., CVE-2019-10589, CVE-2022-33257, CVE-2022-33273) represent the most critical and severe Qualcomm TEE vulnerabilities.

- **Linaro's TEE system** has the second-highest number of critical vulnerabilities (7 CVEs), with all categorized as memory management issues (CVE-2019-1010296), and 5 severe vulnerabilities, also inherited from memory management issues (CVE-2019-1010294), improper input handling (CVE-2022-46152) and from the lack of security access configuration (CVE-2021-44149)
- **Nvidia TEE system** shows no critical vulnerabilities, 8 for severe (e.g., CVE-2021-34374), 11 for medium (e.g., CVE-2021-34385) and 4 for low criticality level (e.g., CVE-2021-34393).
- **AMD TEE system** shows no critical vulnerabilities and has a total of 5 vulnerabilities, distributed across all severity levels (e.g., CVE-2020-12931).
- **Arm TEE system** shows no critical vulnerabilities and has a total of 6 vulnerabilities, four classified as severe (e.g., CVE-2021-43619) and two classified as medium (e.g., CVE-2021-40327).
- **Trustonic TEE system** shows 1 critical vulnerability, where trustonic Kinibi allows arbitrary memory mapping (CVE-2020-13831).

After collecting the vulnerability reports and assessing their severity, we proceeded to analyze the underlying causes of these vulnerabilities using the Common Weakness Enumeration (CWE) framework. CWE provides a standardized taxonomy for describing security weaknesses, enabling us to categorize and understand the root causes of vulnerabilities more effectively.

By mapping CVEs to CWEs, we identified the specific weaknesses or flaws in software systems that led to the reported vulnerabilities. Additionally, our analysis revealed a range of CWEs that can be categorized into distinct issue categories based on their nature and impact. Table 5 summarizes the CWEs categorization by mapping CWE to the correspondent issue categories.

The grouping of CWEs into issue categories facilitates the process of mapping TEE system vulnerabilities by preventing redundant issues from being categorized into different categories. For example, CWE-119 (improper restriction of operations within the bounds of a memory buffer) and CWE-125 (out-of-bounds read), both refer to memory management issues by not preventing buffer overflow. Additionally, as part of this categorization is important to note that the total number of CWEs may exceed the number of CVEs themselves, i.e., one CVE can be mapped to multiple CWEs. For instance, CVE-2021-34376 is assigned to both CWE-20 (Improper Input Validation) and CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), indicating dual categorization, either for improper input handling or memory management issues. Table 6 presents how many CVEs for each TEE map to each CWE category.

Qualcomm. Qualcomm reveals numerous vulnerabilities across diverse categories. There are 33 instances of memory management issues, indicating potential risks in data storage and memory access.

Table 5: List of CWE categorization.

Category	List of CWEs
Improper Input Handling	**CWE-20**: Improper Input Validation, **CWE-94**: Improper Control of Generation of Code, **CWE-129**: Improper Validation of Array Index, **CWE-754**: Improper Check for Unusual or Exceptional Conditions, **CWE-755**: Improper Handling of Exceptional Conditions, **CWE-307**: Improper Restriction of Excessive Authentication Attempts
Memory Management Issues	**CWE-119**: Improper Restriction of Operations within the Bounds of a Memory Buffer, **CWE-120**: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'), **CWE-125**: Out-of-bounds Read, **CWE-190**: Integer Overflow or Wraparound, **CWE-191**: Integer Underflow or Wraparound, **CWE-252**: Unchecked Return Value, **CWE-401**: Missing Release of Memory after Effective Lifetime, **CWE-415**: Double Free, **CWE-416**: Use After Free, **CWE-476**: NULL Pointer Dereference, **CWE-697**: Incorrect Comparison, **CWE-732**: Incorrect Permission Assignment for Critical Resource, **CWE-787**: Out-of-bounds Write
Authentication and Authorization Issues	**CWE-287**: Improper Authentication, **CWE-285**: Improper Authorization, **CWE-862**: Missing Authorization, **CWE-863**: Incorrect Authorization
Cryptographic Issues	**CWE-330**: Use of Insufficiently Random Values, **CWE-327**: Use of a Broken or Risky Cryptographic Algorithm, **CWE-347**: Improper Verification of Cryptographic Signature
Concurrency Issues	**CWE-367**: Time-of-check Time-of-use (TOCTOU), **CWE-362**: Concurrent Execution using Shared Resource with Improper Synchronization
Information Disclosure	**CWE-200**: Exposure of Sensitive Information to an Unauthorized Actor, **CWE-203**: Observable Discrepancy, **CWE-212**: Improper Removal of Sensitive Information Before Storage or Transfer
Resource Management Issues	**CWE-770**: Allocation of Resources Without Limits or Throttling, **CWE-276**: Incorrect Default Permissions
Type Handling Issues	**CWE-704**: Incorrect Type Conversion or Cast
Type Confusion Issues	**CWE-843**: Access of Resource Using Incompatible Type
Other Issues	**CWE-191**: Integer Underflow (Wrap or Wraparound), **CWE-269**: Improper Privilege Management, **CWE-254**: Security Features, **CWE-1066**: Missing Serialization Control Element

Table 6: Mapping of each analysed CVE to the respective group of CWEs.

CWE	Qualcomm	AMD	Samsung	Trustonic	Nvidia	Linaro	Arm
Improper Input Handling	6	2	9	1	8	2	1
Memory Management	33	3	21	1	16	9	5
Authentication and Authorization	2	0	1	0	0	0	1
Cryptographic	0	0	0	0	1	2	0
Concurrency	2	1	1	0	0	0	0
Information Disclosure	4	0	1	0	0	0	0
Resource Management	0	0	2	0	3	0	0
Type Confusion	0	0	9	0	0	0	0
Type Handling	2	0	0	0	0	0	0
Other	0	0	0	0	2	1	0
Total	49	6	44	2	30	14	7

Additionally, 6 CWEs were identified in input validation handling, posing risks of unauthorized data infiltration.

AMD. AMD reveals 3 memory management issues, 2 improper input handling issues, and 1 concurrency issue.

Samsung. Samsung emerges as the one with vulnerabilities distributed across more categories. Among them, Table 6 underscores 9 CWEs in improper input handling, 21 in memory management, and 9 in related to data type confusion issues.

Trustonic. Trustonic's 2 vulnerabilities are categorized as memory management and improper input handling.

Nvidia. Nvidia presents 16 CWEs categorized as memory management, 8 as improper input handling, 3 as resource management, and 2 as other issues (related with CWE-1066: Missing Serialization and CWE-754: Improper Check for Unusual or Exceptional Conditions).

Linaro. Linaro exhibits 9 issues identified as memory management, warranting attention to mitigate potential data integrity risks, 2 observed input validation issues, 2 cryptography, and 1 other issue (related to CWE-254: Security features).

Arm. Arm exhibits 5 issues identified memory management, 1 classified as authentication and authorization issues (related with CWE-862: Missing Authorization), and 1 classified as improper input handling (related to CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')).

Overall, the mapping allows for a clear understanding of the prevalent weaknesses across different TEE implementations (i.e., the memory management issues), aiding in the identification of common security challenges. Additionally, it is evident that across all TEE technologies under study, there was at least one memory management issue and one improper input handling issue.

Our analysis reveals widespread vulnerabilities across different TEE implementations, notably in memory management and input handling. All analyzed trusted OSes exhibited issues in these areas. Therefore, decomposing TEE software into multiple isolated environments remains crucial to mitigate these vulnerabilities and enhance overall system security.

3.1.4 TEE Isolation

Arm's TrustZone is widely used in the context of embedded and IoT systems. However, there's a significant issue with the secure-world privilege level in TrustZone [1, 2]. Secure world components, e.g., trusted OS, can access any system resource in the entire system, leading to high-impact security-critical vulnerabilities. To address this, we propose breaking down the monolithic design of TrustZone TEEs into multiple domains. This decomposition should enable multiple trusted OS to co-exist securely in the secure world, allowing the decomposition of trusted OS stacks into multiple isolated environments, for example, to isolate system functionality TA from third-party TA.

3.1.4.1 TEE Isolation Feature: APU TEE Isolation

Several works have targeted the topic of TEE isolation [27, 3, 28, 2], with FFA on Armv8.4 also enabling this. To address this issue in commercial off-the-shelf platforms, existing research suggests three strategies: i) implementing software-based virtualization techniques in the secure world, ii) utilizing existing control units and auxiliary processors, and iii) transferring the secure world software stack to the normal world using virtualization techniques.

Software-Based Virtualization: Works like TEEv [3] and PrOS [28] 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 slightly, 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 also needs to be a well-defined and secure entry point that transitions the execution from the guests to the hypervisor and vice-versa.

Repurposed Control Units: It is common for platforms to feature system-wide control units 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 [2].

Virtualization: Approaches such as vTZ [27], MyTEE [29], and TEEVseL4 [30], leverage normal world virtualization to execute trusted OSes in VMs.

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 necessitate. Instead, CROSSCON promotes the use of existing control units and advocates for hardware-assisted virtualization in the REE.

APU TEE Isolation - Repurposed Control Units

Our approach is based on ReZone [2] and thus relies on similar assumptions and mechanisms. ReZone requires that the platform features a Platform Partition Controller (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 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 3 illustrates the architecture for this solution.

Hardware Architecture. Hardware-wise, the system relies on a typical TrustZone-enabled platform. For controlling memory access permissions, in addition to a 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 recon-

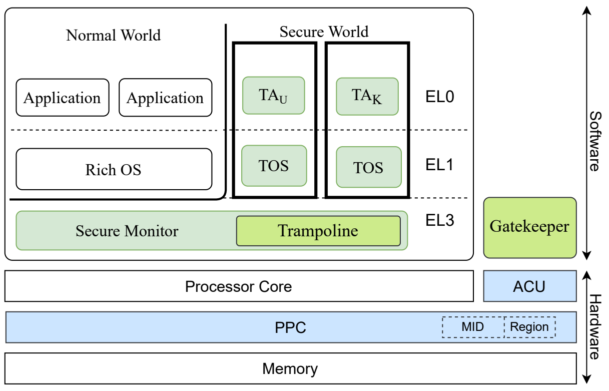


Figure 3: Overview of Trustzone's architecture decomposition on secure world.

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

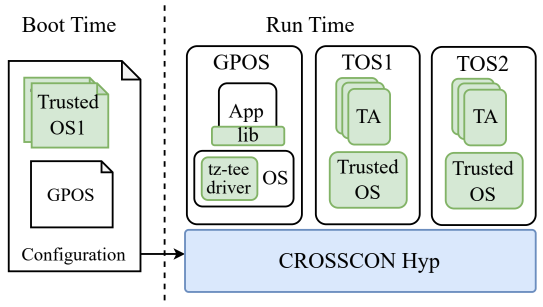
Software Architecture. Software-wise, secure world TEE decomposition comprises the *secure monitor* and the *gatekeeper*. The former consists of standard secure monitor software (e.g., implemented by Arm Trusted Firmware) augmented with a secure world TEE decomposition-specific sub-component named *trampoline*. 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 guest 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

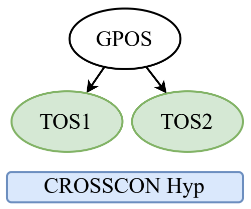
An important security limitation in TrustZone is the Trusted OS's access privileges [2, 1]. The Trusted OS can access any normal and secure world resource, while the normal world can only access normal world resources. With the CROSSCON Hypervisor, it is possible to restrict the memory access privileges of Trusted OSes running in VMs, preventing them from arbitrarily accessing the resources of the GPOS. Additionally, we allow for the instantiation of multiple TrustZone-TEEs serving one GPOS. Several works have targeted this topic [27, 3, 28, 2].

Software Architecture. To implement this feature, we build upon CROSSCON Hypervisor support for per-VM TEE support and develop it further. For details on running trusted OSes in VMs in CROSSCON Hypervisor, refer to section 3.2.5.2. To support multiple TEE VMs per VM we've updated the TEE CROSSCON Hypervisor internal module, to be able to identify which TEE VM the GPOS is requesting interaction with. Additionally, this required modifications at the GPOS level, to be able to provide TEE driver instances for each TEE separately. This means that when performing SMC calls, the OS will use different IDs depending on the targeted trusted OS.

Figure 4a illustrates the architecture. A configuration file distributes the resources over the GPOS and TOS VMs. The CROSSCON Hypervisor can host several GPOS and trusted OS VMs depending on the system design and requirements. Figure 4b illustrates the runtime VM hierarchy, where the GPOS controls the execution of one or more Trusted OSes. The trusted OSes can be used for different purposes. For



(a) TEE isolation architecture.



(b) VM execution hierarchy.

Figure 4: TEE decomposition in normal world.

example, one Trusted OS can serve only system security needs, while other Trusted OS instances host application-focused functionality (e.g., DRM, electronic payments), with one trusted OS VM provided for each trusted service developer.

3.1.4.2 TEE Isolation Feature: 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 TEE software stacks in this context has been put forth by uTango [31]. In this scenario, the secure world is comprised of monitor software that performs context switching between software stacks in the normal world. This approach allows for effective isolation and secure operation of different software components. We adapt this solution for the CROSSCON Hypervisor on MCUs. This idea will be further developed and presented in the final WP3 deliverables.

3.1.4.3 TEE Isolation Feature: RTU TEE Isolation

Currently, there are no TEE solutions explicitly designed for Real-Time Processing Unit (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. This idea will be further developed and presented in the final WP3 deliverables.

3.1.5 TEE Abstraction

TEEs are often complex to program, requiring direct interaction with the underlying firmware to manage the various security hardware modules. In response, industry and academia offer trusted OSes that can abstract most of the complexity of the underlying systems. These OSes typically provide fundamental security services, including cryptographic primitives and secure storage capabilities. Still, they support the deployment of TAs, i.e., applications that leverage the trusted OS to offer specific services to the unsecured world. Execution of such services requires TAs to interact seamlessly with the trusted OS, whether for soliciting cryptographic keys or storing critical data. Consequently, the interface between the trusted OS and the TA emerges as a key aspect in the TA's development lifecycle.

Several trusted OSes exist. Section 3.1.3 analysis QTEE, OP-TEE, AMD-TEE, mTomwer, TEEGRIS, Kinibi, Trusty, TZVault. To aid the TA developers, these trusted OSes often offer SDK kits and extensive documentation, thus allowing the development of a TA with the ability to interact with their specific trusted OS. Although the trusted OSes are designed and implemented differently from one another, their interface often adheres to a standard to facilitate the TA developers. Unfortunately, some trusted OSes (e.g., Trusty) don't make such an effort, sticking to proprietary APIs and hampering the TA developers. In addition, proprietary operating systems that adhere to a standard rarely release public documentation for their APIs.

To facilitate the TA development and enable TAs interoperability, i.e., allowing them to be run unmodified on different trusted OSes, CROSSCON envisions a public TEE abstraction model as a set of APIs that should be implemented by any TEE running on top of the CROSSCON stack.

In addition to interoperability at the TA level, interoperability at the TEE model level is also a concern. Given all TEE models presented in Figure 2, it is clear to understand their differences in system stack structures and hardware-based technologies they rely on. Since TEE technologies are widely available on different COTS platforms produced by different vendors, relying on different dedicated hardware, e.g., Arm TrustZone, Intel SGX, or AMD-SEV, they tend to be more heterogeneous. Additionally, given the market's competitive growth, there's a tendency for TEE vendors to produce TEEs independently with proprietary features enforcement mechanisms, and protection models. Due to this TEE heterogeneity, interoperability and compatibility challenges emerge as TEE developers struggle in terms of reusability, requiring them to be proficient in various TEE technologies when moving platforms.

TEE Standard Selection

Initially, our goal was to map interfaces of different TEEs; however, given the widespread adoption of Global Platform TEE specifications [4] among different TEE implementations, interoperability is not a significant concern at the TA level. In this sense, rather than implementing an abstraction model from scratch, we choose to follow an existing set of APIs to boost the compatibility of CROSSCON TEEs with existing systems.

We consider mainly two different sets of APIs: PSA Certified API and GlobalPlatform APIs. PSA Certified APIs are part of a broader set of standards (PSA Certified), whose goal is to establish a security baseline from hardware to software, paving the way for the development of comprehensive secure systems. GlobalPlatform APIs focus on the software interfaces between TEE OS, TA , and Client Application (CA). PSA Certified APIs are proposed by Arm, and, although are compatible with any architecture, they are part of an Arm-focused vision. On the contrary, GlobalPlatform APIs are completely independent of any architecture or chip manufacturer. Furthermore, they currently represent the leading standard, with wide adoption from various chip manufacturers and software solutions. Most trusted OSes adhere to these APIs, such as iTrustee, OPTEE, QTEE, and Kinibi, thus making them CROSSCON compliant. For these reasons, we choose the GlobalPlatform APIs as our TEE Abstraction Model.

Global Platforms APIs are divided into two major sets: the client APIs and the core APIs. Although both of these are important for the interoperability of security services, they define different interfaces. The client APIs regulate the interaction between an untrusted client and an arbitrary TA . The core APIs regulate the interaction between TA and TEE OS.

TEE Technology heterogeneity challenges

The challenge of TEE heterogeneity has prompted researchers to explore various solutions, including emulation and hardware abstraction. Emulation efforts, such as HyperEnclave, aim to enable TEE functionality on platforms that lack native support, like Arm and AMD servers. However, these approaches often lack support for coexisting TEE programming models and limit the development of novel or customizable TEEs. Keystone introduced hardware abstractions for customizable TEEs, utilizing RISC-V PMP, but its reliance on RISC-V architecture hampers cross-platform portability and lacks seamless compatibility with existing TEE programming models, such as Arm TrustZone.

In response to TEE heterogeneity, two main challenges were considered, i.e., portability and interoperability. Portability refers to developers being able to run legacy stacks across various platforms and architectures. Interoperability, allows them to run multiple and different TEE models simultaneously.

We aim to develop a solution that tackles interoperability and portability challenges among TEE technologies by addressing existing approach limitations. Our proposal utilizes hardware virtualization primitives found in COTS platforms such as Arm VE, Intel-VT, and RISC-V Hypervisor extension. These primitives are employed to establish TEE-compatible isolated execution environments by emulating custom platform behavior.

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 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 as many microarchitectural resources such as last-level caches, interconnects, network stack, and memory controllers remain shared among partitions. Existing VM, hypervisor, and containers technology (i.e., microK8) often depend on a large general-purpose OS (typically Linux) either to boot, manage VMs, or provide a myriad of services, such as device emulation or virtual networks, thus are unsuitable to host security-critical applications.

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 layer. However, because static partitioning hypervisors 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 or VM Monitor (VMM), 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.

IO 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, System on Chip (SoC)s 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.

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 [**Armeverywhere**]. 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) [**Arm_virt**] 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 [**Arm_virt**]. 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 General Interrupt Controller (GIC). The GICv2 [**Arm2022gicv2**] 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 [**Arm2022gic**] 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 Physical Addresses (IPAs), which are then subject to a second stage of translation, mapping IPAs to system physical addresses. The first stage of translation is managed by the guest OSes. The second stage, 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 [32] and SMMUv3 [33], 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 [34] is an open-source, royalty-free ISA for designing computer processors gaining significant traction in the last few years. Unlike proprietary ISAs, 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 [35]: (i) machine mode (M-mode) is the most privileged level, hosting the firmware which implements the supervisor binary interface (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 [35]. The RISC-V Hypervisor extension execution model follows an orthogonal design where the supervisor mode (S-mode) is modified to a hypervisor-extended supervisor mode (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 addresses 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) [36] 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. In response to the PLIC limitations, the RISC-V community developed a new interrupt controller specification. The RISC-V Advanced Interrupt Architecture (AIA) [37] 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 APLIC and IMSIC. The protection against undesirable accesses is guaranteed at the core level, via the PMP, and at the system level via the IO Physical Memory Protection (IOPMP) or RISC-V IOMMU. Virtualization support is offered through the MSIC, while APLIC offers compatibility with interrupts signaled by wire on devices that do not support MSI.

IO Memory Management Unit. The RISC-V IOMMU defines three methods for managing DMA-capable devices using virtual memory [38]. 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 Real-Time OS (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 [**Armv8r-virt**] 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) [39]. Current work in WP4 is integrating a preliminary version of this specification into a BA5x core.

Interrupts. Interrupt management in real-time virtualization processors closely resembles that of application class virtualization. The interrupt controller, a shared peripheral, requires the hypervisor to control access and mediate guest's access. For Arm real-time platforms, a GICv2 controller, specifically the GIC 400, is typically featured. In the case of RISC-V, WP4 is currently working on implementing the APLIC in 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 IOPMU 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. 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

Microcontrollers 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. Microcontroller Units (MCUs) do not have built-in virtualization facilities, such as an extra CPU execution mode for the hypervisor or virtual memory. However, some techniques can overcome some of the hardware limitations [40, 41, 42, 43]. Platforms with CPUs with Armv6-M, Armv7-M, and Armv8-M without Trustzone, architectures feature two privilege modes, thread and handler modes, with handler mode being more privilege and being able to restrict access to software running in thread mode e.g., through the MPU. In these platforms virtualization is achieved through para-virtualization, meaning that the guests are aware of the underlying hypervisor, and are modified to collaborate with a hypervisor through specific hypercalls for performing sensitive/critical functions. RISC-V platforms targeting the same class of performance also offer two privilege modes, in this case, the machine and user modes. However, RISC-V MCUs benefit from a fully virtualizable architecture, meaning that it is possible to transparently execute originally high-privilege software at a low-privilege level and securely handle the execution of high-privilege operations. On Armv8-M platforms featuring TrustZone-M, the higher privilege of the secure world can be leveraged to execute software that controls the normal world execution, allowing for the establishment of access control over system resources and interrupt management [42]. In other words, it is possible to use the Armv8-M secure world to implement a hypervisor. Software executing in the normal world cannot access the hypervisor's CPU state or memory. Execution control can be taken, for example, through secure world interrupts, explicit invocations, or exceptions on accesses to specific memory regions. For memory protection, the guest configuration of the MPU must be saved and restored during context switches.

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. This allows for seamless transitions between different guests, ensuring each has the appropriate access to system resources.

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. These units control devices' access to normal and secure world memory. A straightforward approach to solve this might involve reprogramming this protection unit 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 these optimizations, 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 [44, 45, 46, 47].

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 reaccess 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 and Meltdown 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. The arbitration policy may result in leaking information by enabling an attacker 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 [48].

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, 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 last-level cache is not practical. Even without the last-level cache, 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.

3.2.3 Hypervisors Feature Analysis and Selection

CROSSCON Hypervisor utilizes a static partitioning hypervisor as a starting point, providing 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 built-in mechanisms like cache coloring, which ensures isolation for shared resources such as last-level caches.

The use of a thin static partitioning hypervisor 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 (SPV)

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 virtual CPUs to physical CPUs (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). Static Partitioning Hypervisor (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 [49], 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 last-level cache (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 hardware 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 [50]. 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) [**Arm_sdei**] 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.

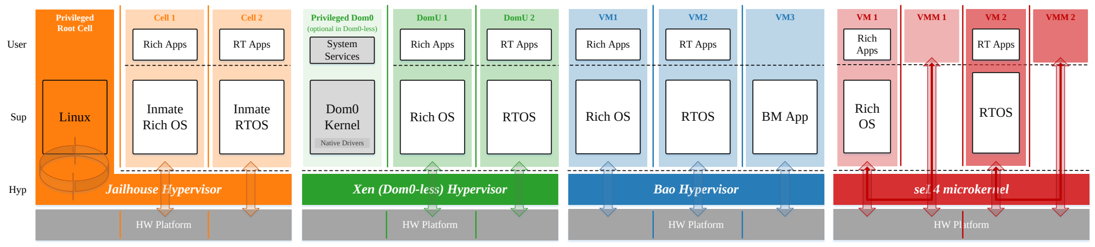


Figure 5: Architectural overview of the assessed hypervisors: Jailhouse, Xen (Dom0-less), Bao and seL4 CAmkES VMM.

3.2.3.2 Static Partitioning Hypervisors

Multiple static partitioning hypervisors are currently available, developed by both academia and industry. This section highlights the main features of the currently available works (depicted in Figure 5).

Jailhouse Hypervisor. Jailhouse [51, 52] 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[53]. The research community has been actively contributing with mechanisms to enhance predictability, namely: cache coloring, DRAM bank partitioning [54], memory throttling, and device quality of service (QoS) regulation [55]. An unofficial fork including these features is available [56]. Direct injection [57] was also implemented.

Xen (Dom0-less) Hypervisor. Xen [58] 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 [59] 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 [60] as Dom0. Besides Armv8-A, Xen also supports x86, and Armv8-R and RISC-V ports are underway.

Bao Hypervisor. Bao [61] is an open-source static partitioning hypervisor 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 [61]. The mainline now includes support for RISC-V [62], 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 [63]. 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 [64]. A user-level VMM uses its resource capabilities to create VMs. As of this writing, only the seL4 CAmkES VMM [65, 66] 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 [67]. Cache coloring has also been implemented in seL4 [68], 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 [69, 70]. There are also ongoing efforts to extend the formal verification to prove the absence of covert timing channels [71]. Finally, CAmkES is being deprecated shortly in favor of the seL4 Core Platform (seL4CP) [72].

3.2.4 Static Partitioning Hypervisor Analysis

We have performed an analysis of static partitioning hypervisors in [73]. 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 latencies.

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 [**Arm2022gic**]. 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 physical CPUs; 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 static partitioning hypervisors seL4 CAmkES, Xen (Dom0-less), Bao, and Jailhouse, across the following dimensions:

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 static partitioning hypervisors 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 and Design

The CROSSCON Hypervisor, being built upon Bao, is grounded in a static partitioning hypervisor. However, the inherent constraints of pure static partitioning hypervisors 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. Moreover, specific use cases necessitate a full-fledged hypervisor, prompting the need for simultaneous support of two or more hypervisors. This section outlines the preliminary design and implementation of the first two features: Dynamic VMs and per-VM TEE services. Additionally, microarchitectural isolation features will be implemented in the final demo and included in the final WP3 deliverables, i.e., D3.3 and D3.4.

Multiple VMs executing on one CPU. A common requirement, for both the Dynamic VM Creation and Management and per VM TEE Support features, is the ability for vCPUS belonging to different VMs to share a single physical CPU.

Our initial approach to enabling VMs to share CPUs is based on the observation that systems typically feature one main OS that requests services from other system components.

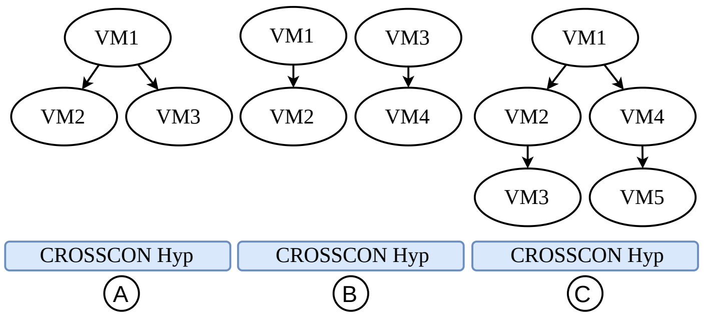


Figure 6: VM parent and VM child dynamic.

Figure 6 illustrates the currently implemented model. 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 featuring its own child; and (C) child VMs featuring their own children.

Combinations of these instantiations are supported. We have specified a CROSSCON Hypervisor's configuration file to allow for the establishment of the parent-child dynamic between VMs, further details are provided in D3.2.

The CROSSCON Hypervisor must provide interfaces that allow for the invocation of, and the return of execution from, child VMs. These APIs are the push and pop hypervisor calls. The scheduling burden is placed on the main OS to decide which VMs to invoke and when. However, this does not prevent the main OS from receiving timely interrupts. Our initial design models these execution requests as a stack. When a parent VM, VM1 in Figure 7, yields execution to a child VM, VM2 in the figure, the parent VM is placed into the execution stack.

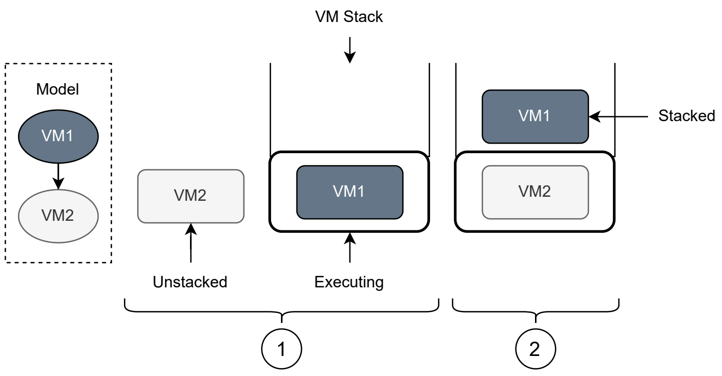


Figure 7: VM execution stack for CPU sharing.

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.

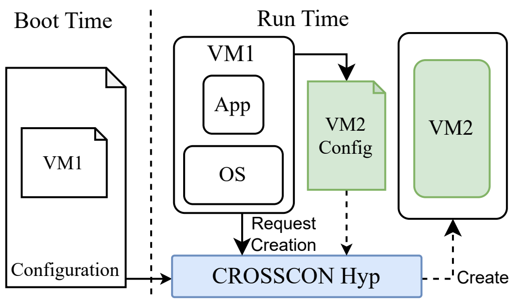
3.2.5.1 Feature 1: Dynamic VM Creation and Management

During the boot time, CROSSCON Hypervisor initiates the VMs setup by reading a 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. To leverage existing mechanisms to implement dynamic VM creation, we allow guests to request VM creation using the existing config file infrastructure. This is achieved by sharing a configuration file with the hypervisor that will be parsed in run time. Additionally, it is necessary to enable multiple VMs to execute concurrently on a single CPU. For this, we leverage the mechanisms detailed above. The implementation of dynamic VMs was explored in part in the, yet-unpublished, paper "AnyTEE: An Open and Interoperable Software Defined TEE Framework".

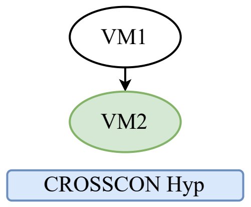
Dynamic VMs. For dynamic VM creation, VMs need access to a hypervisor interface that enables them to send a config file to the hypervisor, specifically through the VM create hypercall. The CROSSCON Hypervisor then parses this file to instantiate the new child VM. During the creation of a dynamic VM, all resources are verified to originate from the parent VM, with the hypervisor ensuring that the parent VM does not reference any resource outside its scope. After this, the hypervisor proceeds to instantiate the child VM, while removing all resources, except for the physical CPUs, from the parent VM. This process ensures the integrity and isolation of each VM, crucial for system security. Figure 8 illustrates the boot time setup of one VM, VM1, and the dynamic creation of a second VM, VM2. Dynamic VMs require that a CROSSCON Hypervisor driver on the host OS interacts with the CROSSCON Hypervisor through the CROSSCON Hypervisor hypervisor call interface. This interface serves three main objectives: VM Creation, VM Destruction, and VM Invocation. A communication protocol is implemented between the application and TA to establish a connection.

Figure 8a illustrates the architecture for implementing dynamic VMs. A configuration file establishes the GPOS VM at boot time. Additional config files are loaded by the GPOS and are used to create additional VMs. Figure 8b illustrates the runtime VM hierarchy, where the GPOS controls the execution of the VM created dynamically.

VM Creation. When a parent VM requires the creation of a child VM, it will issue a request to the OS through the CROSSCON Hypervisor driver to allocate memory for the child VM. The OS will then take some of its memory, and place in it the code and data of the child VM. The child VM image is previously



(a) Dynamic VM Architecture.



(b) Dynamic VM execution hierarchy.

Figure 8: CROSSCON Hypervisor dynamic VM support.

created and stored in a file. After the OS 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, 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 TA 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. CROSSCON Hypervisor will destroy the VM, another modification we introduce in CROSSCON Hypervisor. 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.

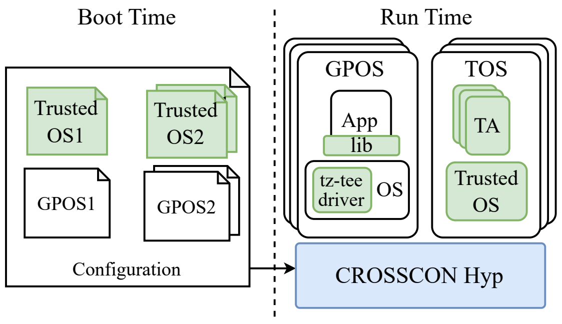
Programming Model. Our current implementation of dynamic VMs follows an enclave programming model. Our implementation relies on a user-space runtime and a tailored CROSSCON Hypervisor kernel driver. The user space runtime provides three core functions to host applications: `create_vm`, `destroy_vm`, 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.

3.2.5.2 Feature 2: Per VM TEE service support

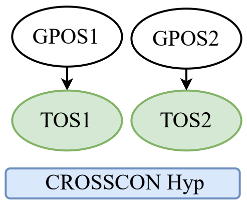
To offer per-VM TEE support, we leverage the VM stack mechanism to enable VMs to share physical CPUs. Furthermore, we utilize the VM stacking model to bind VMs together, thereby linking a GPOS VM with a Trusted OS VM. We implement the execution of a trusted OS within a VM, binding it to a single GPOS VM. This concept is further explored in section 3.1.4, where we provide a solution to move from a single trusted OS in the secure world to multiple trusted OS VMs running in the normal world.

Software Architecture: To offer per VM TEE services, we introduce the TEE and GPOS VM concepts within the CROSSCON Hypervisor. TEE VMs serve as hosts for a trusted OS, such as OP-TEE, and are bound to single GPOS VMs, typically running Linux. We adopt a modular approach within the CROSSCON Hypervisor, implementing dedicated modules for each VM type: GPOS and TEE. GPOS VM events are handled by a dedicated GPOS module within the hypervisor. Similarly, Trusted OS VM events are also handled by a dedicated module to deal with both trusted OS operations and relevant GPOS events, including GPOS interrupts and trusted OS-related calls. Moreover, we modify Bao's configuration file in two ways. The first adds support for developers to establish a parent/child hierarchy between VMs. The second adds a VM type field to identify VMs as either GPOS or TEE, allowing the CROSSCON Hypervisor to effectively manage events specific to each VM type.

During runtime, requests for the TEE typically originate from applications. These applications interact with the trusted OS driver, initiating SMC requests to the secure monitor. These calls are intercepted transparently by the hypervisor, which then performs a VM context switch to the appropriate trusted OS VM. Currently, the trusted OS VM exclusively supports OP-TEE Trusted OS. Other trusted OS may necessitate direct support.



(a) Per-VM TEE support architecture.



(b) Per-VM TEE VM execution hierarchy.

Figure 9: CROSSCON Hypervisor per-VM TEE support.

Figure 9a illustrates the architecture. A configuration file distributes the resources over the GPOS and TOS VMs and identifies the VM types for correct internal module event binding. Figure 9b illustrates the runtime VM hierarchy, where each GPOS is assigned a TOS VM. In the illustrated scenarios the GPOS VMs run in different cores and are thus strongly isolated.

3.3 New Trusted Applications

This section covers the development of novel trusted applications, 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 trusted applications 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 trusted applications for deployment within our use cases to complement or enhance the security and privacy. This set of trusted applications 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 trusted application 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* [74, 75, 76, 77]. The Verifier executes the protocol to verify the identity of the Prover.

The standard protocol is depicted in Figure 10 and comprises two steps:

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 Pairs (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.

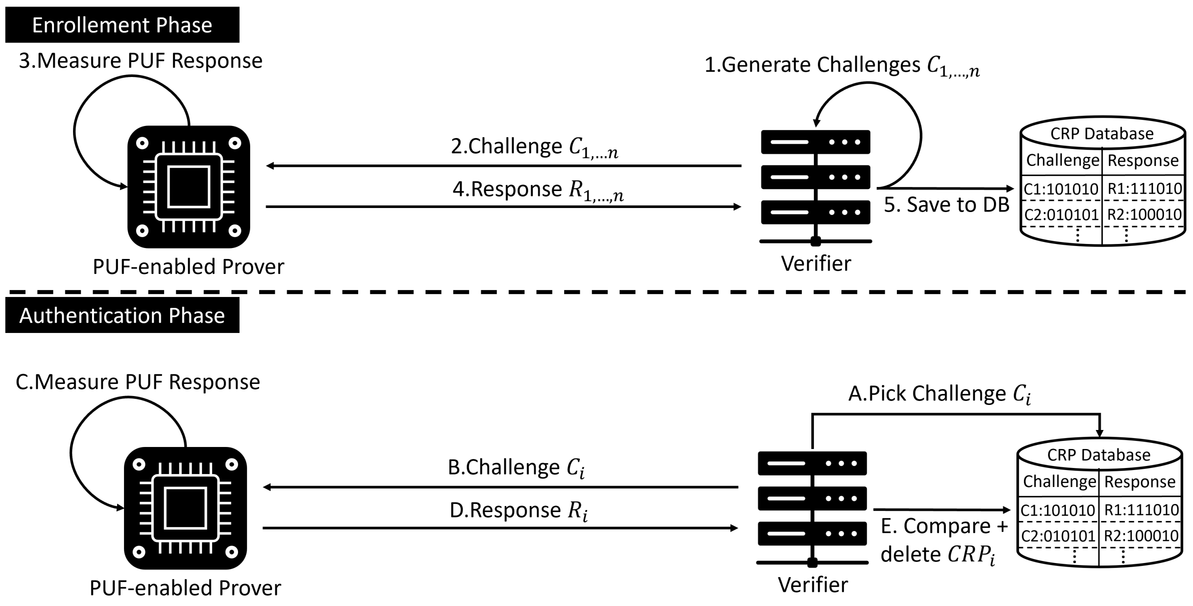


Figure 10: Standard PUF-based authentication protocol.

Device-to-server approaches are considered in works [78, 79, 80, 81, 82, 83], 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 [84, 85]. 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 [86, 87, 88, 89] 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 [77, 90, 91, 92, 74, 93, 94, 95, 96]. 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 [97, 98, 99, 100, 101], 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 [102, 103]. 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 [97, 98, 99, 100, 101], 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 [102, 103]. 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 Consequently, 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. Further, the scheme must be lightweight due to the inherent hardware limitations of IoT devices, particularly in terms of storage and computational power. The ambition is to craft an innovative solution that offers PUF-based device-to-device authentication for resource-constraint devices or within heterogeneous IoT networks.

This system aims to empower even low-end devices to effectively assume a dual role: acting as a prover while also verifying received authentication messages. The system is envisioned to be lightweight, streamlining both the authentication and verification processes to accommodate the diverse array of devices within the IoT ecosystem.

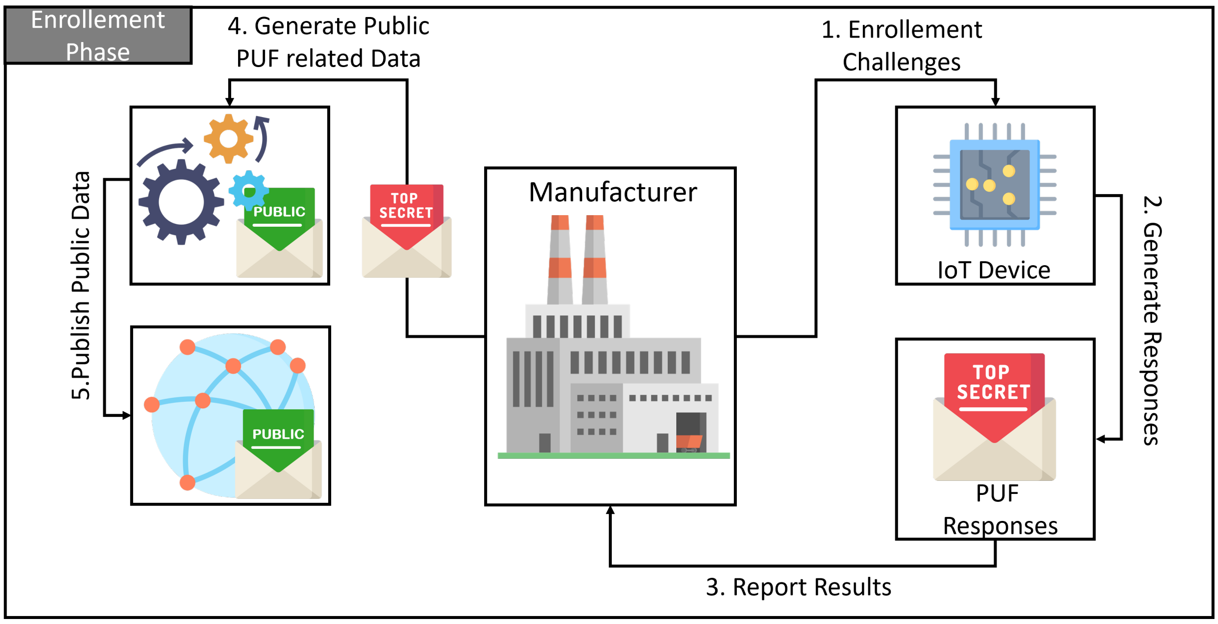
Approach The innovative aspect of this work lies in its approach to confidential PUF responses. Instead of securing these responses as Challenge-Response Pairs (CRPs) or through ML models, the scheme utilizes confidential data to generate public information. This public information, while correlated to the confidential PUF responses, is designed to reveal nothing about the actual responses, allowing it to be safely stored on a public platform, such as a bulletin board or a distributed ledger.

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.

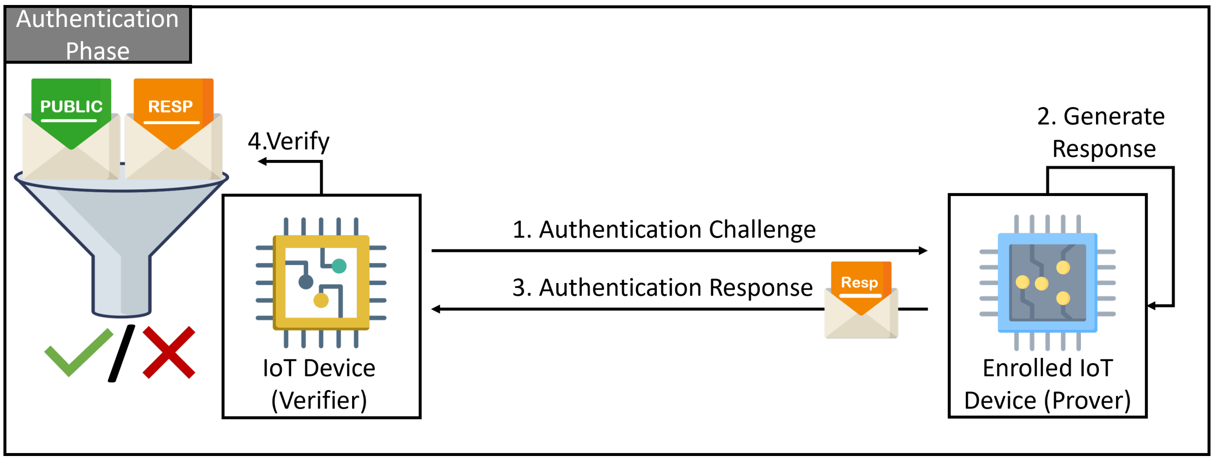
System model The system model of the proposed system incorporates three parties: the prover, the verifier, and the manufacturer, and it is divided into two distinct stages: the *Enrollment Phase* and the *Authentication Phase*, as depicted in Figure 11. While the manufacturer is trusted, its involvement 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 11a) 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



(a) Enrollment Phase



(b) Authentication Phase

Figure 11: System Design Overview

manufacturer to store confidential PUF-responses for later authentication, our approach incorporates two additional steps (steps 4 and 5 in Figure 11a. 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 11b, 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.

3.3.1.1 Innovation Aspects

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

- **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 solutions like 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 11 was realized in three different novel and distinctive methods that will be outlined in the following.

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.

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.

We view this time-synchronization as a promising new trusted service for CROSSCON, with potential benefits extending to various scenarios. Currently, this service is in its nascent stages of development. We are exploring several potential approaches to implement it, including the use of a distributed heartbeat mechanism through threshold signatures [104] or distributed hash chains [105]. We plan to integrate the actual design of this service into a subsequent version of the deliverable.

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. While we seek to investigate potential resolutions to this challenge throughout CROSSCON, our research has expanded to explore a solution that leverages the effectiveness of hash functions without necessitating time synchronization. This pursuit has resulted in the development of a third solution, dubbed **PAWOS: PUF-based Authentication with One-time Signatures**. 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.

3.3.2 Context-based Authentication

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

Background on Context-based Authentication 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 Wireless Fidelity (Wi-Fi) protocol as it is a well established protocol for IoT devices [106]. 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 [107].

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 [108].

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) [108]. 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 [109][110][111][112].

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 [108].

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 [113]. 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 [113]. 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 [114]:

- **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.

Objectives of the Context-based Authentication 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.

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 [115]. 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.
Advantageously, Siamese networks are capable of learning the differentiating characteristics even with a limited amount of available samples, which is beneficial for our approach as our service could be relatively quickly deployed in new locations.

Our Approach to Context-based Authentication Our approach of providing authentication to be location-bound is to utilize the trusted application 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 and trains the network to assess the similarity of fingerprints between enrollment phase and verification phase.

3.3.3 Remote Attestation

3.3.3.1 Background

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).

3.3.3.2 Objectives of Remote Attestation

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) [116, 117]. Therefore, our Attestation framework finds its application in attesting vulnerable/security-relevant application within CROSSCON stack.

Classic Control-Flow Attestation (CFA) schemes assume to have access to the complete Control-Flow Graph (CFG), however, it is not always possible to reconstruct a Complete-CFG, as it falls under the family of NP-hard problems [118]. In real-world, CCFGs can only be approximated [119, 120, 121, 122, 123, 124]. Even though a CFG can be approximated, approaches are still far from generating near-to-complete CFGs. Specifically, Rimsa *et al.* [125] showed that static and dynamic analyses can be combined to create more comprehensive CFGs. However, they can only approximate up to 46% of SPEC CPU2017[1].

We overcome the limitations of existing CFA schemes by leveraging Unsupervised Graph Neural Networks (GNNs) to identify deviations from benign executions. The core intuition behind our appraoch is to exploit the correspondence between execution trace, execution graph, and execution embeddings to eliminate the unrealistic requirement of having access to a complete CFG.

3.3.3.3 Our Approach

Within CROSSCON, our goal is to implement a Graph-Neural-Network (GNN) based ML approach for the verification of collected execution traces by the prover.

The system operates in two distinct phases:

- Training Phase: In this initial phase, benign traces are gathered for a particular application and its specific version. Within the CROSSCON framework, a TEE (e.g., TrustZone) is responsible for collecting these traces, storing and sending. The verifier gains access to this traces. The graphs of the traces are constructed through a feature extraction and preprocessing step. Subsequently, a ML model is trained using this dataset of benign traces.
- Detection Phase: The second phase is the operational stage, which commences when an untrusted entity collects traces, digitally signs them using the TEE, and transmits them to the verifier via a secure channel. Upon receipt, the trusted verifier checks the signature; if it is valid, the traces are input into the trained model. If not, the traces are discarded, and an error message is sent back. When valid traces are processed by the model, the resulting prediction is communicated back to the prover using the same secure channel.

Training and Inference The objective is to discern between benign and malicious partial Control-Flow Graphs (CFGs) from execution traces. The appearance of malicious data can vary greatly depending on

[1]https://www.spec.org/cpu2017/

the attack type (e.g. ROP/DOP). Therefore, it is advantageous to train a model exclusively on benign data, enabling it to recognize the inherent patterns of such data. Consequently, the model can determine whether a specific trace appears benign. Malicious traces, on the other hand, are utilized solely for evaluation purposes.

As software complexity increases, learning benign behavior becomes more challenging, which in turn makes the detection of benign traces more difficult. This issue can be mitigated by including a larger number of benign traces in the training dataset.

The inference process necessitates receiving trusted data from an untrusted source. To address this critical issue, a TEE can be employed. The TEE ensures the security and integrity of data: it gathers memory traces that require verification and assigns them a unique cryptographic signature. After this preparatory phase, the data must be sent through an encrypted channel to the verifier.

Upon receiving messages from an untrusted source (prover), the trusted party (verifier) must authenticate the signature. If deemed valid, the verifier proceeds to the preprocessing phase to extract sequences from the received execution traces. Following these steps, the data is input into the trained ML model to estimate the likelihood of the data being benign or malicious. The results are then relayed back to the requesting party. Upon reviewing the returned analysis, the untrusted party may decide to implement measures to thwart malicious activities.

Attestation The verifier generates an attestation report in accordance with the requirements. This report is digitally signed using the verifier's key. The report is then disseminated, enabling other participants to access and validate it. The attestation report encompasses various elements, including identifiers to ensure freshness and a digital signature to safeguard against forgery (Signature), and the attestation outcome (Result).

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, a.k.a, bitstreams, throughout provisioning and configuration, thereby preventing unauthorised access to the user's code or data. This assurance instils 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. Two primary services are essential to realising secure FPGA provisioning: the secure FPGA configuration service and the secure FPGA configuration/bitstream scanning service.

3.3.4.1 Secure FPGA Configuration Service

This service aims to securely deliver and configure the user's task onto the FPGA. 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, as shown in Figure 12.

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. This approach involves partitioning the FPGA into a static region, the region that cannot be reconfigured at runtime, and one or more partially reconfigurable regions that can be reconfigured at runtime.

The static region is leveraged to implement the FPGA shell, a trusted hardware circuit within the FPGA

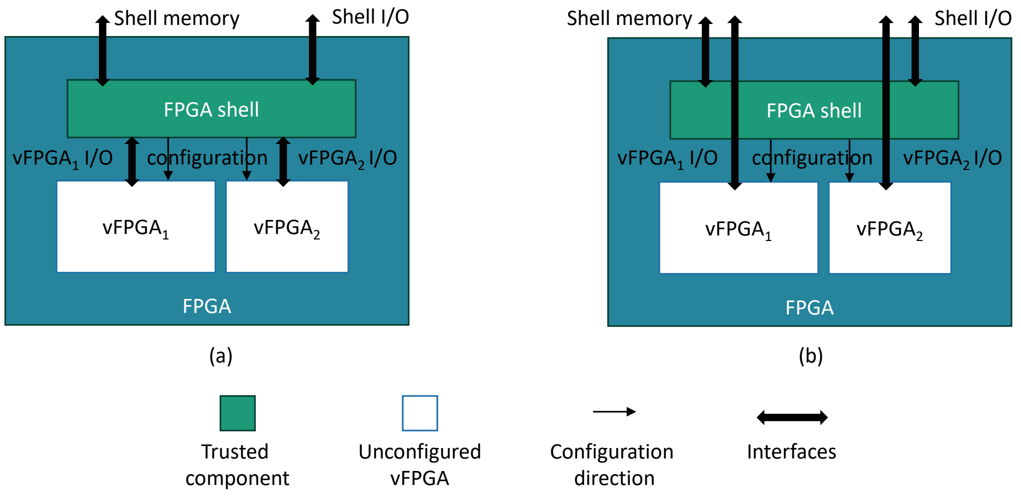


Figure 12: vFPGAs interfacing options.

fabric that provides vFPGAs with essential clocking resources and external interfaces. Moreover, it ensures the security of the user's IP by preventing unauthorised access to the FPGA's configuration memory through external ports. Additionally, the shell can be designed to configure vFPGAs internally by utilising the internal configuration port of the FPGA configuration engine. A partial bitstream is sent to the FPGA shell by loading it to a pre-defined memory region dedicated to the FPGA shell; see Figure 12. The partial bitstream is then read by the FPGA shell and forwarded to the internal configuration port of the FPGA configuration engine.

Note that access to the FPGA shell and its memory region is restricted to CROSSCON Hypervisor or to a dedicated TA /VM reserved for managing FPGA resources, which we refer to as TA_{FPGA}, see Figure 13. TA_{FPGA} can take over the decryption and verification of partial bitstreams before their configuration. Otherwise, the FPGA shell shall perform this process. Both TA_{FPGA} and FPGA shell can be attested.

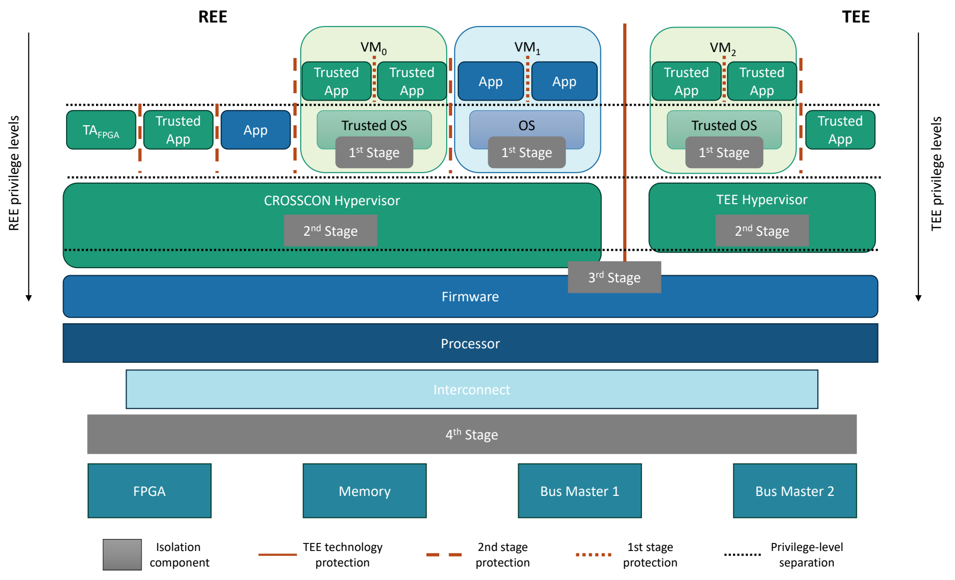


Figure 13: Refined CROSSCON Stack with TA_{FPGA}.

Each vFPGA can be managed and configured separately. Furthermore, each vFPGA has a pre-defined I/O interface with a distinct memory address range for communication. Depending on the architecture of the FPGA device, vFPGAs can be accessed through shared or dedicated I/O interfaces. For example, the vFPGAs share the shell I/O interface, where the shell and vFPGAs have non-overlapping address ranges and are connected to the same bus master through an FPGA-internal bus interconnect (see Figure 12a), or each vFPGA can be mapped to a different bus master (see Figure 12b).

Users 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. Assuming a trusted application or VM is requesting to run a task on the FPGA, the following steps will be taken:

1. vFPGA allocation: The process is initiated with a request from TA/VM to execute a task on the FPGA, specifying the location, count and sizes[2] of the task's partial bitstreams (i.e., FPGA configuration files) stored on disk. Given that TA_{FPGA} is available, vFPGA allocation requests are directed to TA_{FPGA}. Subsequently, the CROSSCON Hypervisor/TA_{FPGA} verifies the availability of a vFPGA and its corresponding partial bitstream. If both are available, CROSSCON Hypervisor/TA_{FPGA} assigns the vFPGA and provides the requesting TA with the FPGA shell/TA_{FPGA} public key.
2. Secret key passing: Following vFPGA allocation, the TA uses the public key to encrypt the secret key used for encrypting and signing the task's partial bitstream. The encrypted secret key is then sent to CROSSCON Hypervisor/TA_{FPGA}. If decryption and verification occur in the TA_{FPGA}, the TA_{FPGA} loads the partial bitstream in the shared memory with the FPGA shell and instructs it to start the configuration process. Otherwise, the encrypted secret key and the encrypted partial bitstream are loaded on the shared memory region for the FPGA shell to decrypt and configure.
3. vFPGA configuration: The FPGA shell configures the partial bitstream on the allocated vFPGA. Upon successful configuration, the FPGA shell confirms the completion of the process.
4. vFPGA access control: Access control rules for the accelerator can be set using the requesting TA/VM ID. Consequently, the task is ready for execution. Note that access control to accelerators on vFPGAs can be achieved through (i) TA_{FPGA} when communication between the allocated vFPGA and the requested TA/VM occurs through TA_{FPGA} or (ii) isolation components. In the future, we will investigate how to support vFPGA access control through the perimeter guard (PG) in CROSSCON SoC.
5. vFPGA deallocation: Since FPGAs do not support pre-emption, it is left to the system administrator to decide whether a vFPGA is allocated for a specific task until the task finishes or for a time slot (fixed or variable). Consequently, the TA must be able to terminate a task by sending a request to CROSSCON Hypervisor/TA_{FPGA}. CROSSCON Hypervisor/TA_{FPGA} can also send a termination request to the TA to allow it to copy or delete sensitive data before terminating the task on the vFPGA. This ensures efficient resource utilisation and effective management of FPGA resources based on task priorities and system requirements. Note that upon vFPGA deallocation, a special partial bitstream is configured to erase the previous configuration.

3.3.4.2 Secure FPGA Bitstream Scanning Service

This service aims to ensure that hardware designs to be configured on the FPGA are free from malicious circuits before being deployed on the FPGA device while upholding the privacy of the bitstream. FPGA designs can be complex and may involve multiple contributors, making them susceptible to malicious insertions. CROSSCON leverages existing scanning and vetting techniques to examine the FPGA bitstream and detect potential anomalies or malicious components that could compromise the system's security. During the scanning process, the service validates that the bitstream adheres to the intended design specifications, i.e., ensuring that it does not configure or overwrite neighbouring vFPGA resources and that no malicious circuits are included. Any discrepancies or unauthorised alterations are flagged. Once

[2]Partial bitstream size depends on the size of the targeted vFPGA; therefore, one or more sizes can be passed in the request

the scanning process is complete, a detailed report is provided to the system administrator, outlining the analysis results. The report includes information about the integrity of the IP design and any detected anomalies. To ensure the user's IP protection, a dedicated TA performs the inspection to protect the bitstream against unauthorised access. Similarly, the encrypted secret key and the task's encrypted partial bitstream(s) are passed to the inspecting TA.

3.3.5 Behavioral-Based Trusted Service

Behaviour-based security is a method in which all device's relevant activities are monitored to identify any deviations from normal behavioural patterns [126]. Given the scope of the project on the security of connected devices, so the scope of 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) [127] have shown significant advances and improvements in terms of algorithms and scalability to constrained environments [128]. As such, TinyML is expected to play an important role in security of computing environments at the edge of IoT networks [129].

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.

We note that traditional intrusion detection systems are in majority signature-oriented where the software monitors network traffic and compares the traffic (i.e., the packet data) to known signatures of known threats [130]. The anomaly detection service operates differently - it also monitors network traffic streams, but it compares the network streams (also known as network flows) to a baseline of normal behaviour and looks for anomalies [130].

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 [131].

3.3.5.1 Innovation Aspects

There are two main innovation aspects to be addressed:

- *Network telemetry* suitable for IoT protocol behavioural analysis. Ensure necessary visibility of lower IoT network protocols and access to network traffic of a device. Furthermore, the service will offer 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]https://www.tensorflow.org/lite
[4]https://www.tensorflow.org/lite/performance/model_optimization

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 such anomaly, i.e. what aspects of the network traffic are suspicious and significant for the decision of anomaly. The explainability aspect 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.

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

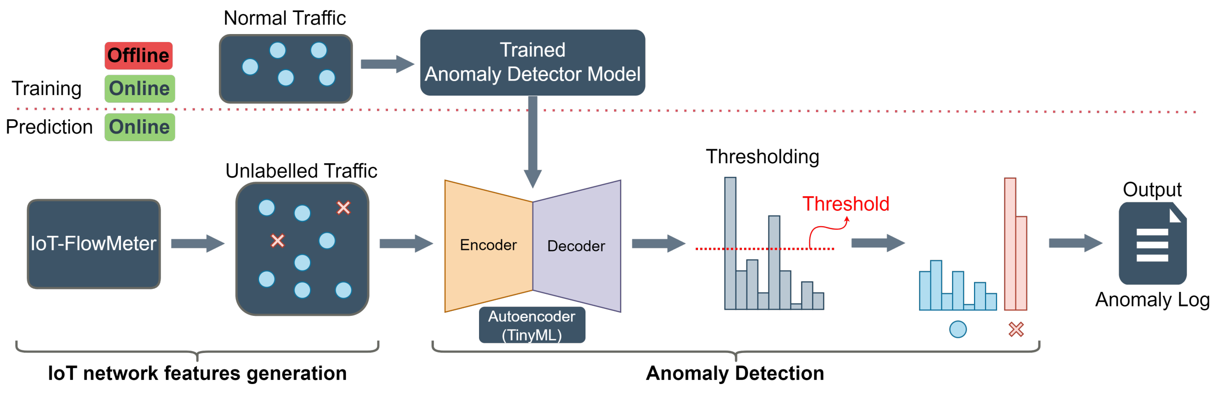


Figure 14: Behavioral-based trusted service workflow.

Figure 14 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 either online or offline. Once the model is trained, the tool switches to monitoring or inference mode.

The first module is 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

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

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, what features have been the most critical for detection, and gives evidence showing the deviation of the anomaly from the normal training data.

3.3.6 Control Flow Integrity Trusted Service

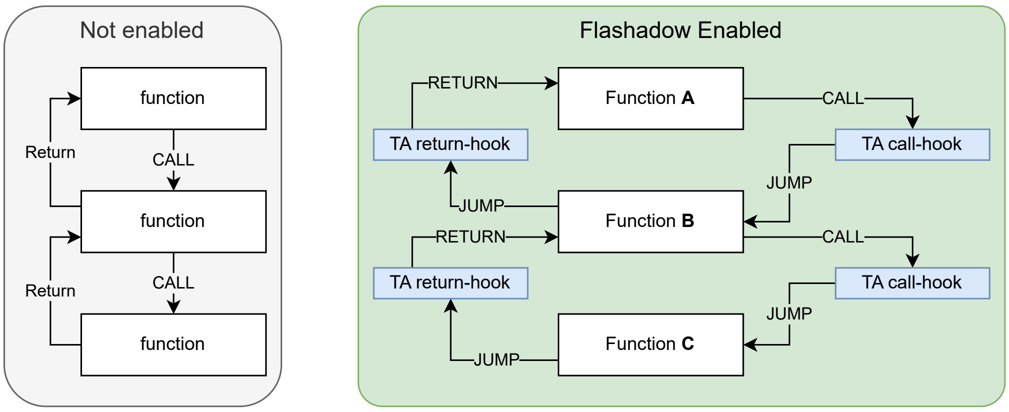


Figure 15: Difference between an unsecure application and an application using Flashadow.

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, reason for which several chip manufacturer are working to create hardware solutions that can offer such a guarantee. Unfortunately, relying on hardware is not always on option, especially in the context of IoT that counts a plethora of different devices lacking such a specific hardware. With CROSSCON we propose to address this security gap by leveraging the TEE rather than specific hardware. TEEs often enable trusted services: distinct security-oriented applications that offer some security service to untrusted clients. However, these TAs are independent, self contained, and passive. A passive service cannot properly enforce control flow integrity for 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. Our contribution is two-folds: we propose Flashadow for the non-MPU bare-metal TEE (for Class 0 devices), and uIPS for the MPU bare-metal TEE (for Class 1 devices).

Flashadow 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. Combined, 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 our 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 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.

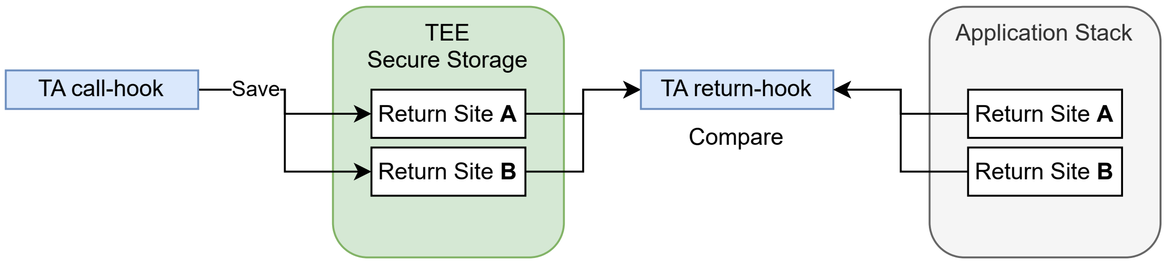


Figure 16: The operations performed in the two Flashadow hooks.

We propose a second design, uIPS, for our MPU-enabled bare-metal TEE. Contrarily to Flashadow, 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 TEE. As for the forward-edges protection, uIPS provides and 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 legit 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

3.4.1 Existing IoT Update Mechanisms and Standards

There is no consensus on a standard for firmware updates in the IoT world [132]. 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 [133].

Broadly speaking, the possible firmware update mechanisms fall in two categories: (1) *wired* updates (e.g., via a USB port), or (2) *wireless/Over-the-Air (OTA)* updates. Moreover, the update can be either *full* or *partial* (e.g., only a piece of the firmware or the OS), it may include signature and integrity verification and it may require an attestation of delivery and installation.

In the following we shall focus on the Firmware OTA (FOTA) update process, which is the most relevant for the CROSSCON project. We first briefly describe the main proposals available in the literature, and then consider in detail the Software Updates for Internet of Things (SUIT) draft standard proposed by the IETF.

3.4.1.1 Previous Work on FOTA Update

Perito and Tsudik [134] propose a Secure Code Update By Attestation (SCUBA) framework for sensor networks which can be used to identify and repair a compromised sensor through firmware updates. However, the (software-based) attestation technique heavily relies on the timing characteristics of the measurement process and the use of an optimal checksum function; due to these assumptions, SCUBA is not a suitable approach for IoT settings.

The Update Framework (TUF) introduced in [135] provides a generic specification for FOTA updates that developers can adopt in any software update system. However, this approach does not seem adequate for resource-constrained IoT devices.

In [136] the authors propose a secure FOTA update scheme for automotive Electronic Control Units. Their solution employs separation of entities (e.g., OEM, Apps provider) and verification of the system's integrity and authenticity, together with firmware versioning and entitlements for each vehicle and its corresponding electronic control units.

In [137] the authors present a FOTA update system providing a low-power mesh protocol, route discovery, and establishment. The solution is based on the Lightweight Mesh network protocol (LWMesh) over a peer-to-peer mesh architecture (P2PMesh). However, in addition to the proprietary aspect of LWMesh, the proposal misses the security aspects.

Doddapaneni et al. [138] propose a FOTA update scheme for IoT devices by defining a new secure object called Firmware Object Signing and Encryption (FOSE). The object is encoded using a secure format such as JSON Object Signing and Encryption (JOSE). The main goal is to solve the problems of integrity (in case of network loss or break) and security (by encrypting the payload). The paper also proposes a procedure for over-the-air updates. However, it addresses only the case where there is an intermediary application between the end-device and the device manager.

In [139] the authors design an architecture (ASSURED) for a secure update framework of realistic embedded devices. To demonstrate its feasibility, ASSURED is instantiated and evaluated on two commodity hardware platforms, HYDRA and Arm TrustZone-M. The proposal is an enhancement of the TUF architecture, and adopts the same terminology proposed by the SUIT Working Group. However, the authors did not take into account the scalability issues when dealing with hundreds of IoT devices (for instance to guarantee the attestation of update installation) and the case of very constrained devices.

The UpKit proposal [140] is a portable and lightweight software update framework for constrained IoT devices that aims to encompass all phases of the update process, from generation to installation.

Among industry solutions, Cloud IoT Core [141] is a Cloud-based product provided by Google which allows a secure device connection and management from a few to millions of IoT devices. Upon a configuration update from Cloud IoT Core, an update from another external source such as a FOTA update from the manufacturer, the device state update in the platform is triggered.

AWS IoT device management [142] is a solution provided by Amazon to register, organize, monitor and remotely manage IoT devices. Among the multiple functionalities it has, the solution provides means to query the states of the managed IoT devices and to send FOTA update, mainly for FreeRTOS devices.

OTA Download (OAD) [143] is an ecosystem provided by Texas Instruments that allows the update of the firmware image running on BLE devices wirelessly.

Mbed OS is an open-source OS for platforms using Arm microcontrollers designed specifically for IoT devices. In the newest version of Mbed OS (currently at v5.15), Pelion Device Management [144] is used to interact and manage the connected devices that implement Mbed OS, in a simple, flexible and secure way. Among its services, it enables the provisioning and connection of IoT end nodes with cost-effective, and provides secure and reliable software updates from remote locations OTA, ensuring, hence, a long product lifetime.

3.4.1.2 The SUIT Proposed Standard

The IETF SUIT working group is developing an internet standard, described in RFC 9019 [145], proposing a new firmware update architecture that uses state-of-the-art security and can be deployed even on very constrained IoT devices (Class 1, as defined in RFC 7228 [146]). The architecture is based on a metadata structure, known as the *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 can be flexible enough to be applied in many situations, but at the same time can be made robust with regard to functionality and security.

The following informal requirements for the update process have been considered by the SUIT working group:

Authentication: new firmware images must be accepted only from the authenticated sources; this is usually achieved by public key encryption.

Version Control: only the newest and freshly generated updates must be accepted, and only if they are intended for the particular device.

Integrity: the device must check that the firmware was not tampered with or changed during the update process.

Reduced users interaction: user involvement should be minimized, as the user is a possible source of errors and may not be expert enough to perform complex operations. However, their consent must be taken into account.

Platform and OS independent: the update process must be generic enough to work in most of the well-known hardware and OSs.

Scalability: the process must be able to handle the ever-increasing number of IoT devices efficiently.

An information model for the SUIT manifest has been described in RFC 9124 [147], and a proposal for a concrete encoding scheme is currently in the draft stage [148]; it is based on the CBOR binary serialization format (described in RFC 8949) and the associated 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.

Let us briefly describe the main architectural points of the SUIT standard. We first define the various entities participating in the firmware update process.

- ▶ The **Author** is responsible for creating a new firmware, uploading it to the distribution server and notifying the device management platform.
- ▶ The **Device Operator** is responsible for the day-to-day operation of the IoT devices and can approve a new firmware, triggering the update using the status tracker.
- ▶ The **Network Operator** is responsible for the operation of the network to which the IoT devices are connected, and can also interact with the status tracker.
- ▶ The **Trust Provisioning Authority (TPA)** is responsible for the distribution of trust anchors and authorization policies. Usually the original equipment manufacturer (OEM) will perform the duty of the

TPA, but this can change in more complex scenarios.

- The **User** is the end user of the device.

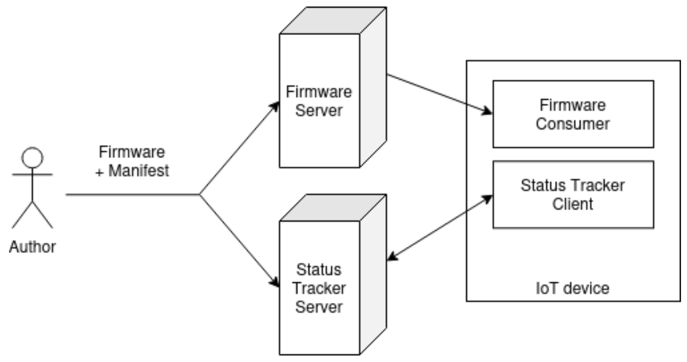


Figure 17: The SUIT secure update architecture.

The architecture of the SUIT update process is schematically depicted in Figure 17 and involves four main components:

- an IoT device, or more specifically, a component inside the device that needs to receive the firmware update, called the **Firmware Consumer**;
- a **Firmware Server** that can store manifests and firmware images and make them available upon request;
- a **Status Tracker Server** that keeps track of software and hardware information about each device on the network and the availability of updates;
- a **Status Tracker Client** 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 workflow of the device.

Once the update is initiated, the Firmware Consumer receives the manifest and must validate the signature to assert its authenticity. The standard does not cover how the signature is performed and checked, and it assumes a trust anchor is already present in the device. 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.

3.4.1.3 Software Bill Of Materials

In the manufacturing industry the *Bill of Materials (BOM)* is a complete inventory detailing all the items that are used in the manufacturing process of a product. In the automotive industry, for example, manufacturers maintain a detailed BOM for each vehicle, listing both the components built by them and those provided by third-party suppliers. When a defective part is discovered, the manufacturer knows precisely which vehicles are affected and can notify vehicle owners of the need for repair or replacement.

The same idea can be transposed to the digital world and applied to software, as proposed for instance in the US government Executive Order 14028 on improving cybersecurity standards [149], which has been issued in May 2021 as a fallout of the SolarWinds attack.

A *Software BOM (SBOM)* is defined as a list of all the open source and third-party components present in a codebase. An SBOM also lists the licenses that govern those components, the versions of the components used in the codebase, and their patch status, which allows security teams to quickly identify any associated security or license risks. The value of the SBOM is twofold: it increases the traceability of the software components (and thus its security), and it provides information for a more accurate estimation of the cyber risk.

There are three main standards currently in use for specifying a SBOM: Software Package Data eXchange (SPDX) (ISO/IEC standard), Software Identification (SWID) (older ISO/IEC standard) and CycloneDX (OWASP standard).

- SPDX is an open standard promoted by ISO and the Linux Foundation. Its specification defines a standard data format for communicating the component and metadata information associated with software packages. There is also a more lightweight profile called SPDX Lite. It only uses a subset of the full specification, making it more suitable for use in IoT.
- A SWID tag document is composed of a structured set of data elements that identify the software product, characterize the product's version, the organizations and individuals that had a role in the production and distribution of the product, information about the artifacts that comprise a software product, relationships between software products, and other descriptive metadata.
- CycloneDX is a modern standard for BOM, not restricted to software, used both in IT and OT. It is supported by Lockheed Martin, ServiceNow, IBM, Contrast Security, Sonatype, and many others. The Hardware BOM (HBOM), supports documentation of components, devices, firmware, configurations, and many other fields. It is ideal for IoT devices.

3.4.2 Requirements for Integration with DevSecOps Platforms

3.4.2.1 Introduction

DevSecOps, short for Development, Security, and Operations, represents a contemporary framework that seamlessly integrates security practices into every phase of the software development lifecycle[6]. This method aims to significantly reduce the risks of deploying software with security vulnerabilities by prioritizing collaboration, automation, and well-defined processes. Unlike traditional models where security was often an afterthought, DevSecOps ensures that security responsibilities are evenly distributed across the team.

This approach is an evolution of the DevOps model, which was developed to overcome the challenges of the traditional linear and segmented software development process. These older methods often resulted in delays and late-stage security issues due to their separate phases for planning, design, development, integration, and testing. DevOps improved this by advocating for rapid delivery of smaller, high-quality code updates through constant collaboration between development and operations teams. However, DevOps frequently relegated security to a final step, not integrating it throughout the process.

DevSecOps enhances the DevOps model by embedding security practices into every aspect of the development process. It is guided by the 'shift left' security principle, which involves incorporating security considerations from the onset of a project. This ensures that security is a continuous, shared responsibility throughout the development cycle, leading to safer and more secure software solutions.

[6]https://aws.amazon.com/what-is/devsecops/

3.4.2.2 Key Components of DevSecOps

- Continuous Integration: Integrating regular code contributions into a central repository to identify issues early, complemented by tools for tracking, managing, and reporting on software or requirement changes.
- Continuous Delivery: Automating the progression of code from the build stage to staging environments, ensuring that the software remains in a ready-to-deploy state.
- Continuous Security: Embedding security throughout the entire development lifecycle from the planning stage, incorporating automated security checks at the code commit phase, and continuous testing, including security assessments during build and test phases, penetration testing, and continuous monitoring.
- Communication and Collaboration: Promoting effective teamwork and ongoing communication to address code conflicts and align objectives, while integrating security practices early in the planning and development stages.
- Security Training for Developers: Educating software developers in security best practices to elevate their security awareness and skills, reinforcing the team's overall security posture.

3.4.2.3 Common DevSecOps Tools

Here we introduce the most common classes of DevSecOps tools [7],[8].

- Static Application Security Testing (SAST): These tools analyze proprietary source code to identify vulnerabilities.
- Software Composition Analysis (SCA): SCA automates the visibility into open-source software use for risk management, security, and license compliance.
- Interactive Application Security Testing (IAST): IAST tools assess potential vulnerabilities in production environments, using special security monitors that run within the application.
- Dynamic Application Security Testing (DAST): DAST tools simulate hacker attacks by testing the application's security from outside the network.

3.4.2.4 Requirements

In developing a toolchain for embedded systems, which includes tasks like creating firmware updates and secure cross-compilation, it's important to integrate DevSecOps practices. CROSSCON toolchain is designed to meet the specific needs of embedded systems, including managing firmware updates and ensuring the security of the compilation process. It's capable of handling the challenges of embedded systems, such as limited resources, the need for high security, and cross-compiling for different hardware architectures. The CROSSCON toolchain also supports managing a testbed of devices like Raspberry Pi and FPGAs, with features like remote power control.

The following requirements for the DevSecOps platforms have been established to ensure they align well with the unique aspects of the CROSSCON toolchain:

- *Source Code Management and Version Control*: Support for robust version control systems (like Git) to manage firmware and toolchain code.
- *Continuous Integration / Continuous Development (CI/CD)*: Ability to automate the build, testing, and deployment processes of firmware and toolchain updates.

[7] https://aws.amazon.com/what-is/devsecops/

[8] https://www.microsoft.com/en-us/security/business/security-101/what-is-devsecops

- *Automated Testing Integration*: Integration with automated testing frameworks and the ability to trigger tests on remote devices.
- *Remote Device Access and Control*: Capability to remotely update firmware, power on/off devices, and control testbed hardware.
- *Monitoring and Logging*: Tools for monitoring the health and performance of the testbed devices and logging activities.
- *Scalability and Flexibility*: Ability to scale with the addition of more devices and adapt to different types of embedded hardware.
- *Integration with External Tools*: Ability to integrate with additional tools or services as needed (e.g., for specialized FPGA programming).

So, the CROSSCON toolchain needs a DevSecOps platform, which is robust, flexible, and capable of handling the specific demands of embedded systems development and testing. It should integrate seamlessly with hardware-specific tools and provide secure, automated, and efficient workflows to manage the entire lifecycle of firmware development, from coding to deployment and testing on actual hardware.

Comparison of existing platforms

In this section, we will conduct a comprehensive comparison of existing DevSecOps platforms, evaluating their features, capabilities, and overall effectiveness. The aim is to provide a clear understanding of how these platforms differ and which might best suit the need of CROSSCON toolchain.

Jenkins[9][10]

Jenkins is an open-source automation server noted for its extensive plugin ecosystem, which aids in automating complex CI/CD pipelines in various environments. Its open-source status allows for a high level of customization, enabling integration with many tools and technologies. However, Jenkins can be challenging in terms of setup and maintenance, being resource-intensive and having an interface that might be less intuitive than newer CI/CD tools. It often requires considerable effort in configuration and ongoing management, presenting a learning curve for teams new to the platform.

GitHub Actions[11]

GitHub Actions, a CI/CD service, is integrated within the GitHub ecosystem, designed for projects hosted on GitHub. It offers a user-friendly setup and deep integration with GitHub's repository and workflow features, enabling easy automation of build, test, and deployment workflows. This service is especially convenient for teams familiar with GitHub. However, its utility is primarily limited to GitHub-hosted repositories, which may not suit teams using multiple version control platforms. While GitHub Actions is user-friendly, it might lack some advanced CI/CD features and customizations that are available in more specialized tools.

GitLab CI/CD[12]

GitLab CI/CD, integral to the GitLab ecosystem, offers an all-in-one solution for software development, combining repository management with CI/CD functionalities. Its unified structure streamlines the development process, aiding teams in adopting and maintaining CI/CD practices more efficiently. This tool is particularly effective for projects seeking a streamlined pipeline within a single service. However, its integrated environment may reduce flexibility, and being resource-intensive, it might pose challenges for smaller projects or teams.

[9] https://hackr.io/blog/what-is-jenkins
[10] https://phoenixnap.com/kb/what-is-jenkins
[11] https://docs.github.com/en/actions
[12] https://docs.gitlab.com/ee/ci

CircleCI [13]

CircleCI, a cloud-based CI/CD tool, is known for its efficiency and speed, enabled by advanced caching and parallel execution features. It supports various programming languages and integrates well with multiple version control systems. CircleCI is particularly effective in delivering fast build times and offers significant customizability, suitable for projects with complex workflows. However, setting up CircleCI can be intricate for detailed pipelines, and its pricing, particularly for large-scale use, may be a factor for budget-conscious teams.

Travis CI[14]

Travis CI is a cloud-based CI/CD service recognized for its simplicity and seamless integration with GitHub. It provides automated testing and deployment, making it popular among open-source projects. Its straightforward configuration is ideal for smaller projects or teams seeking an uncomplicated CI/CD solution. However, Travis CI may lack the customization and advanced features found in more robust platforms like Jenkins or GitLab CI/CD. Critiques of Travis CI often focus on its build times and resource allocation, especially when compared to other CI/CD services.

In the following we analyze the above platforms against the specific requirements for developing an embedded system toolchain with a testbed of Raspberry Pi devices and FPGAs. The analysis focuses on how well each platform aligns with the requirements such as CI/CD, security, remote device management, and support for embedded systems.

Jenkins

Source Code Management integrates well with version control systems like Git, enabling efficient tracking and coordination of changes in the code base. This support dovetails nicely into the system's CI/CD capabilities. Primarily, the highly customizable CI/CD pipelines are excellent for complex workflows, offering a robust mechanism for integrating changes into the project.

Additionally, the platform provides secure cross-compilation. It can integrate with cross-compilation tools through plugins, creating a seamless interfacing between different systems. Notably, it also supports various testing frameworks enhancing its capability for automated testing.

When it comes to remote device control, the flexibility to extend with plugins for remote control is available, although this might require custom development. Similarly, monitoring and logging capabilities are extensive. The monitoring setup can be achieved via plugins, ensuring the system health and activity are always visible and manageable.

The platform's scalability and flexibility are also mention-worthy. It is highly scalable and flexible, making it suitable for complex and evolving environments. Moreover, it showcases a high degree of integrability with external tools, allowing teams to use their preferred tools in conjunction with this one. In sum, these features combine to create a system that is comprehensive, adaptable, and effective in managing varying software development needs.

GitLab

With built-in robust version control, the Source Code Management tool has a strong integration with Git. This lends itself effectively to its integrated CI/CD pipelines, which are notable for being easier to set up than Jenkins. Coupled with this is the good support for automated testing within these CI/CD pipelines, streamlining the testing process.

Despite its extensive features, the Remote Device Control has limited native support, meaning it likely requires third-party tools or custom solutions for full functionality. However, the Security and Compli-

[13] https://circleci.com/docs/about-circleci/

[14] https://docs.travis-ci.com/user/for-beginners/

ance elements are comprehensive, featuring various security features and offering more out-of-the-box security integration than Jenkins.

While it provides reliable monitoring and logging, enhancements might be needed for specialized hardware monitoring. Despite being scalable, it's slightly less flexible than Jenkins in terms of the plugin ecosystem, potentially affecting customizability.

Finally, the Integration with External Tools supports various integrations, but the connectivity isn't as extensive as Jenkins. This could potentially limit which external tools you can seamlessly integrate with. Despite some limitations, the tool's strengths in source code management, CI/CD, and security measures prove it to be a reliable option for many development needs.

GitHub Actions

The Source Code Management system enjoys seamless integration with GitHub repositories, streamlining access and control over code versions within this popular platform. This seamless integration extends into the realm of CI/CD, where automated workflows are easily configured and controlled within the GitHub environment.

When it comes to secure cross-compilation, it's conveniently configurable; however, it highly depends on the available GitHub Actions. Automated Testing is another area where the platform shines, benefiting from an integrated approach with GitHub’s ecosystem to support and streamline testing procedures.

However, it has limited native capabilities in the area of Remote Device Control and would require custom setup or third-party actions for optimal functionality. For monitoring and logging, while there are basic capabilities present, in-depth analysis might require the use of external tools.

In terms of scalability and flexibility, the system is scalable within the GitHub framework but may present less flexibility outside it. Similarly, when it comes to the integration with external tools, the platform integrates well within the GitHub marketplace but is less open than Jenkins. This could potentially limit some functionality if you rely on a variety of external tools. Despite these minor limitations, the system excels at effectively leveraging the GitHub environment and could be a great choice if your team mostly operates within GitHub.

CircleCI

The Source Code Management feature of this system comfortably integrates with Version Control Systems (VCS) like GitHub and Bitbucket. Its CI/CD feature stands out by providing efficient and fast pipelines. While they may be less customizable than Jenkins', they fare better in terms of ease of setup.

The platform supports secure cross-compilation through configurable jobs. This enhances the diversity of tasks that can be performed within the system. Coupled with good support for automated testing, this makes the platform conducive for rigorous code validation and deployment pipelines.

On the flip side, Remote Device Control is not a native feature and would require custom scripts or integration. This could add a layer of complexity in setting up remote access capabilities. When it comes to Monitoring and Logging, the platform provides adequate features, although additional tools may be needed for specific hardware.

In terms of scalability and flexibility, the system delivers satisfactory performance. It is scalable and fairly flexible, but not to the extent of Jenkins. Finally, it has good integration capabilities with third-party tools, broadening the range of external resources that can be utilized alongside it. Despite a few missing features, the platform provides a fast, efficient, and user-friendly environment that caters to a variety of software development needs.

Travis CI

The Source Code Management system has good integration with GitHub, which facilitates effective code version control. The CI/CD pipelines are simple and easy to use, which can expedite the development process. Secure cross-compilation can be set up, but it might not be as straightforward as you'd expect, potentially requiring extra configuration steps.

Automated testing is supported by the system, but with less customization than available with Jenkins or GitLab. This means you might have less control over the nuances of test setup and execution. With regards to remote device control, the platform has limited native support and would require additional scripting or tool integration for fully-fledged operability.

Monitoring and logging capabilities are basic, and might require enhancement or additional tools for specialized use, suggesting it might not provide sufficient insights for some specific use cases. The platform is scalable within its own framework, but its flexibility is somewhat limited, making it less customizable than some alternatives.

Finally, while the platform supports integrations, its range is less extensive than Jenkins. This might limit the scope for using external tools alongside the platform. Despite a few limitations, however, the platform offers a user-friendly environment with good scalability and source code management features that can be an efficient choice for many development teams.

The next table represents the above result of the comparison of popular DevSecOps platforms based on our requirements:

Requirements	**Jenkins**	**GitLab**	**GitHub Actions**	**CircleCI**	**Travis CI**
Source Code Management	Excellent	Excellent	Excellent	Good	Good
CI/CD	Highly Customizable	Integrated, Easy Setup	Integrated, Easy for GitHub	Efficient, Fast	Simple, Easy
Cross-Compilation	Good with Plugins	Configurable	Configurable	Good	Possible but Limited
Automated Testing	Extensive	Good	Good	Good	Good
Remote Device Control	Possible with Plugins	Limited	Limited	Possible but Limited	Limited
Monitoring and Logging	Extensive	Adequate	Basic	Adequate	Basic
Scalability and Flexibility	Highly Scalable, Flexible	Scalable, Less Flexible	Scalable within GitHub	Scalable, Fairly Flexible	Scalable
Integration with External Tools	Excellent	Good	Good	Good	Good

For the CROSSCON toolchain, focused on firmware updates and secure cross-compilation with a Raspberry Pi and FPGA testbed, the chosen DevSecOps platform is Jenkins. This decision is primarily influenced by Jenkins' feature of high customizability and its extensive plugin ecosystem. These attributes make Jenkins particularly well-suited for complex and specialized workflows, which are typical in embedded systems development involving hardware like Raspberry Pis and FPGAs. The ability to tailor the tool to our specific needs, from firmware development to testing and deployment, offers a significant advantage. However, it's important to acknowledge that this choice comes with the trade-off of requiring more setup and maintenance effort. Jenkins' open-source nature and the need for manual configuration and management mean that we'll need to invest more resources in its initial setup and ongoing upkeep. Nonetheless, the flexibility and adaptability it provides align well with the unique demands of the CROSSCON project.

https://about.gitlab.com/solutions/open-source/

3.4.3 Literature Review on Secure Compilation and TA Cross-Compiler

A *secure compiler* is a compiler that performs source-to-target translation by preserving a given security policy. This goal is complementary to more traditional properties of compilation like correctness (preservation of behaviour from source code to target code), and has started to attract the researchers' interest only in recent times; see [150] for a survey of formal work in this field up to 2019.

Two main approaches to secure compilation exist in the literature, depending on whether the target language is typed or not.

In the first case, one may try to recast the security preservation property in terms of the type systems of the two languages considered (which are required to be sufficiently expressive). In a typical scenario, the source language is a high-level language in which security properties are either enforced implicitly (for instance by automatic memory management, scoping rules, etc.) or easily guaranteed by the well-typedness of a program, whereas the target language is either a typed intermediate language which offers some (different) form of security guarantees, or an untyped assembly language with no safeguards at all. In this second case the target language needs to be supplemented by an ad-hoc type system guaranteeing the low-level security properties of interest (e.g. non-interference between memory regions). In both cases, the ultimate goal is to prove that the compiler transforms well-typed source programs into well-typed target programs (subject reduction), thereby ensuring the preservation of the security properties under consideration. See [151, 152] for some seminal examples of this kind of approach.

In the second approach one tries to ensure security preservation by modifying the translation process in a suitable manner, for instance by inserting runtime checks in the generated code or by leveraging some existing protection mechanisms in the target architecture.

For the first case, the main approaches considered in the literature are control-flow integrity [153], which involves a rewriting of target code to enforce that no jumps can be made outside of the locations specified in the target control-flow graph, and dynamic check insertion (see e.g. [154], where defensive wrappers provide dynamic type checks for the untyped target-level code).

As regards target-level protection mechanisms, these can be provided either by the hardware or by an underlying OS. In the first class we can mention capability machines, whose most important example is the CHERI model [155], and Trusted Platform Module (TPM)s [156, 157] like for instance the Intel SGX platform or Arm TrustZone extensions; both these platforms have found some degree of support in currently-available processors. In the second class of protection mechanisms, a well-explored technique is Address Space Layout Randomization (ASLR) [158, 159], as provided for instance by OpenBSD, Linux and Android.

It is important to note that the two techniques outlined above address different attack modes. In type-oriented (static) approaches, all the software up to the linker is trusted; at linking time the attacker code is typechecked and rejected when ill-typed (i.e. insecure). In contrast, protection-oriented (dynamic) approaches generally consider a setting in which the TCB is much smaller, and the attacker potentially controls all the software stack except for the compiler and the linker. Insecure behaviour is then avoided by relying on the additional runtime checks.

3.4.4 Design of CROSSCON Secure Update

We base our design on the SUIT standard (Software Updates for IoT), as described in subsection 3.4.1.2, with two fundamental additions to increase the security of the overall process:

- the integration of a SBOM, in order to improve the transparency of the process and ease the tracking of dependencies and vulnerabilities;
- an extension of the SUIT manifest format, in order to accommodate new fields whose intended pur-

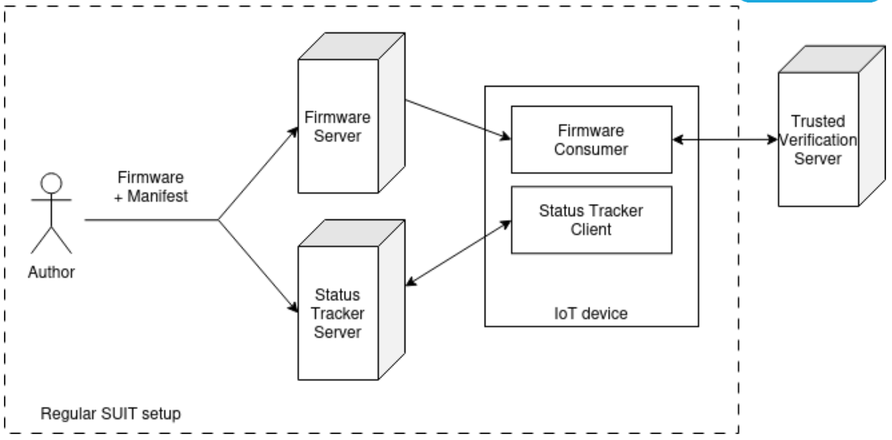


Figure 18: The SUIT secure update architecture, with CROSSCON additions.

pose is to host formal proofs of selected firmware security properties (*certification manifest*).

We supplement the high-level architecture of the SUIT mechanism described in Figure 17 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 18.

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

3.4.4.1 Update Generation

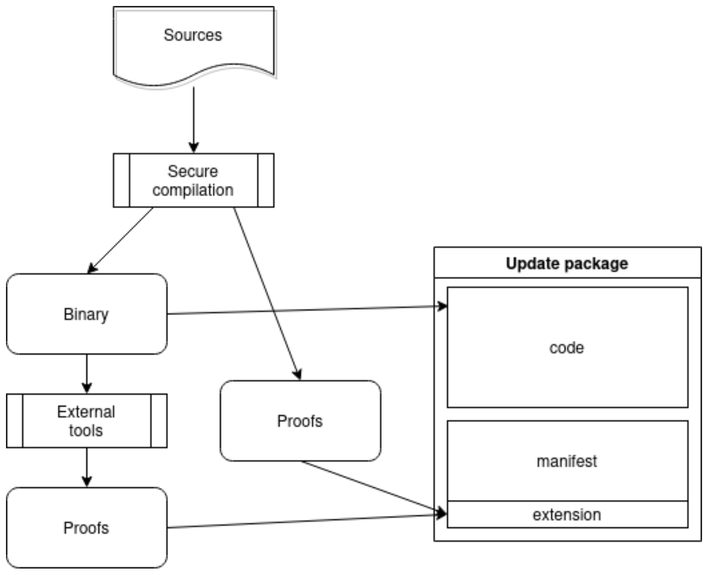


Figure 19: Workflow for update generation.

The workflow for the construction of the update package is summarized in Figure 19. The agent responsible for producing the firmware update (i.e. the *Author*, in the terminology of 3.4.1.2) starts by compiling the source code using the CROSSCON secure compiler; the outputs of this step are the compiled binary and a formal proof of the security properties guaranteed by the secure compilation process (see subsection 3.4.6 below).

At this stage the Author can run additional tools on the binary, extracting further security proofs that can be used by the secure update process to guarantee the preservation of some selected property of the updated component.

For instance, the security policy may require the new component to have the same control flow graph of the replaced component. In order to implement such a requirement, the Author must provide with each update a suitable representation Γ of the control flow graph (CFG) of the supplied code. At update installation time, the firmware consumer can then compare the graph Γ which represents the CFG of the current version of the component with the graph Γ representing the CFG of the updated version, and has a chance to reject the update if (for instance) the two graphs are not isomorphic.

Finally, the Author sends the resulting binary and the manifest (which includes all the formal proofs that have been generated in the previous steps) to the Firmware Server, in order to enable distribution of the update, and notifies the Status Tracker Server of the availability of the update.

3.4.4.2 Update Installation

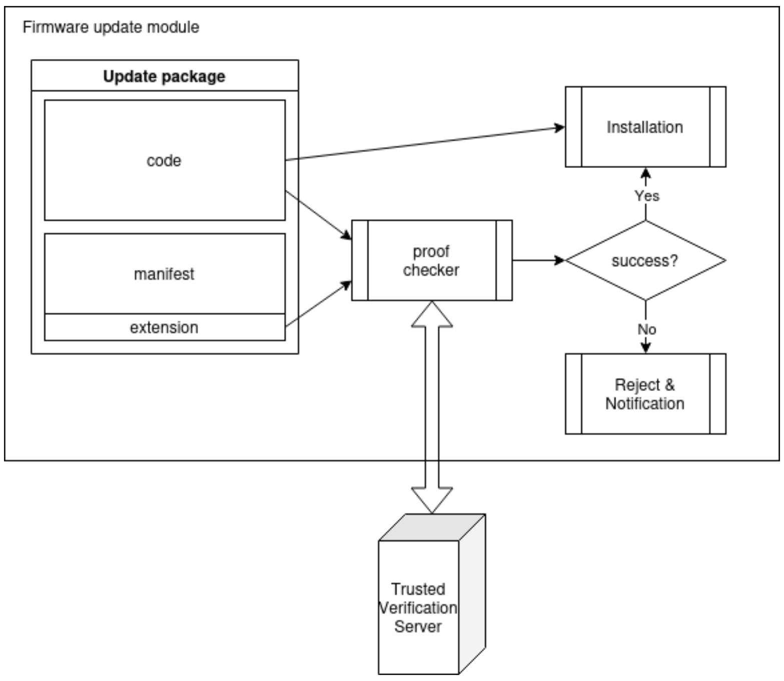


Figure 20: Workflow for update installation.

The corresponding workflow for the installation of an update package is depicted in Figure 20.

At update reception time, the proofs contained in the manifest are extracted and checked one by one. If all the proofs are successfully verified, the update is installed; otherwise it is rejected as invalid, and an appropriate notification is sent.

To generate and verify these formal proofs we can adopt the framework proposed in [160], where the security properties are expressed using the well-known *Linear Temporal Logic* (LTL) formal language. The verification of such a property involves, in general, two different tasks:

- as a first step, a set of *proof obligations* must be discharged by proving that they are valid;
- as a second step, a *formal proof* (typically obtained by resolution) must be checked for (syntactic) validity.

The first step requires a trusted SAT solver and is, in general, more computationally expensive than the second (which may be performed on device). This step can be offloaded to a trusted server, which represents the only addition to the regular SUIT setup that is necessary in this scheme.

It is worth noting that the proof obligations that appear in the first step are generated only when the property to be verified is expressed by a genuinely temporal formulas; simpler properties (e.g. invariants) can be checked much more efficiently.

As an example of a security property to be checked at this stage we can mention the verification of the correctness of the new configuration of the CROSSCON Hypervisor to be deployed in the update. This verification must involve the assumptions detailed in the Deliverable D2.2, as for instance the disjointness of the memory regions managed by the separation kernel.

3.4.5 Implementation and Integration with DevSecOp

In the realm of firmware development and embedded systems, implementing a DevSecOps workflow is essential for maintaining security and quality standards throughout the development lifecycle. This workflow is particularly tailored to accommodate both software and firmware development processes, including specialized steps for hypervisor configuration verification and physical testing on a hardware testbed.

The planned DevSecOps workflow is the following:

- Triggering Analysis with New Commits/Releases:
  - The workflow is initiated automatically with each new commit or release, integrating continuous integration and deployment practices.
- Static Analysis on Source Code:
  - SAST Tools: Tools such as Spotbugs, Semgrep, etc., are employed for SAST to detect vulnerabilities and coding issues in the source code.
  - BAO Config Checker: As BAO is a hypervisor used in the system, the BAO config checker specifically verifies its configuration for any security problems. This step is crucial for ensuring the hypervisor layer's integrity and security, which is foundational in embedded systems.
- Firmware Image Analysis (Optional for Unavailable Source Code):
  - When the source code for certain parts of the firmware isn't available, tools like BugProve are utilized for firmware image analysis, identifying potential vulnerabilities at the firmware level.
- Change Analysis with DeltAICert:
  - Requirement Validation Tests: DeltAICert is utilized to run requirement validation tests. These tests ensure that changes or new additions meet the specified requirements and do not introduce any regressions or new vulnerabilities.
  - Physical Testing on Testbed: Beyond virtual testing, tests are conducted physically on the testbed, which includes Raspberry Pi devices and FPGAs, to verify real-world performance and stability.
- Firmware Manifest Inclusions:
  - SBOM: The firmware manifest comprises an SBOM to provide transparency about all components, libraries, and dependencies.
  - Formal Proof of Functionality: A formal proof of the firmware's functionality is included to certify its performance and security.
  - Certification: Successful test completion leads to certification, denoting compliance with quality and security benchmarks.
- Firmware Signing and Deployment:

- The firmware undergoes digital signing post-verification and certification, safeguarding its authenticity and integrity.

▶ This signed firmware is then uploaded to the update server for deployment.

3.4.6 Secure Cross-Compilation for TA

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 categorized 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 array accesses 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 TAs, 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, potentially allowing an attacker to exploit it.

In order to mitigate this risk, we propose a secure compiler for the C programming language, leveraging the well-known LLVM compiler infrastructure [161]. The LLVM infrastructure is rich in tools and features, among which there is an expressive Intermediate Language (IR) that can be used for analysis and instrumentation of the software. Specifically, the IR allows multiple programming language to share a single semantic, which can then be optimised and converted for the desired target binary.

Our secure compilation is based on an enhanced pointer colouring schema, leveraging the Memory Tagging Extension (MTE) and any future RISC-V tagging proposal. MTE is an Armv9 feature that is likely to be widely adopted in the future years. This technique introduces tags, a.k.a. colours, to memory with a key-lock mechanism: each memory location is associated with a tag which is also embedded in every pointer to that location. In principles, this ensures that a pointer can only access the memory location it originally points to, therefore becoming resilient to corruption. In practice, the limited number of tags makes this approach suitable for a weaker probabilistic form of spatial memory safety. To build on top of MTE, we enhance the state-of-the-art stack colouring techniques with a novel partial tag integrity technique that boosts its security guarantees.

The secure compilation comprises three different stages, depicted in Figure 21, enabled by LLVM. The first stage takes as input the source code of a TA, it analyses it and then produces the corresponding IR representation.

During the second stage the IR is instrumented with new instructions that apply our secure pointer colouring schema.

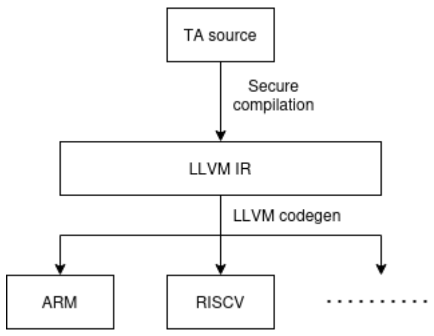


Figure 21: Secure cross-compilation workflow.

In the third and final stage we rely on the LLVM code generator to lower the IR to machine code for the selected platform.

3.5 CROSSCON Bare-Metal TEE

In computer science, bare-metal is often used to refer to systems where the application run without underlying OS, i.e. directly on the hardware. If, on the one hand, this allows a boost in the performance of higher-end systems, on the other hand it might be the only viable options on very low-end devices, e.g., microcontroller units (MCUs). The IoT is teemed with such devices whose constrained hardware cannot support fully fledged OSs. Therefore, we can refer to bare-metal devices as a restricted class of devices that are designed to run applications on the "bare-metal" hardware, i.e. without the presence of an underlying kernel/OS. Typically, these applications are in charge of configuring the hardware of the device as well as performing their intended operations.

Bare-metal devices are widely employed in both industry and research, because of their cost-effectiveness and power-efficiency. However, the features offered by such devices are very limited, often forsaking memory virtualisation and caching. Notably, the absence of common features also impacts the security capabilities of these devices. Bare-metal systems often lack the advanced security features present on higher-end devices, e.g. TPM, MMU and hardware TEEs. These features often play a crucial role for the establishment of a secure environment, but incur sensible cost and increase the complexity of the system. For this reason, the security of bare-metal devices is entrusted primarily to the applications deployed on them.

One of the goal of the CROSSCON project is the security of a wide spectrum of devices, comprising both high- and low-end systems. While the CROSSCON Hypervisor can be deployed on high-end devices to instantiate common trusted OSes, being an hypervisor prevents it from being compatible with the bare-metal devices. Therefore, CROSSCON propose a bare-metal TEE to allow bare-metal systems to interact securely with the rest of the CROSSCON stack.

3.5.1 Review of Bare-Metal Requirements and Platforms

Although these devices are not particularly security-capable, they still require adequate security guarantees when employed in critical environments. For this reason, CROSSCON aims at securing a wide range of devices, including the bare-metal ones. If, on the one hand, these devices can be secured even without the presence of dedicated security hardware, on the other hand, this entails a considerable effort and leads to slightly weaker security guarantees. Specifically, the constrained resources of these devices make the security operation costly and not always feasible. Nevertheless, there is a limited yet comprehensive set of security features that CROSSCON can introduce.

3.5.1.1 Requirements

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 a software-based solutions. Nevertheless, the TEE must satisfy a set of requirements in order to be considered secure. The first one is *memory isolation*, a property that 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. The second main requirements is *inter-domain communication*: the TEE and the untrusted software must be able to communicate. In practice, this requires that the TEE exposes a set of APIs that can be called by the untrusted software. On the contrary, the TEE has usually complete freedom over the untrusted software, and it can call it however it sees fit. The third key security property is *privilege isolation*, which ensures that the untrusted software cannot execute arbitrary operations with the privileges of the TEE. Without such a guarantee, the untrusted software could re-configure the TEE and void the security it provides.

3.5.1.2 Platforms

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 two different architectures: MSP430 and Armv7-M. Notably, these two architectures are of different classes, with a distinct set of security features available. Specifically, Armv7-M offers: (i) two privilege levels, (ii) an MPU, (iii) memory in the order of MB. On the other hand, the MSP430 offers no MPU, no privilege levels and only hundreds of KB of memory.

3.5.2 State of the Art

Security on lower-end devices is a topic discussed both in the industry and in the literature, with solutions that vary in the threat model they assume, in the security guarantees they provide and in the class of devices they target. We will analyse the available security solutions, separating the industrial ones from the literature ones. Although the line that separates them can be blurred, we define as industrial solutions the ones that have been fully implemented and are available to the public in an easy and accessible way. On the contrary, we define literature solutions as those that either lack a full implementation or that are not easily deployable on the devices.

Industry solutions: The industry is rich in solutions that boast the security guarantees of bare-metal devices. These solutions are often offered by the manufacturers of the devices, as a mean to extend the security of their own product without incurring additional hardware costs. However, there are also solutions offered by third parties to offer a more general approach compatible with devices of multiple vendors and manufacturers. The common denominator of these approaches is their software nature: they don't require modifying or extending the MCU hardware, but rather rely on software. One of the most common tools to add security to applications in real-world contexts are the RTOS. These are lightweight OSs that focus on a limited number of features in order to be compatible with constrained devices. These features - among which we usually find thread management, semaphores, mutexes and other resources useful in managing the execution of small applications - are usually exposed with APIs via a Hardware Abstraction Layer (HAL). These APIs allow the application developers to explicitly configure the hardware of the device. Some RTOS also take security into account, providing implicit or explicit security services to the application developers. One popular example is Mbed OS, an open source RTOS distributed by Arm that supports both complex and bare-metal contexts. Security wise, Mbed OS offers a few security services such as secure communication, cryptographic libraries and a secure partition manager. A similar approach is taken by Nuttx and FreeRTOS, distributed by Apache and Amazon respectively. Both are RTOS whose main focus is providing a feature-rich HAL to the application developers, comprising cryptographic libraries that can be used to establish some degree of security. Contrarily to Mbed OS, Nuttx and FreeRTOS are not provided by an MCU manufacturer and are therefore compatible with a wider gamma of architectures. Other open-source and cross-compatible RTOSes are offered by Segger and the Zephyr project, both of which are security-oriented OSes that offer more comprehensive APIs that can be used by the application developers to granularly configure the security of the MCU, e.g. by specifying security regions with the MPU. If, on the one hand, all of these approaches offer security services and primitives to instil trust in the system, they must be explicitly configured by the developers (an error prone process) and can be limited in the security isolation they provide. Furthermore, they cannot be easily extended to add security features or services, focusing on more canonical RTOS features rather than security. A slightly different approach was taken by HEX-Five security with their Multizone solution. Rather than offering an entire RTOS, Multizone is a trusted firmware highly focused on security. It defines a rich set of security primitives, including a dedicated TEE and secure boot, that can be integrated with bare-metal applications and FreeRTOS.

Literature solutions: Contrarily to the commercial solutions, the literature solutions can be of different types: software-based or hardware-based. Both types usually offer stronger security guarantees, usually trying to meet those of the higher-end devices, while adding costs in terms of performance or hardware extensions. Software-based solutions are similar to commercial solutions, for they leverage the existing hardware on the device. However, these solutions try to broaden the attacker model and extend the security guarantees to the detriment of performance. On the contrary, hardware solutions strive to maintain the original performance of the MCU by adding HW or extending the existing one. If, on the one hand, the extra HW makes this solution unsuitable for most real-world scenarios, on the other hand, it helps in guiding the development of new technologies. A prominent example of a hardware-based solution was proposed by Sancus, which implements memory isolation, remote attestation and secure communication on constrained devices. To do so, Sancus extends the instruction set of the hardware CPU with additional instructions that take some security actions based on the value of the PC (program counter), i.e. on which program is being executed. A different approach was proposed by TrustLite and TyTAN, which extend the MPU and the exception engine to make them execution-aware, i.e. to allow multiple applications to be executed at the same time. Contrarily to Sancus, TrustLite and TyTAN support extensions in the form of security services that can enhance the security guarantees of the board. While these techniques try to create a security infrastructure, there are others that instead focus on a single security service. Two examples are SMART and VRASED, both of which focus on implementing a Remote Attestation schema on a slightly modified hardware. SMART enhances the hardware with a Key Access Control and Rom Execution Control capabilities, while VRASED only introduces the Key Access Control and some ROM. All of these solutions are efficient yet quite impractical: extending the hardware is not feasible in many scenarios, among which in the industry where off-the-shelves devices are employed. To fix this gap, several researchers have proposed software-based solutions that, leveraging the existing hardware, implement some security primitives with code only. An example is Harbor that enhances the security of a dedicated IoT OS with a software MMU, that the OS can use to manage the running applications. Leveraging both a run-time environment and a modified toolchain, Harbor achieves good security guarantees and a feature rich environment. Targeting more constrained devices there are solutions such as SuV, which instrument the code of applications running on Harvard-based MCUs. SuV focusing on isolating the memory of a single application, offering rather weak security guarantees and a limited feature-set.

3.5.3 Baremetal TEE

In order to meet the security requirements of CROSSCON, we propose two different bare-metal TEEs: the BareTEE-noMPU and the BareTEE-MPU. From a high perspective, both TEEs are a bridge between the CROSSCON Hypervisor and the bare-metal devices. Given the constrained resources of the latter,

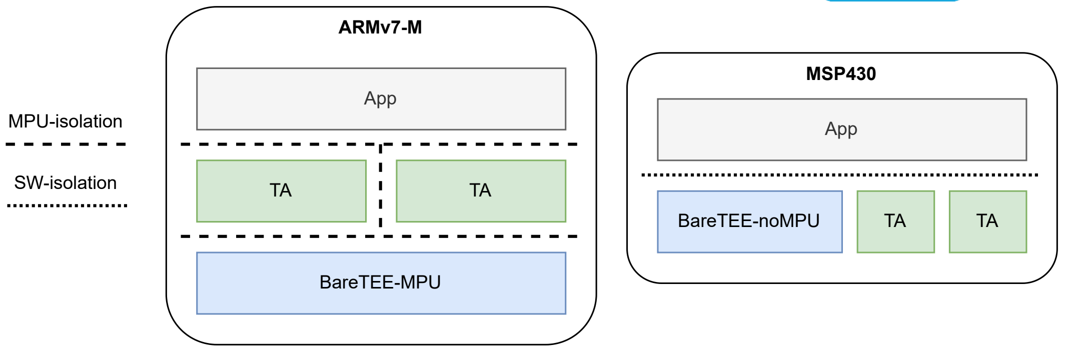


Figure 22: High-level comparison between the memory isolation provided by the two different bare-metal TEEs.

running the full CROSSCON stack is infeasible, reason for which the BareTEE strives to supply a reduced yet comprehensive set of security features to satisfy the bare-metal requirements: memory isolation, privilege separation and cross-domain intercommunication. Depending on the available resources on the target device, we can either deploy the BareTEE-MPU or the BareTEE-noMPU. Specifically, in the presence of an MPU we can use the former, in the absence of it the latter. As part of the CROSSCON project we provide two prototypes: we implement the MPU version for the Armv7-M architecture and nonMPU 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 22 shows the difference between the two versions, which will be highlighted in the following paragraphs.

3.5.3.1 BareTEE-MPU

The MPU is a hardware component that allows basic security features: memory protection. If, on the one hand, such a component is not as flexible as the MMU, on the other hand it still allows enables a good degree of security on the memory. In particular, the MPU defines 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 does not enforce the concept of memory owner. Consequently, defining different access privileges for different entities sharing the same system, e.g. the application and the TEE, is more challenging. Nevertheless, we propose a fully fledged TEE for bare-metal devices with an MPU, providing an implementation for the Armv7-m architecture.

Memory isolation: We define three different entities: CA, Trusted Application (TA) and trusted OS. The trusted OS 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 22). Notably, the memory isolation goes only one direction, with the TA that can access the memory of the CA, and the OS that can access the memory of the entire system, but not vice versa. Furthermore, on top of the vertical isolation, the OS enforces an horizontal separation between the different TAs, preventing them from accessing each others' memory. The memory isolation is enabled by the MPU, which need to be configured dynamically on each domain switch. Specifically, upon execution of any of the three entities, the trusted OS configures the MPU accordingly to enforce our isolation. In details:

- **CA Execution**: whenever the user application is being executed, the MPU is configured to only allow access to the application memory. Accesses to the memory of the TAs and of the OS is prevented.
- **TA execution**: whenever a security service from a TA is invoked, the MPU is configured to allow access to the specific TA memory and to all of the Application memory. Memory accesses to the other TAs' memory and to the OS's memory are disabled.

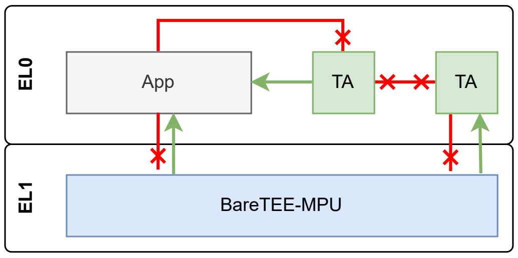


Figure 23: Memory isolation enforced between the three entities on a BareTEE-MPU system: Application, Trusted Applications and trusted OS. Each arrow symbolises the access capabilities of an entity over the memory of the pointed entity.

- **OS execution**: whenever the trusted OS is run, the MPU is configured to allow access to the entire memory.

Figure 23 summarises the memory access rights of each component.

Privilege separation: Armv7-m supports two privilege levels: EL0 and EL1, also known as unprivileged and privileged, respectively. On an 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 EL1 level for our trusted OS and share the EL0 between TAs and CA. This ensures that no entity other than the trusted OS can perform privileged operations without its consent. By managing the MPU 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 in EL0. trusted OS 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 Section 3.1.5. This set of APIs, which facilitate the interaction between the three software entities, is managed by the trusted OS. While our OS 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 trusted OS, that parses the exception and routes the request to the desired functionality.

3.5.3.2 BareTEE-noMPU

In the absence of the basic security hardware, e.g. the MPU, we must resort to an extremely compact and stripped down TEE with only the most basic security services. The goal is to have a solution that is both compatible with the use cases of such constrained devices, while allowing some degree of protection in line with CROSSCON requirements.

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 unstrusted application, whose malicious behaviour is confined in the unsecure 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

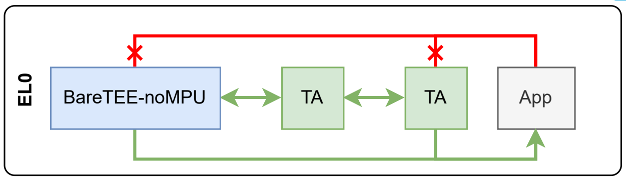


Figure 24: Isolation enforced between the three entities on a BareTEE-noMPU system: Application, Trusted Applications and trusted OS. Each arrow symbolises the access capabilities of the entity over the memory of the pointed entity.

resort to virtualisation. In particular, we replace them 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 25 shows the instrumentation/virtualisation 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. One 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 leveraged to disrupt the functionalities of the TEE when triggered during its execution. To prevent that, we must supervise the handling of interrupts by making sure that the TEE is ready to give up control to the application.

Privilege separation: The MSP430 architecture does not provide different privilege levels, thus making the separation of privileges a challenging task. To supply the lack of privileges, we leverage a code verification technique. Specifically, the trusted OS inspect the binary of the deployed CA to detect any attempt of executing 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 nonMPU BareTEE employs a set of APIs to allow communication between the trusted OS/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 simplify them. This allow us to offer strong security guarantees without increasing the complexity of the code. The TEE Abstraction Model can still be mapped the APIs of this implementation, although with some limitations.

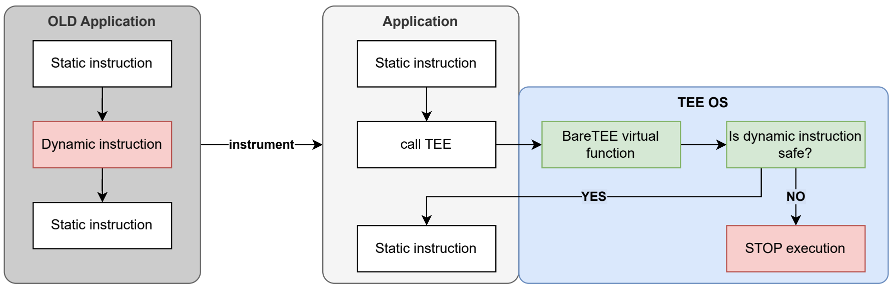


Figure 25: Representation of the instrumentation/virtualisation technique of BareTEE-noMPU.

4 Conclusions

This document has provided insights into the research and development efforts within the CROSSCON project, contributing significantly to the advancement of secure and trusted execution environments.

We've performed a platform analysis and selection, followed by an extensive investigation into various aspects of TEEs, section 2. After, we've document progress in each of the CROSSCON WP3 tasks. In TEE Isolation and Abstraction, section 3.1 we've delved into critical areas such as TEE technologies, vulnerabilities, isolation features, and abstraction proposals. Next, in section 3.2 we've selected Bao as hypervisor to be used as a basis for CROSSCON Hypervisor, and provided design details regarding the developed hypervisor features: dynamic VM creation and per VM TEE service support.

Then, we've discussed the development of new trusted services, section 3.3, including PUF-based authentication, remote attestation, FPGA-related services, behavioral-based services, and control flow integrity services. Regarding the CROSSCON TEE Toolchain, section 3.4, we've examined covering existing IoT update mechanisms, integration requirements with DevSecOps platforms, and design aspects of secure update mechanisms.

Finally, we've detailed CROSSCON Bare-Metal TEE, section 3.5, reviewing requirements, platforms, and implementation details. We've discussed the state-of-the-art approaches and variations of Bare-Metal TEE, including both MPU and non-MPU variants.

References

[1] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro 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] Arm. *GlobalPlatform API Archives.* URL: https://globalplatform.org/specs-library/.

[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*. 2022. arXiv: 2205.12742 [cs.CR].

[7] Priyadarshini Patil, Prashant Narayankar, DG Narayan, and S Md Meena. ``A comprehensive evaluation of cryptographic algorithms: DES, 3DES, AES, RSA and Blowfish''. In: *Procedia Computer Science* 78 (2016), pp. 617–624.

[8] S. Pinto and N. Santos. ``Demystifying Arm TrustZone: A Comprehensive Survey''. In: *ACM Comput. Surv.* (2019).

[9] Arm. *Arm Architecture Reference Manual for A-profile architecture*. URL: https://developer.arm.com/documentation/ddi0487/ja/.

[10] Arm. *Arm Firmware Framework for Arm A-profile*. URL: www.developer.arm.com/documentation/den0077/latest.

[11] 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.

[12] Arm. *TrustZone for Cortex-M.* URL: https://www.arm.com/technologies/trustzone-for-cortex-m.

[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] RISC-V Foundation. *Privileged Architecture v1.12, Ratified*. www.github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12. 2021.

[17] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. ``CoVE: Towards Confidential Computing on RISC-V Platforms''. In: *Proc. of ACM International Conference on Computing Frontiers*. 2023.

[18] Qualcomm. *War of the Worlds - Hijacking the Linux Kernel from QSEE*. URL: https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html.

[19] Taras A Drozdovskyi and Oleksandr S Moliavko. ``mTower: Trusted Execution Environment for MCU-based devices''. In: *Journal of Open Source Software* 4.40 (2019), p. 1494.

[20] Samsung. *SAMSUNG TEEGRIS.* URL: https://developer.samsung.com/teegris/overview.html.

[21] *CVE database.* URL: https://cve.mitre.org.

[22] Qualcomm. *Security Bulletin.* URL: https://docs.qualcomm.com/product/publicresources/securitybulletin/.

[23] AMD. *AMD Product Security.* URL: https://www.amd.com/en/resources/product-security.html.

[24] Samsung. *Security Updates.* URL: https://security.samsungmobile.com/securityUpdate.smsb.

[25] NVIDIA. *Product Security.* URL: https://www.nvidia.com/en-us/security/.

[26] *Common Vulnerability Scoring System v3.0: Specification Document.* URL: https://www.first.org/cvss/specification-document.

[27] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. ``vTZ: Virtualizing ARM TrustZone''. In: *Proc. of USENIX Security*. 2017.
[28] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek. ``PrOS: Light-weight Privatized Secure OSes in ARM TrustZone''. In: *TMC* (2019).
[29] Seung-Kyun Han and Jinsoo Jang. ``MyTEE: Own the Trusted Execution Environment on Embedded Devices.'' In: *Proc. of NDSS*. 2023.
[30] Borna Blazevic, Michael Peter, Mohammad Hamad, and Sebastian Steinhorst. ``TEEVseL4: Trusted Execution Environment for Virtualized seL4-based Systems''. In: *Proc. of RTCSA*. 2023.
[31] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. ``uTango: an open-source TEE for IoT devices''. In: *IEEE Access* 10 (2022), pp. 23913–23930.
[32] Arm Ltd. *Arm System Memory Management Unit Architecture Specification SMMU architecture version 2*. 2016.
[33] Arm Ltd. *Arm System Memory Management Unit Architecture Specification SMMU architecture version 3*. 2023.
[34] RISC-V Foundation. *RISC-V*. https://www.riscv.org/.
[35] Andrew Waterman1, Krste Asanovi, John Hauser. *The RISC-V Instruction Set Manual - Volume II: Privileged Architecture*. 2021.
[36] RISC-V Task Group. *RISC-V Platform-Level Interrupt Controller Specification*. 2023.
[37] John Hauser. *The RISC-V Advanced Interrupt Architecture*. 2023.
[38] IOMMU Task Group. *RISC-V IOMMU Architecture Specification*. 2023.
[39] Dong Du, RISC-V SPMP Task Group. *RISC-V S-mode Physical Memory Protection (SPMP)*. 2023.
[40] Francesco Paci, Davide Brunelli, and Luca Benini. ``Lightweight IO virtualization on MPU enabled microcontrollers''. In: *ACM SIGBED Review* 15.1 (2018), pp. 50–56.
[41] 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, pp. 1485–1492.
[42] 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)*. IEEE. 2019, pp. 293–304.
[43] 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)*. IEEE. 2018, pp. 62–74.
[44] Daniel Gruss. ``Software-based microarchitectural attacks''. In: *it-Information Technology* 60.5-6 (2018), pp. 335–341.
[45] Jakub Szefer. ``Survey of microarchitectural side and covert channels, attacks, and defenses''. In: *Journal of Hardware and Systems Security* 3.3 (2019), pp. 219–234.
[46] 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* 8 (2018), pp. 1–27.
[47] 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 (2021), pp. 1–15.
[48] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. ``BUSted!!! Microarchitectural side-channel attacks on the MCU bus interconnect''. In: (2023).
[49] Abel Gordon et al. ``ELI: Bare-Metal Performance for I/O Virtualization''. In: *SIGPLAN Notices* (2012).
[50] Giovani Gracioli et al. ``A Survey on Cache Management Mechanisms for Real-Time Embedded Systems''. In: *ACM Computing Surveys* (2015).
[51] 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.

[52] Ralf Ramsauer et al. ``A Novel Software Architecture for Mixed Criticality Systems''. In: *Digital Transformation in Semiconductor Manufacturing*. 2020.

[53] 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.

[54] 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.

[55] 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.

[56] *jailhouse-RT: Bu-maintained version of the jailhouse partitioning hypervisor with real-time features*. URL: https://github.com/rntmancuso/jailhouse-rt.

[57] Alessandro Biondi et al. ``SPHERE: A Multi-SoC Architecture for Next-Generation Cyber-Physical Systems Based on Heterogeneous Platforms''. In: *IEEE Access* (2021).

[58] 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.

[59] Giulio Corradi. ``Xen on Arm: Real-Time Virtualization with Cache Coloring''. In: *Proc. of Embedded World Conference*. 2020.

[60] *Zephyr project*. Feb. 2023. URL: https://www.zephyrproject.org/.

[61] 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.

[62] Bruno Sa et al. ``A First Look at RISC-V Virtualization from an Embedded Systems Perspective''. In: *IEEE Transactions on Computers* (2021).

[63] Gerwin Klein et al. ``SeL4: Formal Verification of an OS Kernel''. In: *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[64] Gernot Heiser. *The seL4 Microkernel: An Introduction*. The seL4 Foundation. 2020.

[65] Gerwin Klein et al. ``Formally Verified Software in the Real World''. In: *Communications of the ACM* (2018).

[66] Jesse Millwood et al. ``Performance Impacts from the seL4 Hypervisor''. In: *Proc. of the Ground Vehicle Systems Engineering and Technology Symposium*. 2020.

[67] 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.

[68] Qian Ge et al. ``Time Protection: The Missing OS Abstraction''. In: *Proc. of European Conference on Computer Systems (EuroSys)*. 2019.

[69] 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.

[70] Gerwin Klein et al. ``Comprehensive Formal Verification of an OS Microkernel''. In: *ACM Transactions on Computer Systems* (2014).

[71] Gernot Heiser et al. ``Towards Provable Timing-Channel Prevention''. In: *ACM SIGOPS Operating Systems Review* (2020).

[72] Gernot Heiser et al. ``Can We Put the "S" Into IoT?'' In: *Proc. of IEEE World Forum on Internet of Things*. 2022.

[73] José Martins and Sandro Pinto. ``Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems''. In: *Proc. of RTAS*. IEEE. 2023.

[74] An Braeken. ``PUF based authentication protocol for IoT''. In: *Symmetry* 10.8 (2018), p. 352.

[75] Wenjie Che, Fareena Saqib, and Jim Plusquellic. ``PUF-based authentication''. In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2015, pp. 337–344.

[76] 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*. Springer. 2012, pp. 142–158.

[77] Seungyong Yoon, Byoongkoo Kim, Yousung Kang, and Dooho Choi. ``Puf-based authentication scheme for iot devices''. In: *2020 international conference on information and communication technology convergence (ICTC)*. IEEE. 2020, pp. 1792–1794.

[78] 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, pp. 9–14.

[79] 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*. Springer. 2012, pp. 374–389.

[80] 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)*. IEEE. 2019, pp. 239–244.

[81] 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)*. IEEE. 2018, pp. 1–5.

[82] Jin Wook Byun. ``An efficient multi-factor authenticated key exchange with physically unclonable function''. In: *2019 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE. 2019, pp. 1–4.

[83] 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* 6 (2020), pp. 3–13.

[84] Raffaele Pugliese, Stefano Regondi, and Riccardo Marini. ``Machine learning-based approach: Global trends, research directions, and regulatory standpoints''. In: *Data Science and Management* 4 (2021), pp. 19–29.

[85] 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)* 22.2 (2021), pp. 1–20.

[86] 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)*. IEEE. 2016, pp. 100–103.

[87] Yun-Hsin Chuang and Chin-Laung Lei. ``PUF based authenticated key exchange protocol for IoT without verifiers and explicit CRPs''. In: *IEEE Access* 9 (2021), pp. 112733–112743.

[88] 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)*. IEEE. 2019, pp. 58–63.

[89] Zhao Huang and Quan Wang. ``A PUF-based unified identity verification framework for secure IoT hardware via device authentication''. In: *World Wide Web* 23.2 (2020), pp. 1057–1088.

[90] Karim Lounis and Mohammad Zulkernine. ``T2T-MAP: A PUF-based thing-to-thing mutual authentication protocol for IoT''. In: *IEEE Access* 9 (2021), pp. 137384–137405.

[91] Urbi Chatterjee, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. ``A PUF-based secure communication protocol for IoT''. In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.3 (2017), pp. 1–25.

[92] 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* 80 (2022), p. 103766.

[93] 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)*. IEEE. 2018, pp. 160–165.

[94] 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* 22.18 (2022), p. 7075.

[95] 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)*. IEEE. 2019, pp. 195–200.

[96] Muhammad Naveed Aman, Kee Chaing Chua, and Biplab Sikdar. ``Physically secure mutual authentication for IoT''. In: *2017 IEEE Conference on Dependable and Secure Computing*. IEEE. 2017, pp. 310–317.

[97] 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, pp. 229–236. DOI: 10.1109/SmartIoT52359.2021.00043.

[98] Pim Tuyls and Lejla Batina. ``RFID-tags for anti-counterfeiting''. In: *Cryptographers' track at the RSA conference*. Springer. 2006, pp. 115–131.

[99] 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)*. IEEE. 2023, pp. 28–35.

[100] 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* 75 (2021), p. 103354.

[101] 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* 30.9 (2017), e3195.

[102] 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*. Springer. 2009, pp. 206–220.

[103] 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)*. IEEE. 2019, pp. 189–194.

[104] Victor Shoup. ``Practical threshold signatures''. In: *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14--18, 2000 Proceedings 19*. Springer. 2000, pp. 207–220.

[105] Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. ``Keyless signatures' infrastructure: How to build global distributed hash-trees''. In: *Nordic Conference on Secure IT Systems*. Springer. 2013, pp. 313–320.

[106] Jie Ding, Mahyar Nemati, Chathurika Ranaweera, and Jinho Choi. ``IoT Connectivity Technologies and Applications: A Survey''. In: *IEEE Access* 8 (2020), pp. 67646–67673. DOI: 10.1109/ACCESS.2020.2985932.

[107] Jeyanthi Hall, Michel Barbeau, and Evangelos Kranakis. ``Detection Of Transient In Radio Frequency Fingerprinting Using Signal Phase''. In: (June 2003).

[108] 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* 61.10 (2023), pp. 110–115. DOI: 10.1109/MCOM.003.2200974.

[109] 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* 219 (2022), p. 109455. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2022.109455. URL: https://www.sciencedirect.com/science/article/pii/S1389128622004893.

[110] 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* 12.5 (2015), pp. 585–596. DOI: 10.1109/TDSC.2014.2366455.

[111] 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, pp. 1700–1708. DOI: 10.1109/INFOCOM.2018.8485917.

[112] 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, pp. 1–4. DOI: 10.1109/WISNET.2019.8711803.

[113] ``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), pp. 1–4379. DOI: 10.1109/IEEESTD.2021.9363693.

[114] Yongsen Ma, Gang Zhou, and Shuangquan Wang. ``WiFi Sensing with Channel State Information: A Survey''. In: *ACM Comput. Surv.* 52.3 (June 2019). ISSN: 0360-0300. DOI: 10.1145/3310194. URL: https://doi.org/10.1145/3310194.

[115] 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*. NIPS'93. Denver, Colorado: Morgan Kaufmann Publishers Inc., 1993, pp. 737–744.

[116] 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, pp. 552–561.

[117] 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, pp. 30–40.

[118] Donatella Granata, Raffaele Cerulli, Maria Grazia Scutella, Andrea Raiconi, et al. ``Maximum flow problems and an NP-complete variant on edge-labeled graphs''. In: *Handbook of combinatorial optimization* (2013), pp. 1913–1948.

[119] Giovanna Kobus Conrado, Amir Goharshady, and Chun Kit Lam. ``The Bounded Pathwidth of Control-flow Graphs''. In: *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2023*. 2023.

[120] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. ``Cfinsight: A comprehensive metric for cfi policies''. In: *29th Annual Network and Distributed System Security Symposium*. NDSS. 2022.

[121] Henrik Theiling. ``Control flow graphs for real-time systems analysis: reconstruction from binary executables and usage in ILP-based path analysis''. PhD thesis. Saarland University, 2002.

[122] David Van Horn and Harry G Mairson. ``Relating complexity and precision in control flow analysis''. In: *ACM SIGPLAN Notices* 42.9 (2007), pp. 85–96.

[123] Liang Xu, Fangqi Sun, and Zhendong Su. ``Constructing precise control flow graphs from binaries''. In: *University of California, Davis, Tech. Rep* (2009), pp. 14–23.

[124] Kailong Zhu, Yuliang Lu, Hui Huang, Lu Yu, and Jiazhen Zhao. ``Constructing more complete control flow graphs utilizing directed gray-box fuzzing''. In: *Applied Sciences* 11.3 (2021), p. 1351.

[125] Andrei Rimsa, José Nelson Amaral, and Fernando MQ Pereira. ``Practical dynamic reconstruction of control flow graphs''. In: *Software: Practice and Experience* 51.2 (2021), pp. 353–384.

[126] 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 (2021), p. 13. DOI: 10.1155/2021/8873195.

[127] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2020. ISBN: 978-1492052043.

[128] 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* 21.13 (2021). ISSN: 1424-8220. DOI: 10.3390/s21134412.

[129] Georgios Kornaros. ``Hardware-Assisted Machine Learning in Resource-Constrained IoT Environments for Security: Review and Future Prospective''. In: *IEEE Access* 10 (2022), pp. 58603–58622. DOI: 10.1109/ACCESS.2022.3179047.

[130] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. ``Survey of intrusion detection systems: techniques, datasets and challenges''. In: *Cybersecurity* 2 (2019), p. 20. DOI: 10.1186/s42400-019-0038-7.

[131] 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* 4 (3 2023), pp. 345–365. DOI: 10.3390/iot4030016.

[132] Conner Bradley and David Barrera. ``Towards Characterizing IoT Software Update Practices''. en. In: *Foundations and Practice of Security*. Ed. by Guy-Vincent Jourdan, Laurent Mounier, Carlisle Adams, Florence Sèdes, and Joaquin Garcia-Alfaro. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 406–422. ISBN: 978-3-031-30122-3. DOI: 10.1007/978-3-031-30122-3_25.

[133] Saad El Jaouhari and Eric Bouvet. ``Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions''. In: *Internet of Things* 18 (May 2022), p. 100508. ISSN: 2542-6605. DOI: 10.1016/j.iot.2022.100508. URL: https://www.sciencedirect.com/science/article/pii/S2542660522000142 (visited on 03/05/2024).

[134] Daniele Perito and Gene Tsudik. ``Secure Code Update for Embedded Devices via Proofs of Secure Erasure''. en. In: *Computer Security – ESORICS 2010*. Ed. by Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou. Vol. 6345. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 643–662. ISBN: 978-3-642-15496-6 978-3-642-15497-3. DOI: 10.1007/978-3-642-15497-3_39. URL: http://link.springer.com/10.1007/978-3-642-15497-3_39 (visited on 03/05/2024).

[135] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. ``Survivable key compromise in software update systems''. In: *Proceedings of the 17th ACM conference on Computer and communications security*. CCS '10. New York, NY, USA: Association for Computing Machinery, Oct. 2010, pp. 61–72. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866315. URL: https://dl.acm.org/doi/10.1145/1866307.1866315 (visited on 03/05/2024).

[136] Dimitris Mbakoyiannis, Othon Tomoutzoglou, and George Kornaros. ``Secure over-the-air firmware updating for automotive electronic control units''. en. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Limassol Cyprus: ACM, Apr. 2019, pp. 174–181. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297299. URL: https://dl.acm.org/doi/10.1145/3297280.3297299 (visited on 03/05/2024).

[137] Hans Chandra, Erwin Anggadjaja, Pranata Setya Wijaya, and Edy Gunawan. ``Internet of Things: Over-the-Air (OTA) firmware update in Lightweight mesh network protocol for smart urban development''. In: *2016 22nd Asia-Pacific Conference on Communications (APCC)*. Aug. 2016, pp. 115–118. DOI: 10.1109/APCC.2016.7581459. URL: https://ieeexplore.ieee.org/abstract/document/7581459 (visited on 03/05/2024).

[138] Krishna Doddapaneni, Ravi Lakkundi, Suhas Rao, Sujay Gururaj Kulkarni, and Bhargav Bhat. ``Secure FoTA Object for IoT''. In: *2017 IEEE 42nd Conference on Local Computer Networks Work-*

shops (LCN Workshops). Oct. 2017, pp. 154–159. DOI: 10.1109/LCN.Workshops.2017.78. URL: https://ieeexplore.ieee.org/abstract/document/8110218 (visited on 03/05/2024).

[139] N. Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. ``ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices''. en. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (Nov. 2018), pp. 2290–2300. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2018.2858422. URL: https://ieeexplore.ieee.org/document/8493602/ (visited on 03/05/2024).

[140] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. ``UpKit: An Open-Source, Portable, and Lightweight Update Framework for Constrained IoT Devices''. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. ISSN: 2575-8411. July 2019, pp. 2101–2112. DOI: 10.1109/ICDCS.2019.00207. URL: https://ieeexplore.ieee.org/abstract/document/8884933 (visited on 03/05/2024).

[141] Google. *Cloud IoT Core*. URL: https://cloud.google.com/iot-core.

[142] Amazon. *AWS IoT Device Management*. URL: https://aws.amazon.com/iot-device-management/.

[143] Texas Instrument Semiconductor. *Over the Air Download (OAD)*. URL: https://software-dl.ti.com/lprf/simplelink_cc2640r2_sdk/1.00.00.22/exports/docs/blestack/html/oad/oad.html.

[144] ARM. *The full IoT stack*. URL: https://www.pelion.com/.

[145] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. *A Firmware Update Architecture for Internet of Things*. RFC 9019. Apr. 2021. DOI: 10.17487/RFC9019. URL: https://www.rfc-editor.org/info/rfc9019.

[146] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. DOI: 10.17487/RFC7228. URL: https://www.rfc-editor.org/info/rfc7228.

[147] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. *A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices*. RFC 9124. Jan. 2022. DOI: 10.17487/RFC9124. URL: https://www.rfc-editor.org/info/rfc9124.

[148] 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 draft-ietf-suit-manifest-25. Work in Progress. Internet Engineering Task Force, Feb. 2024. 101 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-suit-manifest/25/.

[149] U.S. Federal Government. *Improving the Nation's Cybersecurity*. May 2021. URL: https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity.

[150] Marco Patrignani, Amal Ahmed, and Dave Clarke. ``Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work''. In: *ACM Comput. Surv.* 51.6 (2019), 125:1–125:36. DOI: 10.1145/3280984.

[151] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. ``From system F to typed assembly language''. In: *ACM Transactions on Programming Languages and Systems* 21.3 (May 1999), pp. 527–568. ISSN: 0164-0925. DOI: 10.1145/319301.319345. URL: https://dl.acm.org/doi/10.1145/319301.319345 (visited on 02/14/2024).

[152] Gilles Barthe, Tamara Rezk, and Amitabh Basu. ``Security types preserving compilation''. In: *Computer Languages, Systems & Structures* 33.2 (July 2007), pp. 35–59. ISSN: 1477-8424. DOI: 10.1016/j.cl.2005.05.002. URL: https://www.sciencedirect.com/science/article/pii/S1477842405000230 (visited on 02/14/2024).

[153] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. ``Control-flow integrity principles, implementations, and applications''. In: *ACM Trans. Inf. Syst. Secur.* 13.1 (2009), 4:1–4:40. DOI: 10.1145/1609956.1609960.

[154] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. ``Fully abstract compilation to JavaScript''. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '13. New York,

NY, USA: Association for Computing Machinery, Jan. 2013, pp. 371–384. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429114. URL: https://dl.acm.org/doi/10.1145/2429069.2429114 (visited on 02/14/2024).

[155] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. ``The CHERI capability model: Revisiting RISC in an age of risk''. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. ISSN: 1063-6897. June 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201. URL: https://ieeexplore.ieee.org/document/6853201 (visited on 02/14/2024).

[156] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. ``Secure Compilation to Modern Processors''. In: *2012 IEEE 25th Computer Security Foundations Symposium*. ISSN: 2377-5459. June 2012, pp. 171–185. DOI: 10.1109/CSF.2012.12. URL: https://ieeexplore.ieee.org/document/6266159 (visited on 02/14/2024).

[157] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. ``Secure Compilation to Protected Module Architectures''. In: *ACM Trans. Program. Lang. Syst.* 37.2 (2015), 6:1–6:50. DOI: 10.1145/2699503.

[158] Martín Abadi and Gordon D. Plotkin. ``On Protection by Layout Randomization''. In: *ACM Trans. Inf. Syst. Secur.* 15.2 (2012), 8:1–8:29. DOI: 10.1145/2240276.2240279.

[159] Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. ``Local Memory via Layout Randomization''. English. In: ISSN: 1063-6900. IEEE Computer Society, June 2011, pp. 161–174. ISBN: 978-1-61284-644-6. DOI: 10.1109/CSF.2011.18. URL: https://www.computer.org/csdl/proceedings-article/csf/2011/05992161/12OmNCcKQAN (visited on 02/14/2024).

[160] Alberto Griggio, Marco Roveri, and Stefano Tonetta. ``Certifying proofs for SAT-based model checking''. In: *Formal Methods Syst. Des.* 57.2 (2021), pp. 178–210. DOI: 10.1007/S10703-021-00369-1. URL: https://doi.org/10.1007/s10703-021-00369-1.

[161] The LLVM Project. *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html.